CLASSES PRÉPARATOIRES

Informatique MP2I

Cours, exercices, travaux pratiques, problèmes et devoirs

Marc de Falco

Un cours figé, remis en ligne pour consultation

Ce livre reproduit le cours que j'ai écrit lorsque j'étais professeur d'informatique en MP2I/MPI. Il n'est plus tenu à jour au fil des programmes.

Il a été **complété et corrigé grâce à Claude** (Anthropic) dans le seul but de le remettre à disposition dans un état propre et consultable. Les passages rédigés ou modifiés par Claude sont signalés par un trait vertical et un symbole dans la marge gauche :

- ✍ en orange (✍) : contenu rédigé par Claude (sections, preuves, corrigés, code) ;
- 🖉 en violet (🖉) : texte de l'auteur corrigé, reformulé ou complété par Claude.

Les corrections ponctuelles (coquilles, mise en forme) ne sont pas signalées.

Ce livre n'a **aucune valeur prescriptive** : ce n'est ni une référence officielle du programme, ni une recommandation pédagogique.

Table des matières

I Programmation 5

1 Introduction à l'informatique 7
2 Introduction à la programmation (impérative) 15
3 Récursivité 47
4 Typage en OCaml 63
5 TP Rappels et types récursifs 67
6 TP Types enregistrements 73
7 Type option en OCaml 81
8 Exceptions en OCaml 85
9 Entrées-Sorties en Python 91

II Programmation en C 93

10 TP Travaux Pratiques d'apprentissage du C 95
11 TP Introduction au C (version notebook) 129
12 TP Tableaux, chaînes et automates de Wolfram (version notebook) 145
13 TP C avancé 155
14 TP DS3 - Examen pratique de C 179

III Structures de données 185

15 Structures de données abstraites et implémentations 187
16 Séquences et leurs implémentations, tableaux, listes chaînées 191
17 TP Tableaux dynamiques 207
18 Piles et files, structures abstraites et implémentations 215
19 TD Piles et files 225
20 TP Piles et files 227
21 Arbres 233
22 TP Arbres - Travaux pratiques 269
23 TP Arbres binaires 291
24 TP Arbres et tries 301
25 TP Arbres binaires de recherche 309
26 TP Tas et files de priorité en OCaml 315
27 TP Arbres binaires en C 321
28 Dictionnaires et Ensembles finis 325
29 TP Arbres rouges et noirs en C 333
30 Graphes 337
31 Exercices Graphes 407
32 TP Graphes 411
33 TP Parcours récursif de graphes 437

IV Algorithmique 445

34 Introduction à l'analyse des algorithmes 447
35 Complexité amortie 477
36 Algorithmique exacte 487
37 Recherche par force brute 495
38 Algorithmes gloutons 515
39 Division en sous-problèmes 535
40 Programmation dynamique 551
41 TP Programmation dynamique - TP 2 (Python) 585
42 Algorithmique des textes 593
43 TP Compression de Huffman 651

V Système 659
44 Gestion de la mémoire dans un programme compilé 661

VI Logique 679
45 Calcul propositionnel 681
46 TP Calcul propositionnel en C 695
47 TP SAT-solver 701

VII Bases de données 705
48 Bases de données 707

VIII Problèmes 717
49 Problème DM - Enveloppe convexe 719
50 Problème Dungeons and Diagrams 731
51 Problème Dungeons and Diagrams - les 64 grilles 759
52 Problème Jeux et nombres 783

IX Devoirs MP2I 2021-2022 785
53 Devoir DS 1 - Programmation OCaml et C 787
54 Devoir DS 2 - Sous-ensembles et points fixes 797
55 Devoir DS 3 - Exponentiation rapide et tri du crêpier 811
56 Devoir DS 3 - Partie pratique (OCaml) 819
57 Devoir DS 4 - Listes chaînées, pavages et backtracking 827
58 Devoir DS 6 - Matrices de Monge et problème du monnayeur 837
59 Devoir DS 8 - Concours blanc - Plus longue sous-suite commune et coloration de graphes 843
60 Devoir Devoir des vacances de Pâques - Dessin de graphes, chasseurs de fantômes, expressions 853
61 Devoir Devoir maison - Lancer de rayons 863

X Devoirs MP2I 2022-2023 871
62 Devoir DS 1 - Récursivité en OCaml 873
63 Devoir DS 2 - À la recherche du M-PIece 879
64 Devoir DS 4 - C et mémoire 897
65 Devoir DS 5 - Listes chaînées 905
66 Devoir DS 6 - Arbres n-aires en C 909
67 Devoir DS 7 - Sac à dos fractionnaire 919

XI Devoirs MPSI, MP et tronc commun 925
68 Devoir DS 1 - Typage, listes et récursivité 927
69 Devoir DS 2 - Autour de la décomposition de Zeckendorf 933
70 Devoir DS 4 - Logique et tas 939
71 Devoir DS 1 - Nombres narcissiques et inversions de permutations 945
72 Devoir DS 1 - Poteaux télégraphiques 951
73 Devoir Interro 1 - Programmation Python 957
74 Devoir DS 2 - Correction (circulation automobile) 961
75 Devoir Interro de rentrée - Révisions Python 967

XII Annexes 971
76 Complexité et chansons algorithmiques 973

PARTIE I

Programmation

1	Introduction à l'informatique	7
2	Introduction à la programmation (impérative)	15
3	Récursivité	47
4	Typage en OCaml	63
5	TP Rappels et types récursifs	67
6	TP Types enregistrements	73
7	Type option en OCaml	81
8	Exceptions en OCaml	85
9	Entrées-Sorties en Python	91

CHAPITRE 1

Introduction à l'informatique

DANS CE CHAPITRE

1.1 Données, Problème, Algorithme
1.2 Modèle de calcul, Machines, Programmes
1.3 Langages, Compilateur, Interprète
1.4 Paradigmes

> **Note** **Roadmap** :
>
> - tout reprendre. Je suis peu satisfait du résultat.

Dans ce chapitre, on présente de manière très survolée l'informatique. On aura l'occasion de revenir en détail sur certaines notions. L'objectif est avant tout d'avoir une vision claire même si elle est naïve de

- ce qu'est un problème informatique
- ce qu'est un programme
- ce qu'est une machine
- ce que signifie de résoudre informatiquement un problème

Cela permet de répondre à des questions comme :

- qu'est-ce qui permet de dire qu'on programme en OCaml alors qu'on a l'impression de ne faire que des suites de définitions ?
- peut-on se contenter de programmer sans chercher à comprendre ?

Précisons ici qu'on parle d'informatique mais que le terme le plus proche de ce que l'on étudie en anglais est *computer science* par complémentarité avec des notions comme celles de *computer engineering* plus proches de considérations matérielles. Il n'y a pas une *unique* vision de l'informatique mais plutôt une grande richesse de points de vue.

1.1 Données, Problème, Algorithme

1.1.1 Données

Définition 1.1

Une *donnée informatique* est un élément d'information fini qu'on peut stocker ou transmettre.

Selon le niveau d'abstraction auquel on se place, on peut considérer une donnée comme étant une succession de bits, c'est-à-dire de valeurs valant 0 ou 1, ou comme une donnée plus structurée, comme un texte, une image, un graphe, ...

En informatique, on adapte souvent son point de vue sur les notions de base comme celle-ci. Si on fait de la transmission ou de la compression de données, c'est important d'en avoir une représentation la plus primitive possible comme des mots binaires. Mais si on s'intéresse à des notions de chemins dans un graphe, il est plus adéquat de considérer un graphe comme un élément de données même si la question de sa représentation n'est pas immédiate.

1.1.2 Problèmes

Définition 1.2

Un *problème informatique* est un ensemble de questions paramétrées par des données (les *entrées*) et dont les réponses peuvent être

- soit Oui ou Non, auquel cas on parle de problème de *décision*
- ou d'autres données dépendant des entrées : les *sorties*, on parle alors de problème de *recherche* ou de *construction*.

Exemple 1.3

- Étant donné un entier, déterminer s'il est premier ou non.
- Étant donnés deux entiers, calculer leur PGCD.
- Étant donné un graphe pondéré et deux sommets, déterminer un chemin de longueur minimale permettant d'aller de l'un à l'autre.
- Étant donnés un programme informatique et une entrée, déterminer si le programme va s'arrêter ou non.
- Étant donné une image et une base de données de visages, déterminer les personnes présentes sur l'image.

On voit que certains de ces problèmes n'ont pas eu besoin d'attendre l'informatique, comme les problèmes d'arithmétique.

On peut prouver qu'un problème comme le problème de l'arrêt ne pourra jamais être résolu par un ordinateur (on dira qu'il est *indécidable*).

1.1.3 Algorithme

Définition 1.4

Un *algorithme* est un procédé systématique et mécanisable permettant de résoudre un problème informatique.

Ce qu'on signifie ici par le mot *systématique* est que le procédé de calcul est précis et non ambigu et par *mécanisable* qu'il s'agit d'un calcul qui puisse être fait par une *machine* dans un sens à venir.

Une autre manière de voir les problèmes est de dire que ce sont des fonctions mathématiques des entrées vers les sorties. Un algorithme est alors une description de la réalisation d'une telle fonction.

1.2 Modèle de calcul, Machines, Programmes

On donne ici des définitions volontairement naïves et imprécises. Elles pourraient être précisées en fixant un cadre théorique plus conséquent, ce qui est prématuré à ce stade.

Définition 1.5

Un modèle de calcul est un cadre permettant de décrire comment calculer la solution de problèmes en fonction des entrées.

Définition 1.6

Une *machine* est un modèle de calcul défini par un ensemble de *configurations* et des opérations élémentaires, aussi appelées *instructions*, permettant de passer d'une configuration à une autre.

Ces instructions sont regroupées sous la forme d'une suite finie appelée un *programme*.

Pour exécuter un programme, il y a dans la configuration un entier qui s'appelle le *pointeur d'instruction* et qui donne le numéro dans le programme de l'instruction courante. Généralement, on part de la première instruction en passant ensuite, sauf contre-ordre, à l'instruction suivante, jusqu'à la fin du programme.

1.2.1 Exemple : ordinateur

Un ordinateur est naturellement un exemple de machine. Une configuration est la donnée de l'état des registres du processeur, de la mémoire et aussi des périphériques d'entrée-sortie. Un programme est une suite d'instructions exécutées par le processeur.

1.2.2 Exemple : les machines à deux compteurs

Les configurations d'une machine à deux compteurs sont des triplets (A, B, PC) où A et B sont deux compteurs entiers naturels, en général initialisés avec les entrées et dans lesquels on pourra lire les sorties, et PC est le pointeur d'instruction, initialisé à la première instruction.

Un programme est une suite finie et numérotée d'instructions élémentaires parmi les suivantes :

- `INCA` : incrémenter A, c'est-à-dire ajouter 1 à la valeur qu'il contient
- `DECA` : décrémenter A, c'est-à-dire soustraire 1 à la valeur qu'il contient (on supposera qu'on ne l'utilise que si $A > 0$)
- `IFA i j` : sauter à l'instruction numéro i si A est nul, sinon sauter à l'instruction numéro j.
- les instructions `INCB`, `DECB` et `IFB i j` respectives vis-à-vis du compteur B.

Ainsi, on peut considérer le programme suivant

```
1: IFA 6 2
2: DECA
3: INCB
4: INCB
5: IFA 1 1
6: INCA
7: DECA
```

Si on initialise les compteurs (A, B) à $(n, 0)$ et qu'on exécute le programme, on obtient alors $(0, 2n)$ dans les compteurs. Ce programme réalise ainsi un doublement du compteur A et place le résultat dans le compteur B.

Exercice

1. Écrire un programme permettant de passer de la configuration (a, b) à $(0, a + b)$.
2. Écrire un programme permettant de passer de la configuration $(n, 0)$ à $(0, 0)$ si n est pair ou $(0, 1)$ sinon.

Solution

1. Programme :

```
1: IFA 5 2
2: DECA
3: INCB
4: IFA 1 1
5: INCA
6: DECA
```

2. Programme (on retire 2 tant que c'est possible ; si on tombe sur 0 après un seul retrait, n était impair et on place 1 dans B) :

```
1: IFA 7 2
2: DECA
3: IFA 6 4
4: DECA
5: IFA 1 1
6: INCB
7: INCA
8: DECA
```

□

1.2.3 Exemple : le noyau fonctionnel pur d'`OCaml`

On considère ici le noyau `OCaml` avec les entiers, leurs opérations élémentaires, les fonctions, les définitions (récursives ou non) et ce qui correspond à l'évaluation.

Ainsi le terme `OCaml` suivant `fun n -> 2 * n` peut être vu simplement comme prenant en entrée un entier n et renvoyant son double.

Ce qui est important ici, sans qu'on s'attarde sur la définition précise de l'évaluation, c'est de comprendre qu'il s'agit d'un modèle de calcul sans avoir de notion de machine, d'instructions ou de programme.

Bien entendu, quand on interprète ou qu'on compile un programme `OCaml`, cela se passe sur un ordinateur, donc une machine.

On considère que les termes sont les programmes dans un tel langage de programmation. La différence principale entre un programme vu ainsi et un dans le sens usuel, comme un programme Python, c'est que l'on n'a pas conscience de la réalité la machine quand on programme : il n'y a pas de notion d'instructions. Bien sûr, on verra qu'il est possible de retrouver cela dans OCaml et ainsi de se ramener en terrain connu, mais c'est important de comprendre que ce noyau existe indépendamment d'une notion de machine.

Remarque 1.7 C'est Alonzo Church qui a défini ce noyau, qu'on appelle le λ-calcul, en 1933 alors qu'il souhaitait présenter un cadre pour exprimer les fonctions calculables.

1.2.4 Exemple : Jeu de la vie

Le *jeu de la vie* défini en 1970 par John Conway dans le but d'être une récréation mathématique s'est révélé être un modèle de calcul très important.

Il s'agit d'un cas particulier d'une plus grande classe de modèles de calcul appelés les **automates cellulaires**.

Une configuration du jeu de la vie est une grille bidimensionnelle de valeurs booléennes. On appelle chaque case une *cellule* et on dit que le booléen permet de déterminer si elle est *vivante* ou *morte*.

L'évaluation de ce modèle consiste à définir une nouvelle configuration en appliquant la règle suivante pour chaque cellule :

- si la cellule est vivante et a deux ou trois voisines vivantes, elle reste vivante dans la nouvelle configuration, sinon elle meurt.
- si la cellule est morte, elle devient vivante si et seulement si elle a exactement trois voisines vivantes.

Quand on parle de cellules voisines, on fait référence aux huit voisines directes sur la grille (on compte ainsi les diagonales).

Le point essentiel dans cette procédure permettant de passer d'une configuration à une autre est qu'on ne modifie pas la configuration courante, tout se passe comme si chaque cellule était modifiée simultanément.

La configuration initiale est à la fois le programme et ses entrées. Un point critique ici c'est qu'il n'existe pas de notion de fin d'évaluation. En effet, même des configurations très simples n'atteignent pas un point fixe. On devra donc choisir en fonction du problème que l'on cherche à résoudre, comment déterminer la fin du calcul.

1.2.5 Exemple : FRACTRAN

FRACTRAN est un modèle de calcul également inventé par John Conway en 1987.

Un programme FRACTRAN est une suite finie de fractions d'entiers naturels comme :

$$P = \left(\frac{17}{91}, \frac{78}{85}, \frac{19}{51}, \frac{23}{38}, \frac{29}{33}, \frac{77}{29}, \frac{95}{23}, \frac{77}{19}, \frac{1}{17}, \frac{11}{13}, \frac{13}{11}, \frac{15}{2}, \frac{1}{7}, \frac{55}{1}\right)$$

Une entrée est un entier et pour *exécuter* un tel programme, on considère un entier courant n, initialisé à la valeur de l'entrée, et on parcourt les fractions de gauche à droite jusqu'à trouver une fraction $\frac{p}{q}$ telle que $n\frac{p}{q}$ soit un entier. Dans ce cas, c'est la nouvelle valeur de n et on recommence le processus. Sinon, le programme termine.

Exemple 1.8

$(\frac{3}{10}, \frac{4}{3})$ sur l'entrée 14 va s'arrêter tout de suite, sur l'entrée 15, on va avoir 20, 6 puis 8 et s'arrêter.

Exercice

Exécuter P sur l'entrée 2 suffisamment longtemps pour noter les quatre premières puissances de 2 prises par l'entier courant. Attention, cela nécessite sûrement de programmer l'évaluation.

Solution On remarque que les puissances de 2 qui apparaissent sont dans l'ordre : $2^2, 2^3, 2^5, 2^7, 2^{11}, 2^{13}, ...$ Ainsi ce programme énumère les nombres premiers.

□

Comment fonctionne FRACTRAN ? En fait, chaque fraction va être de la forme

$$\frac{p_1^{a_1} \dots p_k^{a_k}}{q_1^{b_1} \dots q_l^{b_l}}$$

où les p_i et les q_j sont des nombres premiers distincts.

Pour qu'on sélectionne cette fraction, il faut que n soit divisible par le dénominateur, donc si on écrit n en produit de facteurs premiers, il faut que les puissances correspondant à chaque q_j soient $\geq b_j$. On va alors les diminuer de b_j puis augmenter les puissances correspondant à chaque p_i de a_i.

Ainsi, si on considère qu'un nombre n s'écrit $n = \prod_{i \in \mathbb{N}} p_i^{a_i}$, avec les a_i presque tous nuls et $p_0 < p_1 < p_2 < ...$ une énumération des nombres premiers, on voit qu'un nombre n'est alors qu'une configuration d'une infinité de compteurs indexés par des entiers et qui sont presque tous à 0. Une fraction est alors une règle de la forme :

$$C[j_1] \geq b_1, ..., C[j_l] \geq b_l \to C[j_1] - b_1, ..., C[j_l] - b_l, C[i_1] + a_1, ..., C[i_k] + a_k$$

où on a nommé $C[0], C[1], ...$ les compteurs, $p_1 = p_{i_1}, ..., p_k = p_{i_k}$ et $q_1 = p_{j_1}, ..., q_l = p_{j_l}$.

FRACTRAN est donc une sorte de machine à compteurs dont les instructions sont exécutées comme dans un filtrage `OCaml` en cherchant de gauche à droite la première règle qui s'applique. Le point le plus délicat c'est que la condition permettant d'appliquer une règle est destructrice : on divise, et on ne peut pas utiliser le même compteur pour la condition et pour le résultat, car sinon on simplifierait la fraction.

Remarque 1.9 **Comment fonctionne le programme P ?**

Pour cela, on va réécrire P en suivant la syntaxe précédente, comme on a forcément des décréments de compteurs correspondant à la condition d'application, il est inutile de les répéter et on écrira juste

```
2:-2 3:-2 |> 1:+2 0:+1
```

pour la fraction $\frac{18}{1225} = \frac{2.3^2}{5^{2.7^2}}$ car $p_0 = 2, p_1 = 3, p_2 = 5$ et $p_3 = 7$. Par souci de lisibilité, on peut de plus nommer les compteurs par des lettres en commençant à a et donc écrire

```
c-2 d-2 |> b+2 a+1
```

Comme les règles de P ne font intervenir que des $+1$ et des -1, on pourra encore abréger en écrivant `a-` plutôt que `a-1` et `b+` plutôt que `b+1`.

Ainsi, P devient la suite de règles suivantes (numérotées pour y faire référence ensuite).

```
 0: d-f- |> g+
 1: c-g- |> a+b+f+
 2: b-g- |> h+
 3: a-h- |> i+
 4: b-e- |> j+
 5: j-   |> d+e+
 6: i-   |> c+h+
 7: h-   |> d+e+
 8: g-   |>
 9: f-   |> e+
10: e-   |> f+
11: a-   |> b+c+
12: d-   |>
13:      |> c+e+
```

On commence avec la configuration $n = 2 = 2^1$ donc uniquement 1 dans le compteur `a` et 0 dans les autres. On notera $a : 1$ cette configuration.

On remarque donc que si on a la configuration $a : n - 1$ on va appliquer la règle 11 jusqu'à obtenir la configuration $b : n - 1, c : n - 1$

Ensuite, on applique la règle 13 puis successivement les règles 4 et 5 jusqu'à être dans la configuration $c : n, d : n - 1, e : 1$ La règle 10 permet de passer à $c : n, d : n - 1, f : 1$

On va effectuer une succession de règles afin de déterminer si $n - 1$ divise n. En fait, cette succession de règles va déterminer si d divise c. Imaginons donc qu'on a la configuration $c : n, d : m, f : 1$ on applique successivement les règles 0 et 1 pour obtenir la configuration $a : m, b : m, c : n - m, f : 1$ La règle 9 s'applique alors pour passer à $a : m, b : m, c : n - m, e : 1$ et on effectue alors la succession des règles 4 et 5 qu'on a déjà vue pour passer à $a : m, c : n - m, d : m, e : 1$

Comme il n'y a plus rien dans `b` c'est la règle 10 qui s'applique et on repasse à $a : m, c : n - m, d : m, f : 1$ On repasse alors dans les règles 0 et 1 jusqu'à obtenir $a : 2m, b : m, c : n - 2m, f : 1$ et ainsi de suite. Pour l'arrêt, deux cas peuvent alors se produire :

- Soit on a vidé `c` en effectuant la soustraction de m, ce qui correspond au cas où m ne divise pas n, et alors la règle 1 ne pourra plus s'appliquer car elle nécessite $c \geq 1$. Si $n = qm + r$ avec $0 < r < m$, on aboutit alors avec une dernière application de 0 en $a : n, b : r, d : m - r - 1, g : 1$ Comme la règle 1 ne s'applique plus, on passe alors à la règle 2 vers $a : n, b : r - 1, d : m - r - 1, h : 1$ on enchaîne alors les règles 3 et 6 jusqu'à obtenir la configuration $b : r - 1, c : n, d : m - r - 1, h : 1$ On ne peut alors qu'appliquer la règle 7 qui permet de passer à $b : r - 1, c : n, d : m - r, e : 1$ puis l'alternance 4 et 5 transfère b dans d vers $c : n, d : m - 1, e : 1$ Dans ce cas, si on a $m - 1 > 0$, on va alors repasser dans la règle 10 pour tester la divisibilité de n par $m - 1$. Mais ce qui est subtil ici, c'est que le cas $m - 1 = 0$ a déjà été pris en compte car il correspond à $m = 1$, c'est-à-dire à une divisibilité qui aboutit forcément et qui est donc traité par le cas suivant.
- Soit m divise n, donc $n = km$ et on va aboutir forcément par soustraction après une règle 0 à une configuration de la forme $a : n, d : m - 1, g : 1$ Donc, contrairement au cas précédent, la règle 2 ne s'applique pas car $b = 0$. On passe donc à la règle 8 qui produit $a : n, d : m - 1$ C'est ici qu'on peut produire un nombre premier, car si $m - 1 = 0$, cela veut dire que la seule divisibilité s'est produite pour 1, donc que n est premier. Or, on a alors uniquement n dans le compteur `a`, et ainsi ça correspond au 2^n.

On poursuit alors par une étape de nettoyage pour passer à l'entier $n+1$ et continuer. Pour cela, on applique la règle 11 comme précédemment. Le seul changement c'est qu'avant de passer à la règle 13, on va appliquer la règle 12 pour vider le compteur `d`.

Ainsi, ce programme pourrait être traduit par le code `Python` suivant :

```python
a = 1
while True:
    a = a+1
    for d in reversed(range(1,a)):
        b = a
        reste = False
        while b != 0:
            for i in range(d):
                if b == 0:
                    reste = True
                    break
                b = b-1
        if not reste:
            break
    if d == 1:
        print(a,' est premier')
```

1.2.6 Modèles Turing-complets

Tous les exemples présentés ci-dessus sont équivalents : ils permettent de calculer les mêmes fonctions mathématiques pourvu qu'on précise la manière dont on donne l'entrée et comment on lit la sortie.

Par exemple, s'il n'est pas possible de réaliser l'incrémentation en FRACTRAN, il est possible d'écrire un programme permettant de passer de 2^a à 3^{a+1}.

On dit que ces modèles sont **Turing-complets** en référence au modèle de calcul de référence : les machines de Turing.

1.3 Langages, Compilateur, Interprète

1.3.1 Langages

Un langage de programmation est une manière de représenter textuellement un programme. Cela peut être un programme en lien avec une notion de machine mais aussi une notion plus abstraite comme on va le voir.

Comme on l'a vu, un ordinateur est une machine. Un programme pour cette machine est ce qu'on appelle un binaire et l'ensemble des instructions s'appelle le langage machine. C'est le langage naturel d'un ordinateur et il dépend de son processeur. Ainsi, un binaire pour un téléphone n'est pas directement utilisable sur un ordinateur, car les premiers ont des processeurs utilisant un jeu d'instructions arm et les seconds un jeu d'instructions x86_64.

1.3.2 Compilateur

Un **compilateur** est un traducteur d'un langage à un autre. On considère le plus souvent des compilateurs vers un langage machine, qu'il soit lié à des processeurs réels ou une machine spécifique appelée une machine virtuelle.

1.3.3 Interprète

Un interprète est un programme qui va lire un programme écrit dans un langage donné et l'évaluer en interprétant les instructions. Cela diffère d'une machine virtuelle dans le fait que le programme utilise toute la richesse du langage de programmation dans lequel il a été programmé pour son interprète. De plus, le fait de ne pas passer par une machine virtuelle permet de plus facilement relier l'environnement de l'interprète et celui du langage interprété. Un autre avantage discutable est la possibilité dans le langage cible de faire référence à l'interprète. Cela permet notamment de permettre l'évaluation dynamique de code (`eval` en Python).

Pour autant, les interprètes ont de nombreux défauts qui font qu'on a de plus en plus recours à des compilateurs vers des machines virtuelles. Parmi ceux-ci, citons le fait de devoir relire directement le programme à chaque fois qu'on va l'exécuter, ce qui est particulièrement critique quand on veut faire appel à des fonctions définies ailleurs, ou le fait que l'interprète lui-même est souvent très verbeux.

1.3.4 Unité de compilation, modules

Un programme est souvent structuré sous la forme de nombreux sous-programmes reliés entre eux. C'est une bonne pratique pour regrouper des fonctions selon un même thème et permettre la réutilisation du code sans duplication. On utilise pour cela la notion de modules ou de bibliothèques. Cela correspond au niveau du programme à rajouter du code existant au moment de la compilation ou de l'interprétation. Il est alors possible de réutiliser une partie de la compilation sur ce code, et ainsi, on utilise en général une version intermédiaire qui est du code précompilé utilisable par un autre programme, soit en copiant le résultat dans le programme ou en faisant référence à des versions définies dans le système (bibliothèque dynamique). Un programme est chargé de faire le lien entre les différentes bibliothèques et le programme, c'est l'éditeur de liens.

1.4 Paradigmes

Un paradigme de programmation est une manière de concevoir des programmes. On va citer ici trois paradigmes.

1.4.1 Impératif structuré

Un programme impératif est un programme construit plus ou moins directement en lien avec une notion de machines comme une suite d'instructions. Afin de permettre de programmer convenablement, un tel langage fournit des structures de contrôle (conditionnelles, boucles, fonctions) qui permettent de structurer le code. D'une certaine manière, la notion de classe et la programmation objet enrichit ce paradigme en permettant de structurer le code autour d'une notion d'objets.

1.4.2 Fonctionnel déclaratif

On a vu qu'en programmation fonctionnelle, tout n'est qu'une expression, un terme, qu'on évalue. Un programme fonctionnel déclaratif est ainsi constitué d'une succession de définitions de valeurs ou de fonctions. C'est ainsi qu'un programme `OCaml` n'est qu'une suite de `let` ou `let rec`.

Comment calcule un tel programme ? En évaluant chacune des déclarations, certaines vont effectivement produire des effets de bords et permettre la lecture ou l'écriture. L'usage en `OCaml` est ainsi de finir un programme par une déclaration `let () = ...` (ou `let _ = ...`) qui va se charger d'appeler les déclarations précédentes.

1.4.3 Programmation logique

Le paradigme de la programmation logique est sûrement le plus déroutant des trois. Un programme logique est en fait une formule logique dont une solution correspond à son résultat. Sans rentrer dans les détails, on remarquera que cela correspond à ce qu'on fait en SQL pour les bases de données où une requête de recherche est définie sous une forme de formule logique qui décrit les propriétés qui doivent être vérifiées par les éléments qu'on cherche.

CHAPITRE 2

Introduction à la programmation (impérative)

DANS CE CHAPITRE

2.1 Introduction
2.2 Représenter
2.3 Manipuler
2.4 Répéter
2.5 Abstraire

> **Note** **Roadmap** :
>
> - finir l'écriture avec `Python`.
> - rajouter `OCaml` et `C`.
> - en profiter pour faire une introduction aux références en `OCaml` quitte à rajouter une partie spécifique.
> - rajouter beaucoup d'exemples et d'exercices.

2.1 Introduction

Dans ce chapitre introductif sur la programmation, on va présenter celle-ci à travers quatre notions permettant de comprendre ce que signifie programmer. Un certain soin a été porté au fait de rendre cette présentation indépendante du langage tout en permettant qu'elle serve de support pour l'apprentissage d'un langage particulier. Ainsi, les éléments propres à un langage donné sont clairement séparés.

Comme nous l'avons vu dans le chapitre d'introduction, la programmation consiste à décrire des algorithmes et la manière dont ils sont mis en œuvre sur une machine. En cela, le modèle que l'on suit ici est celui de la programmation dite impérative structurée. Pour autant, de nombreuses notions présentées ici restent valides dans d'autres paradigmes.

La présentation faite ici est indépendante du langage sous réserve que celui-ci comporte des traits impératifs structurés. Pour autant, selon les langages certaines notions sont plus ou moins riches. Des paragraphes spécifiques complètent ainsi le texte et on encourage le lecteur à basculer d'un langage à un autre tout au long de la lecture.

C

`C` est un langage ayant peu d'expressivité directe pour les données. On est donc très vite réduit à devoir définir ses propres structures pour l'enrichir. Or, c'est plus complexe que la mise en œuvre des notions impératives développées ici. Le lecteur complétera alors sa lecture par le chapitre d'introduction aux structures de données en `C`.

Python

`Python` est le langage d'abord le plus aisé sur ces notions, mais aussi celui pour lequel il est vraiment nécessaire d'avoir une discipline d'utilisation afin de ne pas faire n'importe quoi. Ainsi, on a fait le choix de faire une

présentation sans artifice qui n'utilise pas de structures ou de mécanismes propres à `Python` et problématiques. Ceux-ci sont esquissés en remarque en précisant en quoi ils sont effectivement à manipuler avec parcimonie.

OCaml

`OCaml` est un langage qui est avant tout fonctionnel déclaratif. Pour autant, il possède des traits impératifs et permet d'écrire aussi naturellement des programmes impératifs que dans des langages comme `C` ou `Python`. Cependant, cette couche impérative se rajoute par-dessus la couche fonctionnelle et on supposera que le lecteur la maîtrise déjà. Ainsi, dans la présentation qui suit, on se concentrera sur l'exposition spécifique des notions impératives.

2.1.1 Les quatre étages de la programmation

Programmer c'est

- **représenter** des données, allant du booléen valant vrai ou faux, jusqu'aux bases de données permettant de modéliser des relations complexes
- **manipuler** ces données en lien avec un environnement, ce qui signifie autant de pouvoir les stocker que de pouvoir les transformer
- **répéter** ces manipulations élémentaires, car les données sont d'une taille finie, mais non connue à l'avance
- **abstraire** les notions précédentes pour ne pas avoir à se répéter, à travers la notion de fonctions

2.1.2 Exemple

Prenons un exemple simple pour comprendre les différents étages où le but final est de pouvoir calculer des sommes comme $1^2 + 2^2 + \ldots + n^2$.

Représenter Un entier comme 3 est une donnée qu'on pourra représenter telle quelle dans la plupart des langages avec cependant des limitations à garder en tête : la mémoire étant finie, on ne peut pas représenter des entiers quelconques, car il y en a une infinité. Déjà, à ce stade, on peut se poser la question de savoir si on ne veut représenter que des entiers dans un intervalle fixe, afin de borner leur occupation mémoire, ou si on veut les représenter en précision arbitraire. On peut opérer sur ces entiers avec un opérateur tel que `+` pour l'addition, mais le choix de la représentation aura son importance sur ce qui se cache derrière cette opération. L'addition de deux très grands entiers en précision arbitraire prendra plus de temps.

À l'aide de cela, on peut ainsi écrire `1*1 + 2*2 + 3*3 + 4*4 + 5*5`.

Manipuler On peut aussi les stocker directement en mémoire à l'aide de variables. Ainsi

C

```c
int x = 3;
```

Python

```python
x = 3
```

OCaml

```ocaml
let x = ref 3
```

permettra de définir une variable nommée x et contenant la valeur entière 3. On peut alors manipuler cette variable en changeant sa valeur à l'aide des opérations :

C

```c
x = x + 2;
```

Python

```python
x = x + 2
```

OCaml

```ocaml
x := !x + 2
```

Avec ces instructions de manipulation, on peut reprendre le calcul précédent de $1^2 + 2^2 + ... + 5^2$ avec une suite d'instructions :

C

```c
int s = 0;
s = s + 1*1;
s = s + 2*2;
s = s + 3*3;
s = s + 4*4;
s = s + 5*5;
```

Python

```python
s = 0
s = s + 1*1
s = s + 2*2
s = s + 3*3
s = s + 4*4
s = s + 5*5
```

OCaml

```ocaml
let s = ref 0 in
s := !s + 1*1;
s := !s + 2*2;
s := !s + 3*3;
s := !s + 4*4;
s := !s + 5*5
```

Répéter En fait, les 5 instructions d'ajout qu'on vient d'écrire ont la même structure : il s'agit d'ajouter une valeur i^2 à la variable `s` avec i prenant tour à tour les valeurs 1, 2, 3, 4 et 5. Une structure comme une boucle va permettre d'éviter cette répétition et d'écrire qu'il existe une variable `i` prenant ces valeurs et les instructions à réaliser pour chaque valeur de `i`. On pourra donc écrire :

C

```c
int s = 0;
for(int i = 1; i <= 5; i++)
{
    s = s + i*i;
}
```

Python

```python
s = 0
for i in range(1,5+1):
    s = s + i*i
```

OCaml

```ocaml
let s = ref 0 in
for i = 1 to 5 do
    s := !s + i*i
done
```

C'est en fait très proche de la notation $1^2 + 2^2 + ... + 5^2 = \sum_{i=1}^{5} i^2$. En fait, cette notation mathématique est très proche d'une boucle.

Abstraire Imaginons qu'on ait besoin d'effectuer le calcul précédent $\sum_{i=1}^{n} i^2$ à plusieurs reprises pour différentes valeurs de n, on pourrait à chaque fois recopier ces lignes en changeant la valeur maximale prise par i. Mais c'est inefficace pour plusieurs raisons :

- en recopiant du code, on risque de propager et répéter des erreurs. Si jamais on découvre une meilleure manière de faire le calcul, il faudra ainsi changer le code à chaque endroit où on l'a copié.

- là où on a besoin de ce calcul, il est possible que ce soit pour faire d'autres choses avec et donc qu'il s'inscrive dans une logique complexe. En copiant le code, on rend le programme plus compliqué à comprendre, car la boucle pour calculer cette somme est mise sur le même plan que le code qui nous intéresse.

Pour faire un parallèle avec les mathématiques, c'est comme si on recopiait une preuve dans un cas particulier chaque fois qu'on a besoin d'appliquer un théorème.

On introduit ainsi une notion d'abstraction *les fonctions* qui vont nous permettre de faire un code générique de calcul en prenant n en paramètre.

C

```c
int somme_carres(int n)
{
    int s = 0;
    for(int i = 1; i <= n; i++)
    {
        s = s + i*i;
    }
    return s;
}
```

Python

```python
def somme_carres(n):
    s = 0
    for i in range(1,n+1):
        s = s + i*i
    return s
```

OCaml

```ocaml
let somme_carres n =
    let s = ref 0 in
    for i = 1 to n do
        s := !s + i*i
    done;
    !s
```

Comme on le verra, une telle fonction est caractérisée par son nom qui nous permet d'y faire référence, comme lorsqu'on applique le théorème de Thalès, ainsi que des arguments qui seront, à l'exécution, remplacés par les valeurs qui nous intéressent, comme le fait que Thalès est démontré dans un triangle *générique* mais on l'applique sur un triangle particulier et une valeur de retour qui correspond à ce que la fonction calcule.

On peut alors, quand on a besoin de faire le calcul de $\sum_{i=1}^{5} i^2$ juste écrire :

C

```c
somme_carres(5)
```

Python

```python
somme_carres(5)
```

OCaml

```ocaml
somme_carres 5
```

2.1.3 Se remettre sans cesse à l'ouvrage

De notre présentation *étagée*, on pourrait retenir une hiérarchisation des différents étages. Il n'en est rien.

Ainsi, autant il peut être assez direct de représenter des données, comme dans le cas précédent, autant cela peut être une étape cruciale que de choisir la meilleure représentation. C'est le cas quand on s'intéresse au choix d'une structure de données la plus adaptée ou quand on conçoit un schéma de base de données.

La programmation, et plus largement l'informatique, sont ainsi à rapprocher d'un art martial comme l'Aïkido où on travaille tout au long de sa pratique les mêmes gestes simples en se perfectionnant sans cesse. Aller au bout de ce chapitre ne sera donc pas le signe qu'on maîtrise la programmation, mais juste qu'on a fait le premier pas sur la voie du perfectionnement.

Tout au long des autres chapitres, on verra ainsi des méthodes, des notions, qui permettent de mieux comprendre et de mieux pratiquer la programmation. Mais au moment où on se retrouvera à programmer, on ne sortira pas de ces quatre étages.

2.2 Représenter

Les données sont regroupées en informatique autour de la notion de types. Un type de données peut être vu en première approximation comme un ensemble de données de même nature. On considère, par exemple, usuellement le type des entiers naturels ou celui des nombres à virgule flottante.

2.2.1 Les données simples

Les données les plus simples sont celles qui correspondent à des valeurs numériques. Elles dépendent assez souvent des langages de programmation, mais on retrouve toujours un type pour les booléens, un type pour les entiers et un type pour les nombres à virgule flottante, c'est-à-dire pour des représentations de certains nombres réels.

C

On a ainsi :

- le type `int` des entiers **bornés**, le plus souvent codés sur 32 bits, c'est-à-dire compris entre -2^{31} et $2^{31}-1$. Les opérations sont `+ - * / %` où, entre deux entiers, `/` est la division entière et `%` le reste. Un dépassement de capacité sur un `int` est une erreur de programmation dont le comportement n'est pas défini par le langage. Il existe d'autres types entiers (`unsigned int`, `long`, `int64_t`, ...) qu'on verra selon les besoins.
- le type `double` des nombres à virgule flottante comme `1.4` ou `-3.14`, avec les opérations `+ - * /`. Ici `/` est une vraie division, mais attention : `1 / 2` vaut `0` car c'est une division entre deux `int`, alors que `1.0 / 2` vaut `0.5`.
- le type `bool` des booléens, qui nécessite d'inclure `<stdbool.h>`, avec les deux valeurs `true` et `false` et les opérations `!` (négation), `&&` (conjonction) et `||` (disjonction). Les opérateurs de comparaison sont `== != > >= < <=`. Historiquement, `C` n'avait pas de booléens et utilisait des entiers : `0` pour faux et toute autre valeur pour vrai. C'est encore ce qui se passe en coulisses.
- le type `char` des caractères comme `'a'`, qui sont en fait de petits entiers (le code du caractère).

Python

On a ainsi :

- le type `int` des entiers en précision arbitraire muni des opérations usuelles `+ - * // % **` en prenant garde au fait que `//` est la division entière (et `%` le reste). Ces opérations prennent d'autant plus de temps que les nombres ont beaucoup de chiffres.
- le type `float` des nombres à virgule flottante comme `1.4` ou `-3.14`. On verra plus loin les nombres de $\mathbb{R}$ qui sont ainsi représentables. Les opérations sont alors `+ - * / **`. On remarque qu'on a une **vraie division** avec `/` et ainsi `1 / 2` renvoie un `float` correspondant à $\frac{1}{2}$.
- le type `bool` des booléens qui ne comporte que deux valeurs `True` et `False` et des opérations suivantes :
  - la négation `not` qui permet de passer de `True` à `False` et réciproquement.
  - la conjonction `and` telle que `b1 and b2` soit `True` lorsque `b1` et `b2` sont égaux à `True` et `False` sinon
  - la disjonction `or` telle que `b1 or b2` soit `False` lorsque `b1` et `b2` sont égaux à `False` et `True` sinon

  Les booléens sont particulièrement importants, car ils correspondent au type des comparaisons. On dispose des opérateurs de comparaison `== != > >= < <=` qui correspondent aux relations mathématiques $=\neq>\geq<\leq$ (l'égalité `==` et `!=` existent pour tous les types, les comparaisons d'ordre pour les nombres, les chaînes, les tuples, ...).

- le type `NoneType` assez particulier qui ne contient qu'une seule valeur `None` et aucune opération. Ce type permet de donner un sens à l'absence de valeur, comme pour une fonction qui ne renvoie rien.

OCaml

On a ainsi :

- le type `int` des entiers bornés, codés sur 63 bits sur une machine 64 bits : ils sont compris entre `min_int` et `max_int` $= 2^{62} - 1$. Les opérations sont `+ - * / mod` où `/` est la division entière et `mod` le reste. En cas de dépassement, le calcul se fait modulo 2^{63} sans erreur.
- le type `float` des nombres à virgule flottante comme `1.4` ou `-3.14`. Les opérations ont des noms différents de celles des entiers : `+. -. *. /. **`. Il n'y a aucune conversion implicite, `1 + 2.0` est une erreur de typage et on écrit `float_of_int 1 +. 2.0`.
- le type `bool` des booléens avec les valeurs `true` et `false` et les opérations `not`, `&&` et `||`. Les opérateurs de comparaison sont `= <> > >= < <=`. Attention, `==` et `!=` existent aussi mais testent l'égalité **physique** (même emplacement mémoire) : on ne les utilise pas pour comparer des valeurs.
- le type `char` des caractères comme `'a'`.
- le type `unit` qui ne contient que la valeur `()`. C'est le type des expressions qui ne renvoient rien d'intéressant, c'est-à-dire des instructions.

Remarque 2.1 Les divisions entières ne se comportent pas de la même manière sur les nombres négatifs : en `C` et en `OCaml`, le quotient est arrondi vers zéro et `-7 / 2` vaut `-3` (avec un reste `-1`), alors qu'en `Python` il est arrondi vers $-\infty$ et `-7 // 2` vaut `-4` (avec un reste `1`). Dans tous les cas, on a bien $-7 = 2q + r$.

2.2.2 Les données composées ou structurées

Les données simples permettent de tout représenter. En effet, l'élément de donnée le plus primitif est le bit qui correspond à un booléen. Ainsi, la mémoire d'un ordinateur est entièrement constituée de booléens. Mais en disant cela, on ne dit pas grand-chose, car il ne s'agit pas d'une soupe informe de booléens, mais d'une organisation structurée. C'est ainsi qu'on considère des données composées comme les tableaux ou les couples.

On distingue deux types de données composées :

- les données *immuables*, c'est-à-dire celles qui ne permettent pas de changer les valeurs qu'elles regroupent ni leur structure après leur création
- les données *mutables* qui le permettent.

C

Les données composées sont rudimentaires en `C` :

- **(mutable)** les tableaux, de taille fixée à la déclaration, dont les éléments sont tous du même type : `int t[3] = {1, 3, 4};`. On accède à l'élément d'indice `i`, en numérotant à partir de 0, avec `t[i]`. **Il n'y a pas de moyen de retrouver la taille d'un tableau** à partir du tableau lui-même : on doit la conserver à part, par exemple dans une variable.
- les chaînes de caractères comme `"bonjour"` sont des tableaux de `char` dont la fin est signalée par un caractère spécial `'\0'`. Les chaînes écrites directement dans le programme ne doivent pas être modifiées.
- **(mutable)** les structures, qui regroupent plusieurs valeurs de types éventuellement différents dans des *champs* nommés, à la manière d'un n-uplet. On les verra dans le chapitre sur les structures de données en `C`.

Python

Les principaux types de données composées sont :

- **(immuable)** les `tuples` qui permettent de représenter des n-uplets de valeurs de types quelconques. On les écrit comme en mathématiques : `(1, 3.2, 4)`.

  Un tuple ne contenant qu'un seul élément doit tout de même faire apparaître la virgule pour le distinguer d'une valeur. Ainsi on écrit `(1,)` car `(1)` est juste la valeur `1`. Le tuple vide s'écrit `()`.

On dispose d'un opérateur de concaténation `+` qui permet de créer un tuple obtenu en mettant bout à bout deux autres tuples. Ainsi `(1,2) + (3,4,5)` est le tuple `(1,2,3,4,5)`. En mathématiques, si b est entier naturel on a $a \times b = a + ... + a$ où on ajoute b fois a. De même, on dispose d'une opération `*` telle que `(1,2) * 3` corresponde à `(1,2) + (1,2) + (1,2)`, c'est-à-dire au tuple `(1,2,1,2,1,2)`.

On obtient la taille d'un tuple à l'aide de `len`. En effet, `len( (1,2,3) )` renvoie 3. Attention aux doubles parenthèses ici, on le verra, `len` est une fonction et les parenthèses sont nécessaires pour l'appliquer, mais le tuple lui-même s'écrit avec des parenthèses.

On accède aux différentes composantes d'un tuple avec la construction `t[i]` où `t` est un tuple et `i` est l'indice de la composante en commençant par numéroter à `0`. Ainsi `(1, 3.2, 4)[1]` est la valeur `3.2`. On peut aussi utiliser des indices négatifs : pour $1 \leq k \leq$ `len(t)`, `t[-k]` désigne `t[len(t)-k]`, ainsi on peut écrire `t[-1]` plutôt que `t[len(t)-1]` pour accéder à la dernière composante. Un indice en dehors de ces valeurs provoque une erreur.

- **(immuable)** les `str` ou chaînes de caractères qui permettent de représenter des textes comme `'bonjour'` ou `"au revoir"`. On utilise indifféremment `'` ou `"` mais il faut que ce soit le même symbole qui ouvre et ferme une chaîne. Il n'y a pas de type spécifique pour les caractères en `Python` : ce sont des chaînes de longueur 1 comme `'a'`.

  Les opérations sont les mêmes que pour les tuples avec `+` pour la concaténation, `len` pour la longueur ou `[i]` pour accéder au ième caractère. C'est quelque chose qu'on retrouvera pour la plupart des types composés qui sont des suites d'éléments, on parle de **séquences**.

- **(mutable)** les `list` ou tableaux qui correspondent à des tuples dont on peut modifier les valeurs ou même modifier la structure, comme en ajoutant ou en supprimant des éléments. Cela correspond alors à des manipulations et non des opérations, et on les verra au paragraphe suivant.

  Pour définir un tableau, on écrit comme un tuple mais avec des crochets. Ainsi on peut écrire `[1,3.2,3]`. Le tableau vide est `[]` et comme il n'y a pas d'ambiguïté, un tableau contenant un unique élément se note juste `[1]`.

  Là encore, on dispose des opérations communes de concaténation, longueur et accès.

- **(mutable)** les `dict` ou dictionnaires, qui permettent de réaliser des associations entre des valeurs **immuables** et d'autres valeurs. On peut ainsi considérer une valeur comme `{ 'nom' : 'Michel', 'prenom' : 'Louise', 42 : True }` les valeurs à gauche du `:` sont les **clés** qui permettent d'accéder aux valeurs, comme lorsqu'on cherche un mot dans un dictionnaire pour accéder à sa définition.

  Ici, on ne dispose plus de concaténation, car ça ne voudrait pas dire grand-chose d'ajouter deux dictionnaires (que fait-on si une même clé est présente dans les deux ?). On dispose toutefois de `len` pour le nombre d'associations et il est possible d'accéder à la valeur associée à une clé en écrivant `d[cle]` où `d` est une valeur de type `dict` et `cle` une clé présente dans `d`.

  Le dictionnaire vide est `{}` et on verra dans le paragraphe suivant comment manipuler un dictionnaire pour ajouter ou supprimer des associations.

OCaml

Les principaux types de données composées sont :

- **(immuable)** les n-uplets comme `(1, 3.2, 4)` de type `int * float * int`. On accède aux composantes par filtrage : `let (a, b, c) = t in ...` (ou avec `fst` et `snd` pour un couple).
- **(immuable)** les chaînes de caractères comme `"bonjour"`, de type `string`, avec `^` pour la concaténation, `String.length s` pour la longueur et `s.[i]` pour le caractère d'indice `i`.
- **(immuable)** les listes comme `[1; 3; 4]`, de type `int list`, dont tous les éléments ont le même type. Ce sont des données inductives qu'on manipule principalement par filtrage et récursivité.

- **(mutable)** les tableaux comme `[|1; 3; 4|]`, de type `int array`, de taille fixée à la création, dont tous les éléments ont le même type. On obtient la taille avec `Array.length t`, on accède à l'élément d'indice `i` avec `t.(i)` et on crée un tableau de taille `n` rempli de `x` avec `Array.make n x`.
- **(mutable)** les enregistrements dont certains champs ont été déclarés `mutable`, comme dans `type point = { mutable x : int; mutable y : int }`.
- **(mutable)** les tables de hachage du module `Hashtbl`, qui jouent le rôle des dictionnaires.

2.3 Manipuler

2.3.1 Variables

L'élément clé permettant de manipuler des valeurs est de pouvoir les placer à un endroit et de changer la valeur qui s'y trouve. C'est ce qu'on appelle une **variable**. Même si cela ne correspond pas à une réalité selon les langages, il est d'usage de considérer une variable comme une case mémoire disposant d'un nom et dans laquelle on place des valeurs.

On a alors trois éléments :

- définir une variable, souvent en la limitant à un certain type de donnée, on parle de **déclaration**
- accéder à sa valeur
- changer sa valeur, on parle d'**assignation**

Les noms de variables sont le plus souvent composés de lettres, de chiffres et du symbole _, ils doivent commencer par une lettre (ou _).

C

Une variable se déclare en précisant son type, éventuellement avec une valeur initiale :

```c
int x;      // déclaration seule
int y = 3;  // déclaration avec initialisation
x = y + 2;  // assignation
```

Le type d'une variable est fixé une fois pour toutes : `x` ne pourra contenir que des `int`. Pour accéder à la valeur d'une variable, il suffit d'écrire son nom dans une expression, comme `y + 2`.

Attention, une variable déclarée sans initialisation contient une valeur indéterminée (ce qui se trouvait à cet endroit de la mémoire) : la lire avant de lui avoir assigné une valeur est une erreur classique.

Python

Il n'existe pas de mécanisme spécifique pour déclarer une variable distinct de l'assignation. Ainsi, on écrira `x = 3` à la fois pour déclarer une variable nommée `x` contenant la valeur 3 et pour changer la valeur de `x`.

Pour accéder à la valeur d'une variable, il suffit d'écrire son nom directement dans une expression. Ainsi, on pourra écrire `x+2` pour la valeur obtenue en ajoutant 2 à la valeur de `x`.

Les valeurs en `Python` sont typées mais pas les variables. Ainsi on pourra tout à fait écrire

```python
x = 3 # déclare une variable x contenant un entier
### puis plus loin
x = 'Bonjour' # maintenant x contient une chaîne
```

OCaml

Un nom défini par `let x = 3` n'est **pas** une variable au sens de ce chapitre : il désigne la valeur `3` et on ne pourra jamais changer cette association. D'ailleurs, `x = 4` n'est pas une assignation mais une comparaison qui renvoie `false`.

Pour disposer d'une variable, on utilise une **référence**, créée avec `ref`, qui est une case mémoire contenant une valeur qu'on peut modifier. On lit son contenu avec `!` et on le change avec `:=` :

```
## let x = ref 3;;
val x : int ref = {contents = 3}
## !x + 2;;
- : int = 5
## x := 4;;
- : unit = ()
## x;;
- : int ref = {contents = 4}
```

Une référence a un type fixé : `x : int ref` ne pourra contenir que des entiers, et `x := "bonjour"` provoque une erreur de typage. Il faut bien distinguer `x`, qui est la case mémoire, de `!x`, qui est son contenu : écrire `x + 2` est une erreur de typage.

2.3.2 État d'exécution

L'ensemble des variables et des valeurs auxquelles elles sont associées à un moment de l'exécution d'un programme est un état d'exécution. Les instructions de déclaration et d'assignation modifient ainsi l'état.

Suite aux instructions

C

```c
int a = 3;
int b = 2;
```

Python

```python
a = 3
b = 2
```

OCaml

```
let a = ref 3
let b = ref 2
```

l'état est alors

variable	valeur
a	3
b	2

Si on exécute ensuite l'instruction

C

```c
b = a;
```

Python

```python
b = a
```

OCaml

```
b := !a
```

il devient

variable	valeur
a	3
b	3

Dans la suite, on pourra noter `a=3, b=3` un tel état. Ici, le `=` permet une notation intuitive.

2.3.3 Instructions et blocs

Les programmes qu'on va écrire vont maintenant être des suites d'instructions, comme

C

```c
int x = 3;
int y = x + 2;
x = y;
```

Python

```python
x = 3
y = x + 2
x = y
```

OCaml

```
let x = ref 3 in
let y = ref (!x + 2) in
x := !y
```

Ces instructions sont regroupées ensemble sous une notion de blocs. Suivant les langages, ces blocs sont plus ou moins explicites.

C

Un bloc est délimité par des accolades. Chaque bloc définit sa propre notion de portée.

```c
{
    int x = 3;
    int y = 2;

    x = y + 2;
    { // bloc imbriqué
        int y = 4;
    } // la portée de ce y s'arrête ici

    x = y + 2; // ici y n'est pas celui du bloc imbriqué
}
```

Python

Un bloc est un groupement d'instructions à la suite qui sont toutes alignées dans le texte du programme. On dispose ainsi d'une indentation signifiante, c'est-à-dire que cet alignement a du sens et il est directement compris par `Python` pour savoir si une instruction appartient à un bloc ou non.

Ainsi, on ne pourra pas écrire

```python
x = 3
   y = x + 2 # non aligné avec le reste du bloc
x = y
```

OCaml

En `OCaml`, il n'y a pas à proprement parler de notion de blocs car tout est une expression. Cependant, l'opérateur `;` permet d'évaluer successivement deux expressions. On pourra donc écrire

```
x := 3;
y := !x + 2;
x := !y
```

pour exécuter successivement ces instructions. On remarque que, contrairement à certains langages, `;` n'est pas un terminateur mais un séparateur : il est inutile de terminer par `;`.

Comme la valeur des expressions à gauche de `;` est perdue, on suppose qu'on ne le fait que pour des expressions dont la valeur est `() : unit`, c'est-à-dire pour les instructions. Si jamais on utilise `;` avec une expression ayant une autre valeur à gauche, un avertissement nous indique que l'usage est problématique :

```
### 3 ; 2;;
Warning 10 [non-unit-statement]: this expression should have type unit.
- : int = 2
```

Les blocs ont souvent besoin d'être séparés du reste du programme comme on le verra dans la suite. Pour cela, comme il s'agit de simples expressions, on peut utiliser les parenthèses comme ici :

```
(x := 3;
y := !x + 2;
x := !y)
```

mais il est souvent plus clair d'utiliser la paire `begin/end` qui, bien qu'elle soit strictement équivalente à une paire de parenthèses, permet de faire apparaître le fait qu'on a un bloc :

```
begin
    x := 3;
    y := !x + 2;
    x := !y
end
```

Derrière cette notion de bloc on trouve une notion implicite qui est celle de flot d'exécution et qui régit la manière dont les instructions sont exécutées. En accord avec la manière dont on écrit les programmes, les instructions sont exécutées de haut en bas. L'état d'exécution est alors modifié le long du flot.

Une analogie qui sera utile dans la suite est de voir l'état d'un programme comme un train qui circule à travers les rails du flot de contrôle, en rajoutant ou modifiant des wagons à chaque instruction présentée comme un arrêt. Cette image assez naïve permet de faire passer une idée très importante qui est celle de l'ordre d'évaluation et du contexte associé.

Python

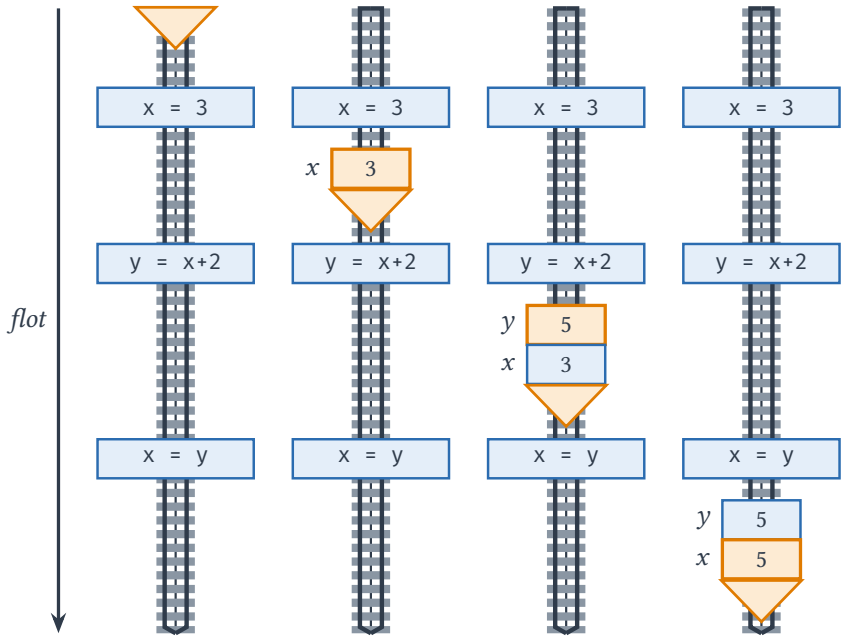


Figure 2.1 — L'état du programme, comme un train, est modifié à chaque instruction du flot

Un exemple important est celui de l'échange du contenu de deux variables `a` et `b`. On ne peut pas écrire

C

```
b = a;
a = b;
```

Python

```
b = a
a = b
```

OCaml

```
b := !a;
a := !b
```

car comme on vient de le voir, la valeur de `b` est déjà perdue après la première assignation. Une solution consiste alors à introduire une variable temporaire qui ne va servir que de support pour pouvoir garder la valeur de `b`. On écrira ainsi :

C

```
// ici, on ne peut pas le faire de manière
// générique, cela dépend du type de a et b.
int temp = b;
b = a;
a = temp;
```

Python

```
temp = b
b = a
a = temp
```

OCaml

```
let temp = !b in
b := !a;
a := temp (* pas besoin que temp soit une ref *)
```

2.3.4 Portée

Une variable a une durée de vie, souvent, elle n'existe que localement dans un programme. Par exemple, uniquement au sein de la fonction ou du bloc dans laquelle on l'a définie. On parle de portée d'une variable pour désigner l'ensemble des instructions dans lesquelles on peut y accéder.

C

La portée d'une variable s'étend de sa déclaration jusqu'à la fin du bloc qui la contient, comme on l'a vu avec le `y` du bloc imbriqué. Une variable déclarée dans l'en-tête d'une boucle `for`, comme `i` dans `for(int i = 1; i <= 5; i++)`, n'existe que dans cette boucle. Une variable déclarée en dehors de toute fonction est dite **globale** et est accessible partout dans la suite du fichier.

Python

La portée des variables est assez dynamique. On peut ainsi définir une variable uniquement dans certains cas comme on le verra avec une instruction conditionnelle. Ce qui fait que selon l'état d'un programme, on définira différemment la portée d'une même variable.

OCaml

La portée est explicite : dans `let x = e1 in e2`, le nom `x` n'est accessible que dans l'expression `e2`. Un `let x = e` écrit au niveau global, sans `in`, est accessible dans toute la suite du programme. Comme une référence est une valeur comme une autre, ces règles s'appliquent aussi aux variables créées avec `ref`.

2.3.5 Manipuler des données composées mutables

Les types de données mutables permettent de modifier leur valeur. Ce sont le plus souvent des **collections**, c'est-à-dire des arrangements structurés de valeurs, comme un tableau qui les arrange en ligne de la première à la dernière. Lorsqu'on peut faire un accès direct à une valeur, on peut alors la modifier comme si c'était une variable. Il peut également être possible de faire des modifications sur la structure de la collection elle-même, comme enlever ou rajouter des éléments.

Modification des valeurs des éléments

C

Pour un tableau `t` de n éléments, `t[i]` se comporte comme une variable pour $i \in [\![0, n-1]\!]$: on peut écrire `t[i] = 3`. De même, pour une structure `p` ayant un champ `x`, on écrit `p.x = 3`.

```c
int t[3] = {1, 0, 2};
t[0] = 3;    // t contient 3, 0, 2
t[2] = t[1]; // t contient 3, 0, 0
```

Attention, `C` ne vérifie pas les indices : accéder à `t[3]` ou `t[-1]` est un comportement indéfini. Le programme peut s'arrêter, mais il peut aussi lire ou écraser silencieusement une autre donnée de la mémoire. C'est une source majeure d'erreurs, et de failles de sécurité.

Python

On a vu deux types mutables : les tableaux (`list`) et les dictionnaires (`dict`).

- Pour une variable `t` contenant un tableau de n éléments, on peut accéder à `t[i]` pour $i \in [\![0, n-1]\!]$ comme si c'était une variable. Ainsi, on pourra écrire `t[i] = 3` pour changer la valeur à la ième case par `3`. Un tableau peut ainsi être vu comme une suite finie de variables. Si jamais on accède en lecture comme en écriture à `t[j]` quand $j \notin [\![-n, n-1]\!]$, cela va déclencher une erreur et non ajouter une association entre j et cette valeur.

  Ainsi, il n'est pas possible de changer la structure d'un tableau par des assignations.

  Si on exécute la suite d'instructions

  ```
  t = [1, 0, 2]
  t[0] = 3
  t[2] = t[1]
  ```

  L'état va alors passer de `t=[1, 0, 2]` à `t=[3, 0, 2]` puis `t=[3, 0, 0]`.

  Tout se passe comme si on avait trois variables `t0, t1, t2` avec les instructions :

  ```
  t0 = 1
  t1 = 0
  t2 = 2

  t0 = 3
  t2 = t1
  ```

- Pour une variable `d` contenant un dictionnaire, on peut accéder à `d[c]`, où `c` est une clé présente dans `d`, en lecture comme en écriture. Si on assigne une valeur à `d[c]` alors que `c` n'est pas une clé présente dans le dictionnaire, elle sera alors ajoutée. Ce qui fait que, contrairement au cas des tableaux, il est possible de modifier la structure du dictionnaire à l'aide d'assignation.

  Si on considère la suite d'instructions

  ```
  d = { 'a' : 2, 'test' : 3 }
  d['a'] = 4
  d['b'] = d['test']
  ```

  On passe de l'état `d={ 'a' : 2, 'test' : 3 }` à `d={ 'a' : 4, 'test' : 3 }` puis `d={ 'a' : 4, 'test' : 3, 'b' : 3 }` (l'ordre d'affichage des associations est l'ordre d'insertion).

OCaml

- Pour un tableau `t` de n éléments et $i \in [\![0, n-1]\!]$, on modifie l'élément d'indice `i` avec `t.(i) <- v`. Un indice hors de ces bornes déclenche l'exception `Invalid_argument "index out of bounds"`.

```
let t = [|1; 0; 2|] in
t.(0) <- 3;    (* t = [|3; 0; 2|] *)
t.(2) <- t.(1) (* t = [|3; 0; 0|] *)
```

- Pour un enregistrement `r` ayant un champ mutable `x`, on écrit `r.x <- v`. Une référence n'est d'ailleurs rien d'autre qu'un enregistrement avec un unique champ mutable `contents`, et `x := v` est une abréviation de `x.contents <- v`.
- Pour une table de hachage `d`, `Hashtbl.replace d c v` associe `v` à la clé `c`, en ajoutant l'association si `c` n'était pas présente, et `Hashtbl.find d c` renvoie la valeur associée à `c` (ou déclenche l'exception `Not_found`).

Partages de valeurs composées entre plusieurs variables

Il y a une différence fondamentale entre écrire

```
a = 3
b = a
a = 2
```

et écrire

```
a = [ 3 ]
b = a
a[0] = 2
```

Dans le premier cas, lors de l'assignation `b = a`, il n'y a pas de lien entre la valeur `3` dans `a` et celle dans `b`. Dans le second cas, après `b = a`, on a `b` et `a` qui pointent vers le même tableau. On notera `b, a = [ 3 ]` cet état. Ainsi, quand on exécute l'instruction `a[0] = 2` on modifie le tableau qui est également associé à `b`. Donc `a` **et** `b` sont associées au tableau `[ 2 ]`.

En fait, quand on écrit `a = [ 3 ]`, on ne **stocke** pas la valeur du tableau dans `a`. On crée le tableau `[ 3 ]` en mémoire et on place dans `a` une référence vers ce tableau. Quand on exécute alors `b = a`, on place dans `b` une autre référence mais vers le même tableau. Une manière simple de voir cela est de représenter les références par des flèches, ainsi l'état pourrait en fait se représenter ainsi :

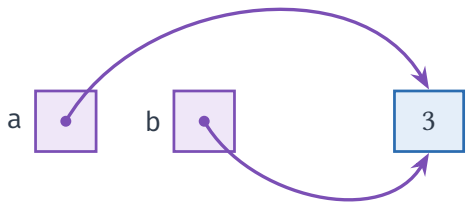


Figure 2.2 — a et b sont deux références vers le même tableau

Si on considère un tableau de tableau comme `[ [ 0 ], [ 1 ] ]` on aura alors dans les cases du tableau principal des références vers les deux tableaux `[ 0 ]` et `[ 1 ]` :

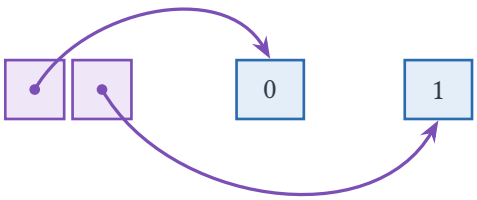


Figure 2.3 — Les cases du tableau principal référencent les tableaux [0] et [1]

Ainsi, si on écrit

```
t = [ [ 0 ], [ 1 ] ]
t[1] = t[0]
t[1][0] = 2
```

on aura alors les deux références qui pointent vers le même tableau, et quand on modifie `t[1][0]`, on modifie donc forcément `t[0][0]` également pour aboutir à l'état :

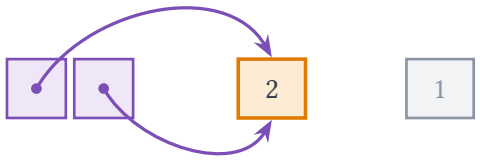


Figure 2.4 — Les deux références désignent le même tableau ; plus rien ne désigne [1]

On peut remarquer que plus rien ne fait référence ici au tableau `[ 1 ]`. Il y a un mécanisme dans `Python` qui détecte cela et qui supprime de la mémoire le tableau.

Ce qui vient d'être dit pour les tableaux en Python est encore vrai avec les dictionnaires, et plus généralement avec la plupart des données.

Modification de la structure des données composées

Pour parler de modification de tableaux ou de dictionnaires, il est nécessaire de comprendre que `Python` est un langage objet. On se contentera ici de dire qu'un objet est une donnée munie d'opérations de manipulation sur cette donnée.

Ainsi, si `t` est un tableau, on pourra écrire `t.append(x)` pour ajouter la valeur `x` comme une nouvelle case à la fin du tableau. `t.append(x)` est une instruction, elle ne renvoie pas un nouveau tableau avec cette modification mais elle modifie directement `t`. On dit que `.append` est une **méthode** de la **classe** `list` de `t`. Il est tout à fait possible d'utiliser `Python` sans écrire de classes, mais dans la mesure où c'est un langage objet dans son cœur, on manipulera forcément des objets.

Par exemple, la suite d'instructions :

```
t = []
t.append(2)
t.append(3)
```

va faire passer l'état de `t = []` à `t = [ 2 ]` puis `t = [ 2, 3 ]`.

Il existe beaucoup de méthodes pour les types `list` et `dict`. On les verra selon les besoins dans la suite. Il est toutefois possible, et souhaitable, de se référer à la documentation pour avoir une description détaillée de celles-ci.

2.3.6 Instruction conditionnelle

L'instruction conditionnelle va permettre d'enrichir le flot de contrôle avec des branchements. Elle permet d'orienter le flot d'exécution dans un bloc ou dans un autre selon qu'une condition soit vérifiée ou non.

On écrit

C

```
if (condition)
{
    // bloc si la
    // condition est vérifiée
}
else
```

```
{
    // bloc si la
    // condition n'est pas
    // vérifiée
}
```

Les parenthèses autour de la condition sont obligatoires.

Python

```
if condition:
    # bloc si la
    # condition est vérifiée
else:
    # bloc si la
    # condition n'est pas
    # vérifiée
```

OCaml

En `OCaml` il y a des expressions conditionnelles qui ont déjà été vues mais qui maintenant prennent une autre signification avec des expressions contenant des instructions. Dans le contexte de la programmation impérative, on pourra utiliser `begin/end` pour accentuer les différences :

```
if condition
then
  begin
    (* bloc si la condition
    est vérifiée *)
  end
else
  begin
    (* bloc si la condition
    n'est pas vérifiée *)
  end
```

Si on reprend l'analogie des trains et des rails, une instruction conditionnelle est un échangeur qui, selon que le train vérifie ou pas la condition va le faire circuler dans une branche ou une autre. Il est important de comprendre qu'une fois une branche exécutée, les flots se rejoignent sur l'instruction qui suit l'instruction conditionnelle.

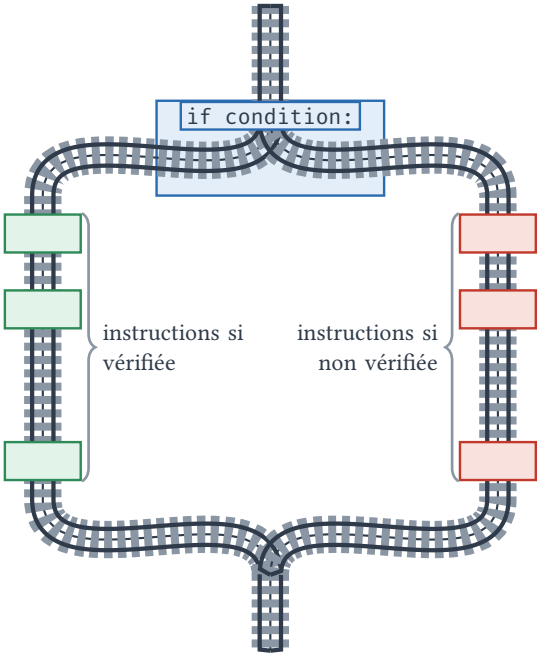


Figure 2.5 — Une instruction conditionnelle aiguille le flot, qui se rejoint ensuite

Il arrive qu'on n'ait rien à faire dans une branche, on peut alors placer un bloc vide, mais, en général, on préfère, quitte à nier la condition, ne pas écrire de `else` comme dans :

```
if condition:
    # bloc si la
    # condition est vérifiée
### le flot saute directement ici si
### la condition n'est pas vérifiée
```

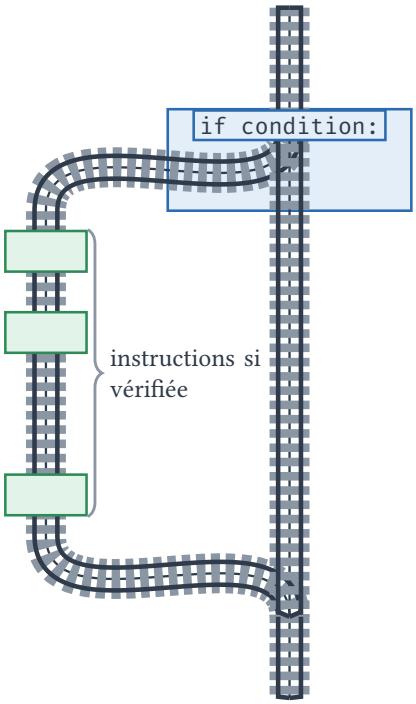


Figure 2.6 — Sans else, le flot saute directement après le bloc si la condition est fausse

Remarque 2.2 En Python, un bloc vide se note `pass` comme dans le programme suivant :

```
if x == 3:
    y = y + 1 # incrémente y si x vaut 3
else:
    pass # ne fait rien sinon
```

qu'on aurait pu écrire :

```
if x == 3:
    y = y + 1 # incrémente y si x vaut 3
```

On pourrait être tenté d'écrire l'instruction a priori inoffensive `y = y` dans le bloc du `else`, mais une telle instruction n'est pas forcément gratuite. En Python, `y = y` ne fait que recopier une référence (voir plus loin), mais dans d'autres langages, comme C avec des structures, une assignation recopie la donnée, ce qui peut être coûteux, et une opération comme `y = y + 0` peut construire une nouvelle valeur. C'est donc une bonne pratique de ne pas écrire des opérations qui, si pour nous elles ne font rien, peuvent en fait avoir un coût caché.

Un principe clé de la programmation est la compositionnalité des instructions de gestion de flot : on peut placer des instructions conditionnelles dans le corps d'une des branches. On pourra donc écrire

```
if condition1:
    if condition2:
        # bloc si condition 1 et condition 2 sont vérifiées
    else:
        # bloc si condition 1 est vérifiée mais condition 2 ne l'est pas
else:
    if condition2:
        # bloc si condition 1 n'est pas vérifiée et condition 2 l'est
    else:
        # bloc si condition 1 et condition 2 ne le sont pas
```

Conditions et opérations sur les booléens

Le rôle des conditions est central dans l'usage des instructions conditionnelles et ainsi c'est très important de bien comprendre le fonctionnement des booléens et de leurs opérations.

Une condition est une formule logique constituée

- de formules atomiques portant sur l'état comme `x == 3`, `x < 2`, ...
- d'opérateurs booléens pour relier ces formules : `not`, `and` ou `or`

Remarque 2.3 Les opérateurs `and` et `or` (`&&` et `||` en C et en OCaml) sont évalués **paresseusement** de la gauche vers la droite : dans `a and b`, si `a` est faux, `b` n'est même pas évalué puisque le résultat est forcément faux. De même, dans `a or b`, `b` n'est pas évalué si `a` est vrai. C'est très utile pour protéger un accès :

```
if i < len(t) and t[i] == x:
    # ici, on est sûr que t[i] existe
```

Écrire la condition dans l'autre ordre, `t[i] == x and i < len(t)`, provoquerait une erreur lorsque `i` est hors du tableau.

Inversion de point de vue par rapport aux mathématiques

En mathématiques, il est courant d'avoir des définitions de valeurs par cas. Par exemple, on pourrait écrire

$$x = \begin{cases} \frac{y}{2} & \text{si } y \text{ pair} \\ \frac{y-1}{2} & \text{sinon.} \end{cases}$$

On pourrait le traduire alors aisément ainsi :

```
if y % 2 == 0:
    x = y // 2
else:
    x = (y-1) // 2
```

Mais en faisant cela, on a inversé le point de vue en plaçant la condition avant l'affectation de `x`.

✍ Les trois langages disposent d'une **expression** conditionnelle qui permet de conserver le point de vue mathématique, où l'on définit `x` par cas :

C

```
x = (y % 2 == 0) ? y / 2 : (y - 1) / 2;
```

Python

```
x = y // 2 if y % 2 == 0 else (y - 1) // 2
```

✍ **OCaml**

En `OCaml`, `if ... then ... else ...` est toujours une expression, c'est même son usage principal :

```
let x = if y mod 2 = 0 then y / 2 else (y - 1) / 2
```

On réserve cette écriture aux cas simples : dès que les branches contiennent plusieurs instructions, l'instruction conditionnelle est plus lisible.

Instructions conditionnelles en cascade

✍ Il est fréquent d'avoir plus de deux cas. On pourrait imbriquer des instructions conditionnelles dans les branches `else`, mais l'indentation deviendrait vite illisible. On dispose donc d'une écriture en cascade, où les conditions sont testées dans l'ordre et **seul le bloc de la première condition vérifiée** est exécuté. Par exemple, pour calculer le signe d'un entier :

C

```
int s;
if (x > 0)
{
    s = 1;
}
else if (x < 0)
{
    s = -1;
}
else
{
    s = 0;
}
```

Python

```
if x > 0:
    s = 1
elif x < 0:
    s = -1
else:
    s = 0
```

OCaml

```
let s =
    if x > 0 then 1
    else if x < 0 then -1
    else 0
```

> **Remarque 2.4** L'ordre des conditions compte. Pour attribuer une mention à une note sur 20, si on teste `note >= 10` avant `note >= 16`, la branche « très bien » ne sera jamais atteinte. Quand les cas se recouvrent, il faut tester du plus restrictif au moins restrictif.

2.3.7 Entrées et sorties

Un programme communique avec son environnement : il lit des **entrées** (clavier, fichier, ...) et produit des **sorties** (affichage à l'écran, écriture dans un fichier, ...). Ces opérations modifient l'environnement et sont donc, elles aussi, des instructions.

C

On affiche avec `printf` (après `#include <stdio.h>`), qui prend une chaîne de format où `%d` (entier), `%f` (flottant), `%c` (caractère) ou `%s` (chaîne) sont remplacés par les valeurs suivantes :

```c
printf("x vaut %d et y vaut %f\n", x, y);
```

On lit symétriquement avec `scanf("%d", &x);`. Le `&` sera expliqué avec la notion de pointeur.

Python

On affiche avec `print(x, y)`, qui sépare les valeurs par des espaces et termine par un retour à la ligne. On lit une ligne au clavier avec `input()`, qui renvoie toujours une chaîne qu'on convertit si besoin : `n = int(input())`. Les fichiers sont traités dans le chapitre sur les entrées-sorties en Python.

OCaml

On affiche avec `print_int`, `print_string`, `print_newline ()`, ... ou avec `Printf.printf` qui fonctionne comme en `C` :

```
Printf.printf "x vaut %d et y vaut %f\n" !x !y
```

On lit une ligne au clavier avec `read_line ()`, qui renvoie une chaîne, ou un entier avec `read_int ()`.

> **Remarque 2.5** Afficher une valeur et la **renvoyer** sont deux choses très différentes. Une fonction qui affiche son résultat au lieu de le renvoyer ne peut pas être réutilisée dans un calcul. Dans ce cours, les fonctions renverront leur résultat et l'affichage sera réservé à la communication avec l'utilisateur.

2.4 Répéter

Jusqu'ici, on a vu des instructions permettant de manipuler l'état d'exécution, éventuellement de manière différente selon sa valeur à l'aide d'instructions conditionnelles. Mais étant donné un état de départ, on connait déjà les instructions qui seront exécutées et surtout **on connait leur nombre** qui est majoré par le nombre total d'instructions. En effet, pour reprendre l'analogie des trains, ceux-ci ne font que descendre le long du flot et le nombre d'arrêts rencontrés est majoré par le nombre d'arrêts total.

Cependant, la force principale de l'informatique vient du fait qu'on peut, à l'aide d'un nombre d'instructions fini, avoir la potentialité d'exécuter un nombre arbitrairement grand d'instructions. À cette fin, nous allons introduire un élément fondamental : la notion de répétition ou de boucles.

On pourrait être étonné de la formulation précédente sur ce nombre arbitrairement grand car la mémoire, et même le temps d'exécution en pratique, étant fini, tout est borné. Si on considère un programme qui va calculer la somme des éléments d'un tableau et qu'on sait que la longueur d'un tel tableau est majorée, on pourrait imaginer une énorme instruction conditionnelle en cascade traitant séparément chaque longueur possible : si le tableau est de longueur 0, la somme vaut 0, s'il est de longueur 1, c'est `t[0]`, s'il est de longueur 2, c'est `t[0] + t[1]`, et ainsi de suite. Un tel programme aurait une taille proportionnelle à la borne, il serait illisible, et il faudrait le réécrire dès que la borne change. Une boucle permet au contraire de décrire une seule fois ce qu'on fait pour un élément, indépendamment du nombre d'éléments.

2.4.1 Répéter n fois

La boucle la plus simple consiste à répéter un bloc d'instructions un nombre de fois connu à l'avance, en disposant d'une variable, appelée **indice de boucle**, qui prend successivement des valeurs entières consécutives. C'est la boucle de `somme_carres` vue en introduction. Par exemple, pour calculer x^n :

C

```c
int p = 1;
for (int i = 0; i < n; i++)
{
    p = p * x;
}
```

L'en-tête `for (initialisation; condition; mise à jour)` est très général : on exécute l'initialisation, puis, tant que la condition est vraie, on exécute le bloc puis la mise à jour. La forme ci-dessus, où `i` prend les valeurs $0, 1, \ldots, n-1$, est de loin la plus courante.

Python

```python
p = 1
for i in range(n):
    p = p * x
```

`range(n)` désigne les entiers $0, 1, \ldots, n-1$ et, plus généralement, `range(a, b)` désigne les entiers de a inclus à b **exclu**. C'est pourquoi on a écrit `range(1, n+1)` dans `somme_carres`. On peut préciser un pas : `range(n-1, -1, -1)` parcourt $n-1, n-2, \ldots, 0$.

OCaml

```
let p = ref 1 in
for _ = 1 to n do
    p := !p * x
done
```

`for i = a to b do ... done` fait prendre à `i` les valeurs de a à b **inclus**, et `for i = b downto a do ... done` les parcourt dans l'ordre décroissant. Quand l'indice ne sert pas dans le corps, on peut le nommer `_`.

> **Remarque 2.6** On ne modifie jamais l'indice de boucle dans le corps de la boucle. En `OCaml`, c'est impossible car `i` n'est pas une référence. En `Python`, une assignation `i = 10` dans le corps n'a d'effet que jusqu'à la fin de l'itération courante, `i` reprenant ensuite la valeur suivante de `range`. En `C`, c'est possible mais rend le programme très difficile à comprendre.

2.4.2 Répéter pour chaque élément

Un cas très fréquent consiste à effectuer un traitement pour chaque élément d'une collection, par exemple d'un tableau.

C

Comme on ne peut pas retrouver la taille d'un tableau, on la transmet avec lui et on parcourt les indices :

```c
for (int i = 0; i < n; i++)
{
    // traitement de t[i]
}
```

Python

On peut parcourir directement les éléments d'une séquence (tableau, tuple, chaîne, ...) sans passer par les indices :

```python
for x in t:
    # traitement de x
```

Si on a besoin des indices, on écrit `for i in range(len(t)):`.

OCaml

On parcourt les indices :

```
for i = 0 to Array.length t - 1 do
    (* traitement de t.(i) *)
done
```

On peut aussi utiliser `Array.iter (fun x -> ...) t`, ou `List.iter` pour une liste, qui appliquent une fonction à chaque élément.
On distingue alors deux schémas classiques selon ce que l'on cherche à calculer.

Accumulateur

On utilise une variable, l'**accumulateur**, qui contient le résultat pour les éléments déjà parcourus et qu'on met à jour à chaque élément. Pour la somme des éléments d'un tableau :

C

```
int somme(int t[], int n)
{
    int s = 0;
    for (int i = 0; i < n; i++)
    {
        s = s + t[i];
    }
    return s;
}
```

Python

```
def somme(t):
    s = 0
    for x in t:
        s = s + x
    return s
```

OCaml

```
let somme t =
    let s = ref 0 in
    for i = 0 to Array.length t - 1 do
        s := !s + t.(i)
    done;
    !s
```

La propriété « après avoir traité les k premiers éléments, `s` contient leur somme » est vraie avant la boucle ($k = 0$, la somme vide vaut 0) et reste vraie après chaque itération. C'est un **invariant** de boucle, notion qu'on précisera plus bas. En fin de boucle, il assure que `s` contient la somme de tous les éléments. En particulier, la fonction renvoie bien 0 pour un tableau vide.

Le choix de la valeur initiale est crucial. Pour le maximum, il n'y a pas d'élément neutre parmi les entiers : on initialise avec le premier élément, ce qui suppose que le tableau est non vide.

C

```
int maximum(int t[], int n)
{
    assert(n > 0);
    int m = t[0];
    for (int i = 0; i < n; i++)
    {
        if (t[i] > m)
        {
            m = t[i];
        }
```

```
    }
    return m;
}
```

Python

```
def maximum(t):
    m = t[0]
    for x in t:
        if x > m:
            m = x
    return m
```

OCaml

```
let maximum t =
    let m = ref t.(0) in
    for i = 0 to Array.length t - 1 do
        if t.(i) > !m
        then m := t.(i)
    done;
    !m
```

Remarque 2.7 Initialiser `m` à `0` serait une erreur : sur un tableau ne contenant que des nombres négatifs, la fonction renverrait 0, qui n'est même pas un élément du tableau.

Drapeau

Pour savoir si **au moins un** élément vérifie une propriété, on utilise un booléen, appelé **drapeau**, initialisé à faux et qu'on lève, c'est-à-dire qu'on passe à vrai, dès qu'on rencontre un tel élément. Pour tester si `x` apparait dans `t` :

C

```
bool appartient(int x, int t[], int n)
{
    bool trouve = false;
    for (int i = 0; i < n; i++)
    {
        if (t[i] == x)
        {
            trouve = true;
        }
    }
    return trouve;
}
```

Python

```
def appartient(x, t):
    trouve = False
    for y in t:
        if y == x:
            trouve = True
    return trouve
```

OCaml

```
let appartient x t =
    let trouve = ref false in
    for i = 0 to Array.length t - 1 do
        if t.(i) = x
        then trouve := true
    done;
    !trouve
```

Remarque 2.8 Une erreur classique consiste à écrire `trouve = (y == x)` dans la boucle : le drapeau est alors baissé dès qu'un élément suivant est différent de `x`, et la fonction ne teste plus que le dernier élément.

Pour tester si **tous** les éléments vérifient une propriété, on fait l'inverse : un drapeau initialisé à vrai qu'on baisse dès qu'on trouve un contre-exemple.

Ici, on continue à parcourir le tableau même après avoir trouvé `x`. En `C` et en `Python`, on peut interrompre la fonction avec `return` dès qu'on l'a trouvé, ce qu'on verra dans la partie sur les fonctions. En `OCaml`, on peut utiliser une boucle `while` ou les exceptions.

2.4.3 Répéter tant qu'une condition est vérifiée

Il arrive qu'on ne connaisse pas à l'avance le nombre de répétitions. On utilise alors une boucle **tant que**, qui teste une condition avant chaque itération et s'arrête dès qu'elle est fausse. Par exemple, pour calculer la plus petite puissance de 2 supérieure ou égale à un entier n :

C

```c
int p = 1;
while (p < n)
{
    p = 2 * p;
}
```

Python

```python
p = 1
while p < n:
    p = 2 * p
```

OCaml

```
let p = ref 1 in
while !p < n do
    p := 2 * !p
done
```

Si la condition est fausse dès le départ, ici si $n \leq 1$, le corps n'est jamais exécuté. À l'inverse, rien ne garantit qu'une boucle tant que s'arrête : si on avait écrit `p = p * 1` par erreur, ou initialisé `p` à 0, la boucle tournerait indéfiniment. Il faut donc justifier, pour chaque boucle, qu'elle **termine** et qu'elle calcule bien ce qu'on attend, ce qu'on appelle sa **correction**.

Variant et invariant de boucle

Définition 2.9

- Un **variant** de boucle est une quantité entière, dépendant de l'état, qui est positive ou nulle tant que la condition de boucle est vraie et qui décroît strictement à chaque itération.
- Un **invariant** de boucle est une propriété de l'état qui est vraie avant la première itération et qui, si elle est vraie au début d'une itération, l'est encore à la fin de celle-ci.

Théorème 2.10

- Une boucle qui admet un variant termine.
- Si une boucle termine, un invariant est vrai en sortie de boucle, en même temps que la négation de la condition de boucle.

Preuve

- Si la boucle effectuait une infinité d'itérations, les valeurs successives du variant formeraient une suite infinie strictement décroissante d'entiers naturels, ce qui est impossible.
- Par récurrence sur le nombre k d'itérations effectuées, l'invariant est vrai après k itérations : c'est la définition pour $k = 0$ et pour le passage de k à $k + 1$. On sort de la boucle quand la condition est fausse.

□

Ainsi, pour prouver qu'une boucle est correcte, on cherche un invariant qui, combiné à la négation de la condition, donne le résultat voulu.

Pour prouver qu'une quantité est un variant, on vérifie donc trois points : elle est **entière**, elle est **positive** tant qu'on entre dans la boucle et elle **diminue strictement** à chaque itération. Pour un invariant I, on vérifie l'**initialisation** (I est vraie avant la première itération) et l'**hérédité** (I vraie en début d'itération implique I vraie en fin d'itération).

Exemple 2.11

On calcule le quotient et le reste de la division euclidienne d'un entier $a \geq 0$ par un entier $b > 0$ par soustractions successives :

C

```c
int q = 0;
int r = a;
while (r >= b)
{
    q = q + 1;
    r = r - b;
}
// ici, q est le quotient et r le reste
```

Python

```python
def division(a, b):
    q = 0
    r = a
    while r >= b:
        q = q + 1
        r = r - b
    return (q, r)
```

OCaml

```ocaml
let division a b =
    let q = ref 0 in
    let r = ref a in
    while !r >= b do
        q := !q + 1;
        r := !r - b
    done;
    (!q, !r)
```

Sur l'entrée $(10, 4)$, les valeurs successives de (q, r) sont $(0, 10)$, $(1, 6)$ puis $(2, 2)$ et on renvoie $(2, 2)$: on a bien $10 = 2 \times 4 + 2$.

Terminaison. La quantité r est un variant :

- elle est entière car elle vaut a au départ et on ne fait que des soustractions d'entiers ;
- elle est positive car, si on entre dans la boucle, c'est que $r \geq b$, donc $r - b \geq 0$;
- elle diminue strictement car $b > 0$, donc $r - b < r$.

Correction. On considère l'invariant $I(q, r) \coloneqq bq + r = a$ et $r \geq 0$.

- *Initialisation* : $b \times 0 + a = a$ et $a \geq 0$, donc $I(0, a)$ est vérifié avant la première itération.
- *Hérédité* : si $I(q, r)$ est vérifié au début d'une itération, alors $a = bq + r = b(q + 1) + (r - b)$ et $r - b \geq 0$ puisqu'on est entré dans la boucle, donc $I(q + 1, r - b)$ est vérifié en fin d'itération.

En sortie de boucle, on a $I(q, r)$ et la négation de la condition, $r < b$. Ainsi $a = bq + r$ avec $0 \leq r < b$: (q, r) est bien le couple (quotient, reste) de la division euclidienne de a par b.

Exercice

On considère la fonction suivante, qui calcule a^n pour $n \geq 0$ par *exponentiation rapide* :

C

```c
int puissance(int a, int n)
{
    int p = 1;
    while (n > 0)
    {
        if (n % 2 == 1)
        {
            p = p * a;
        }
        a = a * a;
        n = n / 2;
    }
    return p;
}
```

Python

```python
def puissance(a, n):
    p = 1
    while n > 0:
        if n % 2 == 1:
            p = p * a
        a = a * a
        n = n // 2
    return p
```

OCaml

```ocaml
let puissance a n =
    let a = ref a in
    let n = ref n in
    let p = ref 1 in
    while !n > 0 do
        if !n mod 2 = 1
        then p := !p * !a;
        a := !a * !a;
        n := !n / 2
    done;
    !p
```

1. Dérouler la fonction sur l'entrée $(2, 5)$.
2. Donner un variant de boucle.
3. Montrer à l'aide d'un invariant de boucle que la fonction renvoie a^n.

Solution

1. Les valeurs successives de (a, n, p) sont $(2, 5, 1)$, $(4, 2, 2)$, $(16, 1, 2)$ puis $(256, 0, 32)$ et on renvoie $32 = 2^5$.
2. n est un variant : il est entier, positif, et si on entre dans la boucle, $n > 0$ donc $\lfloor n/2 \rfloor < n$.
3. On note α et η les valeurs initiales de a et n et on considère l'invariant $I(a, n, p) \coloneqq a^n p = \alpha^\eta$.
   - *Initialisation* : $\alpha^\eta \times 1 = \alpha^\eta$, donc $I(\alpha, \eta, 1)$ est vérifié.
   - *Hérédité* : si $I(a, n, p)$ est vérifié en début d'itération, deux cas se présentent. Si $n = 2n'$, alors $\alpha^\eta = a^n p = (a^2)^{n'} p$, donc $I(a^2, n', p)$. Si $n = 2n' + 1$, alors $\alpha^\eta = a^n p = (a^2)^{n'} (ap)$, donc $I(a^2, n', ap)$. Dans les deux cas, l'invariant est vérifié en fin d'itération.

   En sortie de boucle, $n = 0$ et $I(a, 0, p)$ donne $p = \alpha^\eta$.

Remarque : lors de la dernière itération, on calcule encore `a * a` alors que cette valeur ne servira plus. En C, cela peut provoquer un dépassement de capacité même quand le résultat tient dans un `int` (par exemple pour 7^{10}).

□

2.4.4 Choisir la structure de boucle adaptée

Une boucle `for` peut toujours se réécrire avec une boucle `while` en gérant l'indice à la main, et réciproquement on peut simuler beaucoup de boucles `while` avec un `for` qu'on interrompt. Mais choisir la bonne structure rend le programme plus lisible et évite des erreurs :

situation	boucle adaptée
nombre d'itérations connu avant d'entrer dans la boucle	`for` sur un intervalle d'entiers
traitement de chaque élément d'une collection	`for` sur les éléments (ou les indices)
arrêt dépendant d'une condition calculée au fil des itérations	`while`

Une boucle `for` sur un intervalle termine toujours, sans qu'on ait besoin de le prouver. Avec un `while`, on doit penser à la terminaison, et à ne pas oublier la mise à jour des variables qui interviennent dans la condition.

Remarque 2.12 En `Python`, on ne modifie pas la structure d'une collection, par exemple en supprimant des éléments d'un tableau, pendant qu'on la parcourt avec `for x in t` : le parcours sauterait alors des éléments. On construit plutôt une nouvelle collection.

2.4.5 Boucles imbriquées et dimensionnalité

Le corps d'une boucle peut lui-même contenir une boucle. Pour chaque itération de la boucle externe, la boucle interne est exécutée entièrement. Si la boucle externe fait n itérations et la boucle interne m, le corps de la boucle interne est donc exécuté $n \times m$ fois.

C'est la structure naturelle pour parcourir des données à deux dimensions, comme une matrice représentée par un tableau de tableaux : une boucle par dimension.

C

Une matrice de taille fixée se déclare `int m[3][4];` et on accède à ses éléments avec `m[i][j]`.

Python

On crée une matrice de `n` lignes et `p` colonnes remplie de 0 avec `[[0] * p for i in range(n)]` et on accède à ses éléments avec `m[i][j]`.

Attention, `[[0] * p] * n` semble faire la même chose mais crée un tableau contenant `n` références vers **le même** tableau ligne, comme on l'a vu dans la partie sur le partage : modifier `m[0][0]` modifierait alors toute la première colonne.

OCaml

On crée une matrice de `n` lignes et `p` colonnes remplie de 0 avec `Array.make_matrix n p 0` et on accède à ses éléments avec `m.(i).(j)`. Pour la même raison qu'en `Python`, il ne faut pas écrire `Array.make n (Array.make p 0)`.

La boucle interne peut aussi dépendre de l'indice de la boucle externe. Par exemple, pour compter les **inversions** d'un tableau, c'est-à-dire les paires d'indices $i < j$ telles que `t[i] > t[j]` :

C

```c
int compte_inversions(int t[], int n)
{
    int c = 0;
    for (int i = 0; i < n; i++)
```

```
    {
        for (int j = i+1; j < n; j++)
        {
            if (t[i] > t[j])
            {
                c = c + 1;
            }
        }
    }
    return c;
}
```

Python

```
def compte_inversions(t):
    n = len(t)
    c = 0
    for i in range(n):
        for j in range(i+1, n):
            if t[i] > t[j]:
                c = c + 1
    return c
```

OCaml

```
let compte_inversions t =
    let n = Array.length t in
    let c = ref 0 in
    for i = 0 to n - 1 do
        for j = i+1 to n - 1 do
            if t.(i) > t.(j)
            then c := !c + 1
        done
    done;
    !c
```

Un tableau est trié dans l'ordre croissant si et seulement s'il n'a aucune inversion. Ici, le test est effectué $\sum_{i=0}^{n-1}(n-1-i) = \frac{n(n-1)}{2}$ fois. Compter ainsi le nombre d'itérations des boucles imbriquées est la base de l'analyse de la complexité des algorithmes.

2.5 Abstraire

2.5.1 Définir et appeler des fonctions

On a déjà écrit plusieurs fonctions dans ce chapitre. Une **définition** de fonction comporte :

- un **nom**, qui permettra de l'appeler ;
- des **paramètres**, qui sont des noms désignant les valeurs sur lesquelles la fonction travaille ;
- un **corps**, c'est-à-dire les instructions à exécuter, qui produisent éventuellement une **valeur de retour**.

Un **appel** de fonction consiste à donner des valeurs, les **arguments**, aux paramètres et à exécuter le corps. La valeur de l'appel est la valeur de retour.

C

```
int cube(int x)
{
    return x * x * x;
}
```

Le type de retour (`int`) et le type de chaque paramètre sont précisés. On appelle la fonction avec `cube(3)`. Une fonction doit être déclarée avant son premier appel dans le fichier.

Python

```python
def cube(x):
    return x * x * x
```

On appelle la fonction avec `cube(3)`. Les parenthèses sont obligatoires, même sans argument : `f()`.

OCaml

```ocaml
let cube x =
    x * x * x
```

On appelle la fonction avec `cube 3`, sans parenthèses ni virgules : `f 2 3` pour une fonction à deux arguments. Une fonction sans argument utile prend `()` en paramètre, comme `print_newline ()`.
Il faut bien distinguer la **définition**, qui ne fait qu'associer le corps au nom sans rien exécuter, de l'**appel**, qui exécute le corps.

2.5.2 Action d'une fonction et valeur de retour

En `C` et en `Python`, l'instruction `return e` évalue `e`, **termine immédiatement** l'exécution de la fonction et renvoie la valeur obtenue. On peut s'en servir pour sortir d'une boucle dès qu'on connait le résultat, comme dans ce test de primalité :

C

```c
bool est_premier(int n)
{
    if (n < 2)
    {
        return false;
    }
    int d = 2;
    while (d * d <= n)
    {
        if (n % d == 0)
        {
            return false;
        }
        d = d + 1;
    }
    return true;
}
```

Python

```python
def est_premier(n):
    if n < 2:
        return False
    d = 2
    while d * d <= n:
        if n % d == 0:
            return False
        d = d + 1
    return True
```

OCaml

En `OCaml`, il n'y a pas de `return` : la valeur d'une fonction est celle de son corps. On intègre donc la condition d'arrêt à la boucle :

```ocaml
let est_premier n =
    let premier = ref (n >= 2) in
    let d = ref 2 in
    while !premier && !d * !d <= n do
        if n mod !d = 0
        then premier := false;
        d := !d + 1
```

```
    done;
    !premier
```

Pour retrouver un retour prématuré, on peut utiliser les exceptions.
On s'arrête à $\sqrt{n}$ car si $n = dd'$ avec $2 \leq d \leq d'$, alors $d^2 \leq n$: si n n'a pas de diviseur d avec $2 \leq d$ et $d^2 \leq n$, il est premier.

Une fonction peut aussi agir sur son environnement sans renvoyer de valeur intéressante : on parle d'**effet de bord**. C'est le cas d'un affichage ou de la modification d'un tableau passé en argument, comme l'échange de deux cases :

C

```c
void echange(int t[], int i, int j)
{
    int temp = t[i];
    t[i] = t[j];
    t[j] = temp;
}
```

Le type de retour `void` indique que la fonction ne renvoie rien.

Python

```python
def echange(t, i, j):
    temp = t[i]
    t[i] = t[j]
    t[j] = temp
```

Une fonction sans `return` renvoie `None`.

OCaml

```
let echange t i j =
    let temp = t.(i) in
    t.(i) <- t.(j);
    t.(j) <- temp
```

Le type de `echange` est `'a array -> int -> int -> unit`.

2.5.3 Fonctions et portée

Les paramètres et les variables définies dans le corps d'une fonction sont **locaux** : ils n'existent que pendant l'appel et sont distincts des variables de même nom ailleurs dans le programme. Chaque appel dispose de ses propres variables locales, rangées dans la pile d'exécution, comme on le détaille dans le chapitre sur la récursivité.

Lors d'un appel, la **valeur** de chaque argument est recopiée dans le paramètre correspondant. Modifier un paramètre dans la fonction ne change donc pas la variable de l'appelant :

```python
def incremente(x):
    x = x + 1    # ne modifie que la variable locale x

a = 3
incremente(a)    # a vaut toujours 3
```

En revanche, pour un tableau, ce qui est recopié est la référence vers le tableau, comme on l'a vu dans la partie sur le partage : la fonction et l'appelant manipulent alors **le même** tableau. C'est pourquoi `echange` modifie bien le tableau de l'appelant. Il en va de même en `C`, où un tableau passé en argument n'est pas recopié, et en `OCaml` pour les tableaux et les références.

> **Remarque 2.13** En `Python`, une fonction peut lire une variable globale, mais une assignation dans le corps crée une variable locale, sauf si on a écrit `global x` au début de la fonction. Utiliser des variables globales

rend le comportement d'une fonction dépendant de tout le programme : on préfère passer explicitement les valeurs en arguments et récupérer le résultat avec `return`.

2.5.4 Structurer des programmes avec des fonctions

Pour écrire un programme, on le découpe en fonctions courtes, chacune résolvant un sous-problème clairement identifié. Chaque fonction est décrite par sa **spécification** :

- ce qu'elle prend en entrée, avec les hypothèses faites sur ces entrées, appelées **préconditions** (« `t` est non vide », « $b > 0$ », ...) ;
- ce qu'elle renvoie ou modifie, appelé **postcondition**.

Une fois une fonction écrite et prouvée, on peut l'utiliser en ne se souciant que de sa spécification, sans relire son code. Par exemple, pour compter les nombres premiers inférieurs ou égaux à n :

C

```c
/* n : entier
   renvoie le nombre d'entiers premiers dans [0, n] */
int nb_premiers(int n)
{
    int c = 0;
    for (int k = 0; k <= n; k++)
    {
        if (est_premier(k))
        {
            c = c + 1;
        }
    }
    return c;
}
```

Python

```python
def nb_premiers(n):
    """n : entier
    renvoie le nombre d'entiers premiers dans [0, n]"""
    c = 0
    for k in range(n+1):
        if est_premier(k):
            c = c + 1
    return c
```

OCaml

```ocaml
(* n : entier
   renvoie le nombre d'entiers premiers dans [0, n] *)
let nb_premiers n =
    let c = ref 0 in
    for k = 0 to n do
        if est_premier k
        then c := !c + 1
    done;
    !c
```

Si on trouve un test de primalité plus efficace, il suffira de modifier `est_premier` : `nb_premiers` n'aura pas à changer.

On peut vérifier les préconditions à l'exécution avec `assert`, comme dans la fonction `maximum` en `C`, et tester une fonction sur des cas dont on connait le résultat, en pensant aux cas limites : `assert nb_premiers(1) == 0`, `assert nb_premiers(2) == 1`, `assert nb_premiers(100) == 25`.

2.5.5 Les fonctions comme valeurs

Revenons à `somme_carres`. Si on veut maintenant calculer $\sum_{i=1}^{n} i^3$ ou $\sum_{i=1}^{n} \frac{1}{i}$, on va réécrire la même boucle en changeant seulement le terme ajouté. Pour abstraire ce terme, on passe la **fonction** qui le calcule en argument : une fonction est alors une valeur comme une autre.

C

Pour information, `C` permet de passer en argument un *pointeur de fonction*, avec une syntaxe assez lourde :

```c
int somme_f(int (*f)(int), int n)
{
    int s = 0;
    for (int i = 1; i <= n; i++)
    {
        s = s + f(i);
    }
    return s;
}
```

On appelle alors `somme_f(cube, 5)`.

Python

```python
def somme_f(f, n):
    s = 0
    for i in range(1, n+1):
        s = s + f(i)
    return s
```

On peut appeler `somme_f(cube, 5)` ou, sans nommer la fonction, `somme_f(lambda i: i * i, 5)` qui renvoie 55.

OCaml

C'est l'usage naturel en `OCaml`, où les fonctions sont des valeurs comme les autres :

```
let somme_f f n =
    let s = ref 0 in
    for i = 1 to n do
        s := !s + f i
    done;
    !s
```

On peut appeler `somme_f cube 5` ou `somme_f (fun i -> i * i) 5` qui renvoie 55. Le type inféré est `(int -> int) -> int -> int`.

On retrouve ici le fait que la notation $\sum_{i=1}^{n} f(i)$ est elle-même une opération qui prend une fonction f en argument. Ces fonctions qui prennent ou renvoient des fonctions, dites d'**ordre supérieur**, sont au cœur de la programmation fonctionnelle.

CHAPITRE 3

Récursivité

DANS CE CHAPITRE

3.1 La récursivité
3.2 L'arbre d'appels
3.3 Récursivité terminale
3.4 Types inductifs et induction structurelle

> **Note** **Roadmap** :
>
> - finir l'écriture.

3.1 La récursivité

3.1.1 Principe et exemples

On dit qu'une fonction est récursive lorsqu'elle va s'appeler elle-même lors de son exécution. Le plus souvent, cet appel sera directement visible depuis le corps de la fonction.

Par exemple, la fonction `fact` suivante permettant de calculer $n!$ est récursive :

OCaml

```
let rec fact n =
    if n = 0
    then 1
    else n * fact (n-1)
```

C

```
int fact(int n)
{
    if (n == 0)
    {
        return 1;
    }
    else
    {
        return n * fact(n-1);
    }
}
```

Python

```
def fact(n):
    if n == 0:
        return 1
```

```
    else:
        return n * fact(n-1)
```

On utilise ici un exemple mathématique pour la simplicité des fonctions écrites. En effet, on ne voit rien d'autre dans `fact` que ces appels récursifs et ils correspondent directement à la définition mathématique :

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \times (n-1)! & \text{sinon.} \end{cases}$$

Remarque 3.1 En fait, la définition mathématique se fait plutôt en posant :

$$0! = 1 \qquad (n+1)! = (n+1) \times n!$$

Donc, pour retraduire une telle définition en tant que fonction, il est souvent nécessaire de décaler les rangs.

On remarque tout de suite deux choses :

- il y a une valeur, $n = 0$, pour laquelle on ne fait aucun appel
- dans les autres cas, quand on fait un appel, l'argument $n - 1$ diminue strictement.

Cela permet d'être sûr que, pour un argument $n \geq 0$, la suite des appels s'arrête : les arguments successifs $n, n-1, \ldots$ forment une suite strictement décroissante d'entiers naturels, qui ne peut donc pas être infinie, et elle s'arrête quand on atteint 0. Attention, pour $n < 0$, on n'atteint jamais le cas de base et `fact n` ne termine pas (en pratique, la pile d'exécution déborde). On verra dans la suite comment on peut s'en assurer pour des fonctions plus complexes.

Quand on évalue `fact 5` on va effectuer une série d'appels :

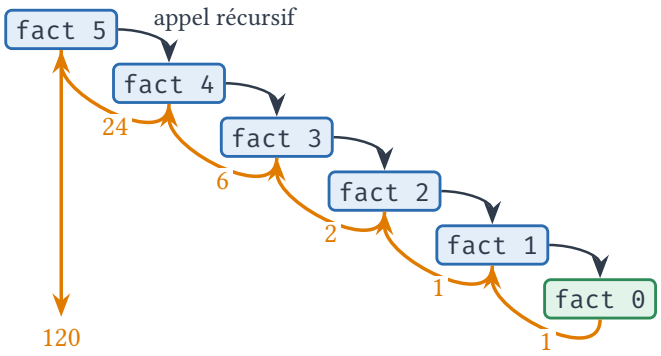


Figure 3.1 — Appels récursifs de fact 5 jusqu'au cas de base, puis remontée des valeurs

Pour calculer la valeur il faut descendre le long des appels récursifs jusqu'à tomber sur un cas de base puis remonter.

3.1.2 Implémentation pratique

Que se passe-t-il quand on appelle une fonction ?

Le principe de définition et d'appels de fonctions est à la base de la programmation structurée : au lieu d'avoir une longue succession d'instructions ou d'expressions, on les regroupe au sein de fonctions pour s'organiser. Cela permet également de *factoriser* du code : au lieu de dupliquer, on regroupe le même code au sein d'une fonction.

La question qu'on se pose ici est celle du mécanisme permettant une telle exécution du code. On pourrait naïvement penser qu'appeler une fonction serait absolument équivalent à copier son corps, c'est-à-dire que le programme

```
def f(x):
    y = x * x
    return x + y
```

```
a = 3
b = f(a)
print(b)
```

serait équivalent à

```
a = 3
### Début de f
x = a
y = x * x
### Fin de f et valeur de retour
b = x + y
print(b)
```

Au début de l'informatique, c'était effectivement la manière dont on appelait les fonctions. Mais on peut se rendre compte facilement du problème en changeant les noms de variables : le programme

```
def f(x):
    a = x * x
    return x + a

a = 3
b = f(a)
print(b)
```

deviendrait

```
a = 3
### Début de f
x = a
a = x * x
### Fin de f et valeur de retour
b = x + a
print(b)
```

Ici, l'appel de la fonction a changé la valeur de `a` (qui vaut 9 après l'appel) car `a` était aussi une variable définie dans `f`. Il est nécessaire d'avoir un mécanisme sophistiqué permettant de sauvegarder le contexte d'exécution d'une fonction et de le restaurer quand l'appel à une fonction est terminé.

Comme les fonctions vont elles-mêmes appeler d'autres fonctions, on ne peut pas se contenter de ne garder qu'un seul contexte. La bonne solution est d'utiliser ce qu'on appelle une **pile d'exécution**. Avant d'évaluer une fonction, on empile son contexte sur la pile, quand elle a terminé, on le dépile. On reverra plus tard la structure de données **pile** mais, pour le moment, on peut se l'imaginer comme une pile d'assiettes à laver : au départ, elle est vide. Chaque fois qu'on empile une assiette, elle grandit. Quand on veut laver une assiette, on prend nécessairement la dernière assiette empilée.

Un exemple

Si on considère le programme :

```
def f():
    return 4

def g(x):
    v = x+f()
    return v

def h(x):
    v = g(x+1) ** 2
    return v

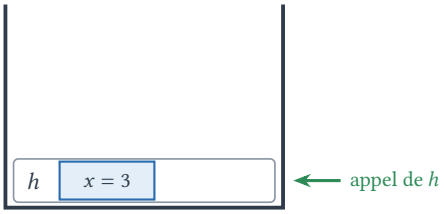
print(h(3))
```

Il va s'exécuter ainsi :

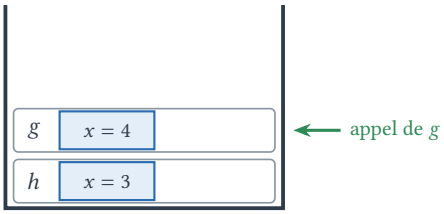
- Au départ la pile d'exécution est vide :



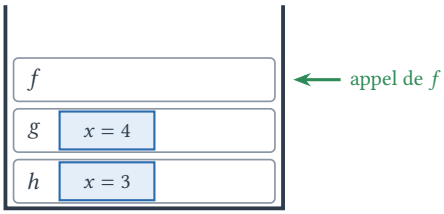
- Ligne 12 : on appelle `h`, on crée donc sur la pile une entrée pour `h`.



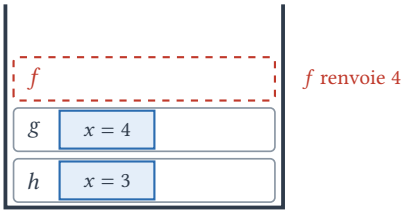
- Ligne 9 : on appelle `g` depuis `h`, on crée donc sur la pile une entrée pour `g`. On remarque notamment que `x` n'aura pas le même sens et la même valeur dans `g` que dans `h`.



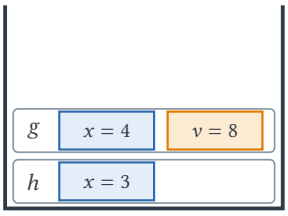
- Ligne 5 : on appelle `f`, on crée donc sur la pile une entrée pour `f`.



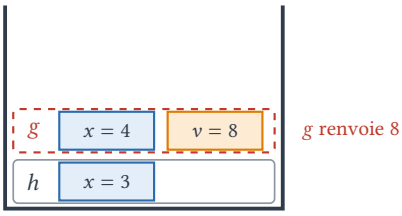
- Ligne 2 : on sort de `f` en renvoyant la valeur 4. On dépile ainsi l'environnement de `f`.



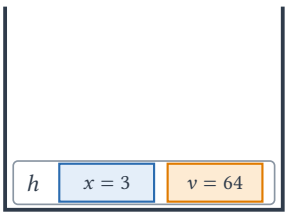
- Ligne 5 : on définit une variable `v` locale à `g` avec la valeur `x + 4`, où 4 est la valeur de retour de `f`, soit 8.



- Ligne 6 : on sort de `g` en renvoyant la valeur 8. On dépile ainsi l'environnement de `g`.



- Ligne 9 : on définit une variable `v` locale à `h` avec la valeur de retour de `g` élevée au carré, soit 64 :



- Ligne 10 : l'appel de `h` est fini, on le dépile.
- Ligne 12 : on a récupéré la valeur de retour 64, on l'affiche avec un appel à `print` (qui sera aussi empilé).

Pour une fonction récursive

Le mécanisme mis en place est suffisamment robuste pour permettre à une fonction de s'appeler elle-même. En effet, le fait d'avoir bien séparé les environnements d'exécution assure qu'il n'y aura pas de problèmes.

Si on reprend l'exemple vu au-dessus de l'évaluation de 5!, on remarque que lorsqu'on descend le long des appels, on empile, et lorsqu'on remonte, on dépile.

3.1.3 Programmer en récursif

Pour écrire une fonction récursive, on raisonne comme pour une preuve par récurrence : on **suppose** qu'on sait résoudre le problème sur des instances plus petites et on cherche à en déduire la solution de l'instance courante. Il faut alors :

- traiter directement un ou plusieurs **cas de base**, pour lesquels on ne fait aucun appel ;
- dans les autres cas, se ramener à des instances **strictement plus petites**, en un sens qu'il faudra préciser.

Exemple 3.2

On veut calculer x^n pour un entier $n \geq 0$. La définition $x^0 = 1$ et $x^n = x \times x^{n-1}$ donne directement une fonction récursive similaire à `fact`. Mais on peut aussi remarquer que

$$x^n = \begin{cases} 1 & \text{si } n = 0 \\ \left(x^{\lfloor n/2 \rfloor}\right)^2 & \text{si } n \text{ est pair} \\ x \times \left(x^{\lfloor n/2 \rfloor}\right)^2 & \text{si } n \text{ est impair} \end{cases}$$

ce qui permet de se ramener à l'instance $\lfloor n/2 \rfloor$, bien plus petite que $n-1$. C'est l'**exponentiation rapide**.

OCaml

```
let rec puissance x n =
    if n = 0
    then 1
    else
        let y = puissance x (n/2) in
        if n mod 2 = 0
        then y * y
        else x * y * y
```

C

```
int puissance(int x, int n)
{
    if (n == 0)
    {
        return 1;
    }
    int y = puissance(x, n/2);
    if (n % 2 == 0)
    {
        return y * y;
    }
    else
    {
        return x * y * y;
    }
}
```

Python

```
def puissance(x, n):
    if n == 0:
        return 1
    y = puissance(x, n // 2)
    if n % 2 == 0:
        return y * y
    else:
        return x * y * y
```

Remarque 3.3 On a pris soin de stocker le résultat de l'appel récursif dans `y`. Écrire `puissance x (n/2) * puissance x (n/2)` donnerait le même résultat mais ferait deux appels récursifs au lieu d'un : on verra avec l'arbre d'appels que cela fait perdre tout l'intérêt de la méthode.

Prouver la correction par récurrence

La structure d'une fonction récursive se prête naturellement à une preuve par récurrence : l'hypothèse de récurrence porte sur les appels récursifs.

Propriété 3.4

Pour tout entier $n \geq 0$, `fact n` renvoie $n!$.

Preuve Par récurrence sur n. Pour $n = 0$, `fact 0` renvoie $1 = 0!$.

Soit $n \geq 1$ tel que `fact (n-1)` renvoie $(n-1)!$. Comme $n \neq 0$, `fact n` renvoie n multiplié par la valeur de `fact (n-1)`, soit $n \times (n-1)! = n!$.

□

Propriété 3.5

Pour tout entier x et tout entier $n \geq 0$, `puissance x n` renvoie x^n.

Preuve Par récurrence forte sur n. Pour $n = 0$, la fonction renvoie $1 = x^0$.

Soit $n \geq 1$ et supposons le résultat vrai pour tout $k < n$. Comme $\lfloor n/2 \rfloor < n$, l'hypothèse de récurrence assure que $y = x^{\lfloor n/2 \rfloor}$.

- Si n est pair, $n = 2\lfloor n/2 \rfloor$ et la fonction renvoie $y^2 = x^n$.
- Si n est impair, $n = 2\lfloor n/2 \rfloor + 1$ et la fonction renvoie $xy^2 = x^n$.

□

Exercice

On considère la fonction suivante, où `t` est un tableau et `n` un entier tel que $0 \leq n \leq$ la longueur de `t` :

OCaml

```
let rec croissant t n =
    if n = 0 || n = 1
    then true
    else t.(n-2) <= t.(n-1) && croissant t (n-1)
```

C

```
bool croissant(int t[], int n)
{
    if (n == 0 || n == 1)
    {
        return true;
    }
    else
    {
        return t[n-2] <= t[n-1] && croissant(t, n-1);
    }
}
```

Python

```
def croissant(t, n):
    if n == 0 or n == 1:
        return True
    else:
        return t[n-2] <= t[n-1] and croissant(t, n-1)
```

1. Montrer par récurrence que l'appel `croissant t n` termine.
2. Montrer par récurrence que `croissant t n` renvoie vrai si et seulement si $t_0 \leq t_1 \leq \ldots \leq t_{n-1}$.

Solution

1. Pour $n \in \{0, 1\}$, il n'y a pas d'appel récursif. Si l'appel pour $n \geq 1$ termine, l'appel pour $n + 1$ effectue une comparaison puis éventuellement l'appel pour n, qui termine. On aurait pu se contenter de remarquer que n diminue strictement à chaque appel en restant dans $\mathbb{N}$: c'est la version récursive de la notion de variant de boucle.
2. On montre par récurrence sur n que `croissant t n` est vrai si et seulement si $\forall i \in [\![0, n-2]\!], t_i \leq t_{i+1}$. Pour $n \in \{0, 1\}$, il n'y a rien à comparer et la fonction renvoie vrai. Supposons la propriété vraie au rang $n \geq 1$. Au rang $n + 1$, la fonction renvoie vrai si et seulement si $t_{n-1} \leq t_n$ et `croissant t n` est vrai, c'est-à-dire, par hypothèse de récurrence, si et seulement si $t_{n-1} \leq t_n$ et $\forall i \in [\![0, n-2]\!], t_i \leq t_{i+1}$, ce qui équivaut à $\forall i \in [\![0, n-1]\!], t_i \leq t_{i+1}$.

□

Remarque 3.6 On a raisonné ici sur des entiers mathématiques. Avec les entiers machine de `C` ou d'`OCaml`, le résultat n'est correct que modulo les dépassements de capacité, par exemple `fact 21` n'est pas 21! en `OCaml` sur 63 bits. En `Python`, les entiers étant en précision arbitraire, il n'y a pas ce problème.

3.1.4 Récursivité croisée

Il arrive que deux fonctions (ou plus) s'appellent l'une l'autre : on parle de récursivité **croisée** ou **mutuelle**. Un exemple volontairement simple est celui de la parité d'un entier naturel : n est pair si $n = 0$ ou si $n - 1$ est impair, et n est impair si $n \neq 0$ et $n - 1$ est pair.

OCaml

En `OCaml`, une fonction ne peut faire référence qu'à des noms déjà définis. On définit donc les deux fonctions simultanément avec `let rec ... and ...` :

```
let rec est_pair n =
    if n = 0
    then true
    else est_impair (n-1)
and est_impair n =
    if n = 0
    then false
    else est_pair (n-1)
```

C

En `C`, une fonction doit être déclarée avant d'être utilisée. On écrit donc d'abord le **prototype** de `est_impair` pour pouvoir l'appeler dans `est_pair` :

```
bool est_impair(int n);

bool est_pair(int n)
{
    if (n == 0)
    {
        return true;
    }
    return est_impair(n-1);
}

bool est_impair(int n)
{
    if (n == 0)
    {
        return false;
    }
    return est_pair(n-1);
}
```

Python

En `Python`, le nom `est_impair` n'est cherché qu'au moment de l'appel, il n'y a donc rien de particulier à faire :

```
def est_pair(n):
    if n == 0:
        return True
    return est_impair(n-1)

def est_impair(n):
    if n == 0:
        return False
    return est_pair(n-1)
```

On retrouvera ce schéma de manière beaucoup plus naturelle avec des types de données mutuellement récursifs ou en analyse syntaxique, où chaque catégorie de la grammaire (expression, terme, facteur, ...) correspond à une fonction.

3.2 L'arbre d'appels

3.2.1 Définition

Définition 3.7

L'**arbre d'appels** associé à l'évaluation d'une fonction récursive sur un argument est l'arbre dont :

- la racine est l'appel initial ;
- les fils d'un nœud sont les appels récursifs effectués *directement* par cet appel, dans l'ordre où ils sont faits.

Les feuilles correspondent aux appels qui n'en font aucun autre, le plus souvent les cas de base.

Pour `fact 5`, l'arbre est un simple chemin, comme sur la figure du début du chapitre. Il en va autrement quand une fonction fait plusieurs appels récursifs, comme le calcul naïf du n-ième terme de la suite de Fibonacci définie par $F_0 = 0$, $F_1 = 1$ et $F_n = F_{n-1} + F_{n-2}$ pour $n \geq 2$:

OCaml

```
let rec fib n =
    if n <= 1
    then n
    else fib (n-1) + fib (n-2)
```

C

```
int fib(int n)
{
    if (n <= 1)
    {
        return n;
    }
    return fib(n-1) + fib(n-2);
}
```

Python

```
def fib(n):
    if n <= 1:
        return n
    return fib(n-1) + fib(n-2)
```

L'arbre d'appels de `fib 4` est alors :

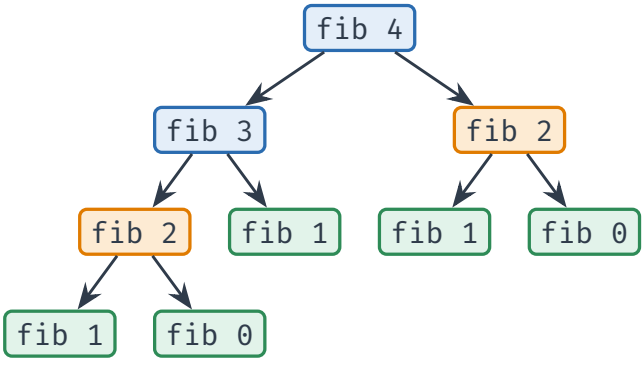


Figure 3.2 — Arbre d'appels de fib 4 : fib 2 est calculé deux fois

On y voit que `fib 2` est calculé deux fois et `fib 1` trois fois : la fonction refait sans cesse les mêmes calculs.

La pile d'exécution permet de relier cet arbre à l'exécution réelle : à chaque instant, la pile contient exactement les appels situés sur le chemin de la racine au nœud en cours d'évaluation. La hauteur de la pile est donc majorée par la hauteur de l'arbre plus un.

3.2.2 Complexité en nombre d'appels

Si chaque appel effectue, en dehors de ses appels récursifs, un nombre d'opérations borné par une constante, la complexité temporelle est proportionnelle au nombre de nœuds de l'arbre d'appels. On compte alors les appels en traduisant la structure de la fonction en une **équation de récurrence**.

Exemple 3.8

Notons $C(n)$ le nombre d'appels effectués lors de l'évaluation de `fact n`. On a $C(0) = 1$ et $C(n) = 1 + C(n-1)$ pour $n \geq 1$, d'où $C(n) = n + 1$.

Propriété 3.9

Notons $A(n)$ le nombre d'appels effectués lors de l'évaluation de `fib n`. Pour tout $n \geq 0$, $A(n) = 2F_{n+1} - 1$. En particulier, en notant $\varphi = \frac{1+\sqrt{5}}{2}$, on a $A(n) \geq 2\varphi^{n-1} - 1$: le nombre d'appels est exponentiel en n.

Preuve On a $A(0) = A(1) = 1$ et, pour $n \geq 2$, $A(n) = 1 + A(n-1) + A(n-2)$ en comptant l'appel lui-même puis les deux sous-arbres.

On montre $A(n) = 2F_{n+1} - 1$ par récurrence double. Pour $n = 0$ et $n = 1$: $2F_1 - 1 = 2F_2 - 1 = 1$. Si c'est vrai aux rangs $n-1$ et $n-2$, alors $A(n) = 1 + 2F_n - 1 + 2F_{n-1} - 1 = 2F_{n+1} - 1$.

Montrons ensuite $F_n \geq \varphi^{n-2}$ pour $n \geq 1$, encore par récurrence double : $F_1 = 1 \geq \varphi^{-1}$, $F_2 = 1 = \varphi^0$ et, pour $n \geq 3$, $F_n = F_{n-1} + F_{n-2} \geq \varphi^{n-3} + \varphi^{n-4} = \varphi^{n-4}(\varphi + 1) = \varphi^{n-2}$ car $\varphi^2 = \varphi + 1$. D'où $A(n) = 2F_{n+1} - 1 \geq 2\varphi^{n-1} - 1$.

□

Concrètement, `fib 30` effectue $2F_{31} - 1 = 2\,692\,537$ appels alors qu'une simple boucle calculant les termes successifs en ferait une trentaine. On verra avec la programmation dynamique comment éviter ces recalculs.

Propriété 3.10

Notons $P(n)$ le nombre d'appels effectués lors de l'évaluation de `puissance x n`. On a $P(0) = 1$ et, pour $n \geq 1$, $P(n) = \lfloor \log_2 n \rfloor + 2$.

Preuve On a $P(0) = 1$ et $P(n) = 1 + P(\lfloor n/2 \rfloor)$ pour $n \geq 1$. On procède par récurrence forte sur $n \geq 1$. Pour $n = 1$: $P(1) = 1 + P(0) = 2 = \lfloor \log_2 1 \rfloor + 2$.

Soit $n \geq 2$ et $k = \lfloor \log_2 n \rfloor \geq 1$, de sorte que $2^k \leq n < 2^{k+1}$. On a alors $2^{k-1} \leq \lfloor n/2 \rfloor < 2^k$, donc $\lfloor n/2 \rfloor \geq 1$ et $\lfloor \log_2 \lfloor n/2 \rfloor \rfloor = k - 1$. Par hypothèse de récurrence, $P(n) = 1 + (k-1) + 2 = k + 2$.

□

Avec la version maladroite évoquée plus haut qui fait deux appels `puissance x (n/2)`, l'équation devient $P'(n) = 1 + 2P'(\lfloor n/2 \rfloor)$ et on vérifie de même que $P'(2^k) = 2^{k+2} - 1$: on fait alors plus d'appels que la méthode naïve en $n + 1$ appels.

3.2.3 Terminaison

On a justifié la terminaison de `fact n` pour $n \geq 0$ en remarquant que l'argument décroît strictement dans $\mathbb{N}$. On généralise cette idée.

Définition 3.11

Soit f une fonction récursive dont les arguments appartiennent à un ensemble D. Un **variant** (ou une **mesure**) pour f est une application $\mu : D \to \mathbb{N}$ telle que, pour tout $a \in D$, chaque appel récursif effectué lors de l'évaluation de $f(a)$ se fait sur un argument $a' \in D$ vérifiant $\mu(a') < \mu(a)$.

Théorème 3.12

Si f admet un variant μ sur D et si, en dehors des appels récursifs, l'évaluation de $f(a)$ termine pour tout $a \in D$, alors l'évaluation de $f(a)$ termine pour tout $a \in D$.

Preuve On montre par récurrence forte sur $m \in \mathbb{N}$ la propriété : « pour tout $a \in D$ tel que $\mu(a) = m$, l'évaluation de $f(a)$ termine ».

Soit $m \in \mathbb{N}$ tel que la propriété soit vraie pour tout $m' < m$ et soit $a \in D$ avec $\mu(a) = m$. Chaque appel récursif fait par $f(a)$ porte sur un $a' \in D$ avec $\mu(a') < m$, il termine donc par hypothèse de récurrence. Comme le reste du calcul termine aussi, l'évaluation de $f(a)$ termine. Le cas $m = 0$ est contenu dans ce raisonnement : il ne peut alors y avoir aucun appel récursif.

□

Exemple 3.13

- Pour `puissance x n`, avec $D = \mathbb{N}$, $\mu(n) = n$ convient car $\lfloor n/2 \rfloor < n$ pour $n \geq 1$.
- Pour `est_pair` et `est_impair`, on applique le même raisonnement aux deux fonctions à la fois avec la mesure $\mu(n) = n$.
- Pour `fact`, l'ensemble D est important : sur $\mathbb{Z}$, $\mu(n) = n$ n'est pas à valeurs dans $\mathbb{N}$ et, effectivement, `fact (-1)` ne termine pas.

Parfois, aucune mesure entière simple ne s'impose. On utilise alors plus généralement un ensemble ordonné $(E, \preccurlyeq)$ **bien fondé**, c'est-à-dire dans lequel il n'existe pas de suite infinie strictement décroissante. La preuve s'adapte : si $f(a_0)$ ne terminait pas, l'un de ses appels récursifs $f(a_1)$ ne terminerait pas, puis l'un des appels de $f(a_1)$, et ainsi de suite, ce qui construirait une suite infinie strictement décroissante $\mu(a_0) \succ \mu(a_1) \succ \ldots$, contradiction.

Exemple 3.14

La fonction d'Ackermann est définie sur $\mathbb{N}^2$ par

$$A(m,n) = \begin{cases} n+1 & \text{si } m = 0 \\ A(m-1,1) & \text{si } m > 0 \text{ et } n = 0 \\ A(m-1, A(m,n-1)) & \text{sinon.} \end{cases}$$

Le premier argument ne décroît pas toujours (appel $A(m, n-1)$) et le second peut augmenter énormément (appel $A(m-1, A(m,n-1))$). En revanche, chaque appel fait décroître strictement le couple (m,n) pour l'**ordre lexicographique** sur $\mathbb{N}^2$: $(m-1, \cdot) < (m,n)$ et $(m, n-1) < (m,n)$.

Cet ordre est bien fondé : dans une suite strictement décroissante $(m_k, n_k)_k$, la suite (m_k) est décroissante dans $\mathbb{N}$ donc constante à partir d'un certain rang, et à partir de ce rang (n_k) est strictement décroissante dans $\mathbb{N}$, ce qui empêche la suite d'être infinie. Ainsi, l'évaluation de $A(m,n)$ termine toujours, même si elle peut prendre un temps colossal : $A(3,3) = 61$ mais $A(4,2) = 2^{65536} - 3$.

3.3 Récursivité terminale

3.3.1 Présentation

Quand on évalue `fact n`, il faut attendre le retour de l'appel `fact (n-1)` pour effectuer la multiplication par `n` : le contexte de l'appel courant doit être conservé sur la pile. C'est ce qui explique la phase de *remontée* vue au début du chapitre.

Définition 3.15

Un appel de fonction est **terminal** lorsque sa valeur est directement la valeur renvoyée par la fonction appelante, sans aucun calcul supplémentaire. Une fonction récursive est **récursive terminale** lorsque tous ses appels récursifs sont terminaux.

`fact` n'est pas récursive terminale à cause de la multiplication `n * ...`. Pour la rendre terminale, on effectue la multiplication **avant** l'appel en la stockant dans un argument supplémentaire, appelé **accumulateur** :

OCaml

```
let fact_term n =
    let rec aux n acc =
        if n = 0
        then acc
        else aux (n-1) (n * acc)
    in aux n 1
```

C

```
int fact_aux(int n, int acc)
{
    if (n == 0)
    {
        return acc;
    }
    return fact_aux(n-1, n * acc);
}

int fact_term(int n)
{
    return fact_aux(n, 1);
}
```

Python

```
def fact_term(n, acc=1):
    if n == 0:
        return acc
    return fact_term(n-1, n * acc)
```

L'évaluation de `fact_term 3` donne la suite d'appels `aux 3 1`, `aux 2 3`, `aux 1 6`, `aux 0 6` qui renvoie directement `6` : il n'y a plus rien à faire à la remontée.

> **Remarque 3.16** Pour prouver la correction, on ne peut pas montrer que `aux n acc` renvoie $n!$, ce qui est faux. On montre par récurrence sur n la propriété plus forte : pour tout entier `acc`, `aux n acc` renvoie $n! \times$ `acc`. On en déduit que `fact_term n` renvoie $n! \times 1$. Généraliser l'énoncé pour faire apparaître l'accumulateur est le point clé de ce type de preuve.

3.3.2 Optimisation

Lors d'un appel terminal, le contexte de la fonction appelante n'a plus aucune utilité puisqu'il n'y a plus rien à calculer après. Le compilateur peut donc **réutiliser** son emplacement sur la pile au lieu d'en empiler un nouveau : l'appel devient un simple saut au début de la fonction avec de nouveaux arguments. Une fonction récursive terminale s'exécute alors avec une pile de taille constante, exactement comme une boucle.

OCaml

`OCaml` garantit cette optimisation. On peut le constater sur le calcul de $\sum_{k=1}^{n} k$:

```
### let rec somme n = if n = 0 then 0 else n + somme (n-1);;
val somme : int -> int = <fun>
### somme 100_000_000;;
Stack overflow during evaluation (looping recursion?).
### let somme_term n =
    let rec aux n acc = if n = 0 then acc else aux (n-1) (acc + n) in
    aux n 0;;
val somme_term : int -> int = <fun>
### somme_term 100_000_000;;
- : int = 5000000050000000
```

La limite exacte dépend de la taille de pile autorisée, mais seule la version terminale fonctionne pour tout n.

C

La norme du langage `C` ne garantit rien. En pratique, `gcc` et `clang` effectuent cette optimisation lorsqu'on compile avec optimisations (`-O2`), mais il ne faut pas s'en remettre à cela pour la correction d'un programme : en `C`, on écrira plutôt une boucle.

Python

`Python` n'effectue **jamais** cette optimisation. De plus, la profondeur de la pile d'appels est limitée par défaut à 1000 :

```
>>> fact_term(10000)
Traceback (most recent call last):
  ...
RecursionError: maximum recursion depth exceeded
```

On peut augmenter cette limite avec `sys.setrecursionlimit`, mais pour des calculs de grande profondeur, on préfère une boucle.

3.3.3 Techniques

Accumulateur

La technique de l'accumulateur vue sur `fact` est très générale : on ajoute un argument qui contient le résultat partiel déjà calculé, et le cas de base renvoie cet accumulateur.

Sur les listes en `OCaml`, le parcours voit les éléments de gauche à droite et, quand on ajoute chaque élément en tête d'un accumulateur, le premier élément vu se retrouve au fond. C'est exactement ce qu'il faut pour renverser une liste :

```
let renverse l =
    let rec aux l acc =
        match l with
        | [] -> acc
        | t :: q -> aux q (t :: acc)
    in aux l []
```

Mais c'est un piège quand on veut conserver l'ordre : on renverse alors le résultat à la fin, ce qui reste linéaire :

```
let map_term f l =
    let rec aux l acc =
        match l with
        | [] -> List.rev acc
        | x :: q -> aux q (f x :: acc)
    in aux l []
```

Du récursif terminal à l'itératif

Une fonction récursive terminale se traduit mécaniquement en boucle : les arguments deviennent des variables, le cas de base devient la condition de sortie et l'appel récursif devient une mise à jour des variables.

OCaml

```
let fact_iter n =
    let n = ref n in
    let acc = ref 1 in
    while !n <> 0 do
        acc := !n * !acc;
        n := !n - 1
    done;
    !acc
```

C

```
int fact_iter(int n)
{
```

```
    int acc = 1;
    while (n != 0)
    {
        acc = n * acc;
        n = n - 1;
    }
    return acc;
}
```

Python

```
def fact_iter(n):
    acc = 1
    while n != 0:
        acc = n * acc
        n = n - 1
    return acc
```

Réciproquement, toute boucle `while` peut s'écrire comme une fonction récursive terminale dont les arguments sont les variables modifiées par la boucle. Récursif terminal et itératif sont donc deux écritures d'un même calcul.

> **Remarque 3.17** Pour une fonction qui n'est pas récursive terminale, comme `fib` ou un parcours d'arbre, le passage à l'itératif est plus délicat : il faut gérer soi-même une pile qui remplace la pile d'exécution. On le fera pour le parcours en profondeur d'un graphe.
>
> Le choix entre les deux écritures relève avant tout de la clarté : une fonction récursive qui suit la structure du problème est souvent plus simple à écrire et à prouver. On ne cherche à la rendre terminale que lorsque la profondeur des appels peut devenir grande, typiquement linéaire en la taille des données.

3.4 Types inductifs et induction structurelle

3.4.1 Définition naïve des types inductifs

Un **type inductif** est un type dont les valeurs sont construites à l'aide d'un nombre fini de **constructeurs**, chacun pouvant prendre des arguments, dont certains sont eux-mêmes du type que l'on définit. Naïvement, une valeur d'un tel type est obtenue en appliquant un nombre **fini** de fois ces constructeurs.

En `OCaml`, les listes en sont l'exemple de base : une liste est soit `[]`, soit `x :: q` où `q` est une liste. On peut définir de même les arbres binaires dont les nœuds sont étiquetés par des entiers :

```
type arbre = Vide | Noeud of arbre * int * arbre
```

Une fonction récursive sur un type inductif suit naturellement sa définition, avec un cas par constructeur et un appel récursif sur chaque argument du type lui-même :

```
let rec taille a =
    match a with
    | Vide -> 0
    | Noeud (g, _, d) -> 1 + taille g + taille d

let rec nb_vides a =
    match a with
    | Vide -> 1
    | Noeud (g, _, d) -> nb_vides g + nb_vides d
```

Une telle fonction termine toujours : le nombre de constructeurs utilisés pour écrire l'argument est un variant, puisque `g` et `d` en utilisent strictement moins que `Noeud (g, x, d)`.

Pour démontrer des propriétés de ces fonctions, on raisonne de la même manière.

Théorème 3.18

Soit P une propriété portant sur les valeurs du type `arbre`. Si

- $P(\texttt{Vide})$ est vraie,
- pour tous arbres g, d et tout entier x, $P(g)$ et $P(d)$ impliquent $P(\texttt{Noeud}\ (g, x, d))$,

alors $P(a)$ est vraie pour tout arbre a. C'est le principe d'**induction structurelle**.

Preuve On montre par récurrence forte sur $k \in \mathbb{N}$ que $P(a)$ est vraie pour tout arbre a écrit avec k constructeurs. Soit a un tel arbre. Si $a = \texttt{Vide}$, c'est la première hypothèse. Sinon, $a = \texttt{Noeud}\ (g, x, d)$ où g et d sont écrits avec strictement moins de k constructeurs : $P(g)$ et $P(d)$ sont vraies par hypothèse de récurrence, donc $P(a)$ par la seconde hypothèse.

□

Exemple 3.19

Pour tout arbre `a`, `nb_vides a = taille a + 1`. En effet, c'est vrai pour `Vide` ($1 = 0 + 1$) et si c'est vrai pour `g` et `d`, alors pour `a = Noeud (g, x, d)` on a `nb_vides a` $= (\texttt{taille g} + 1) + (\texttt{taille d} + 1) =$ `taille a` $+ 1$.

Le principe se généralise à tout type inductif : il y a une hypothèse par constructeur, avec une hypothèse d'induction pour chaque argument du type lui-même. Pour les listes, on retrouve : si $P([])$ et si $P(q)$ implique $P(\texttt{x} :: \texttt{q})$, alors P est vraie pour toute liste. Pour le type `type nat = Z | S of nat`, on retrouve exactement la récurrence sur $\mathbb{N}$.

CHAPITRE 4

Typage en OCaml

DANS CE CHAPITRE

4.1 Principe
4.2 Types génériques
4.3 Principe de l'inférence

4.1 Principe

En `OCaml`, chaque valeur a un unique type. On note `x : t` pour dire que `x` est du type `t`.

Quand on évalue une expression, `OCaml` répond ainsi avec son type :

```
### 2+3;;
- : int = 5
```

Le `-` signifie qu'on n'a pas nommé l'expression, si on écrit un `let`, on aura le nom associé (précédé de `val`) :

```
### let x = 2+3;;
val x : int = 5
```

`OCaml` déduit tout seul le type d'une expression par un algorithme appelé **l'inférence de types**.

Si on écrit `x+y`, comme l'opération plus ne peut se faire qu'avec des valeurs du type `int`, `OCaml` sait que `x` et `y` doivent être du type `int`.

Pour des fonctions, `OCaml` procède de même : il déduit le type des arguments et celui du résultat de la manière dont ils sont utilisés dans le corps de la fonction.

Ainsi `let f x y = x + y` va avoir le type `int -> int -> int` : `x` et `y` sont des `int` car ils sont arguments de `+`, et le résultat est un `int` car c'est le résultat d'une addition.

4.2 Types génériques

Quand on écrit `let f x = x`, `OCaml` n'a aucune contrainte sur le type de `x`. Dans ce cas, il utilise une variable de type `'a` signifiant n'importe quel type :

```
### let f x = x;;
val f : 'a -> 'a = <fun>
```

Quand `'a` apparait dans un type, on peut le remplacer par n'importe quel type **du moment qu'on remplace de même toutes ses occurrences**. On parle de *spécialisation* d'un type. Ainsi `int -> int` spécialise `'a -> 'a`.

4.3 Principe de l'inférence

On présente ici, de manière informelle, la méthode suivie par `OCaml` pour trouver le type d'une expression. Elle est due à Hindley et Milner et se décompose en trois temps :

1. on attribue une **variable de type** inconnue à chaque sous-expression dont on ne connait pas encore le type ;
2. on parcourt l'expression et chaque construction impose des **contraintes**, c'est-à-dire des équations entre types ;
3. on **résout** ces équations : c'est l'**unification**.

4.3.1 Variables de type et contraintes

Dans la suite, on note $\alpha, \beta, \gamma, \ldots$ les inconnues qui représentent des types non encore déterminés. Les contraintes proviennent des règles suivantes :

- une constante a un type connu : `1 : int`, `true : bool`, `(+) : int -> int -> int`, ... ;
- si `e1 e2` est une application avec `e1 : t1` et `e2 : t2`, on introduit une nouvelle inconnue α pour le type du résultat et on pose la contrainte $t_1 = t_2 \to \alpha$;
- pour `fun x -> e`, on donne une nouvelle inconnue α à `x` et, si `e : t`, l'expression a le type $\alpha \to t$;
- pour `if c then e1 else e2`, on pose `c : bool` et on impose que `e1` et `e2` aient le même type, qui est celui de l'expression.

Exemple 4.1

Considérons `let f g x = g (x + 1)`, c'est-à-dire `let f = fun g -> fun x -> g (x + 1)`.

- On pose `g` : α et `x` : β.
- Dans `x + 1`, l'opérateur `+` est de type `int -> int -> int` : on obtient les contraintes $\beta =$ `int` et `x + 1 : int`.
- Dans `g (x + 1)`, on applique `g` à une valeur de type `int`, on introduit donc une inconnue γ pour le résultat et la contrainte $\alpha =$ `int` $\to \gamma$.

Le type de `f` est $\alpha \to \beta \to \gamma$, qui devient, en remplaçant les inconnues déterminées, (`int` $\to \gamma$) $\to$ `int` $\to \gamma$. L'inconnue γ n'est soumise à aucune contrainte, elle devient une variable de type générique `'a` :

```
### let f g x = g (x + 1);;
val f : (int -> 'a) -> int -> 'a = <fun>
```

4.3.2 Unification

Résoudre les contraintes consiste à trouver une **substitution**, c'est-à-dire un remplacement de certaines inconnues par des types, qui rend chaque équation vraie. On traite les équations une par une :

- $t = t$: rien à faire, on la supprime ;
- $\alpha = t$ (ou $t = \alpha$) où α n'apparait pas dans t : on remplace α par t **dans toutes les autres équations** et on retient $\alpha := t$;
- $t_1 \to t_2 = t_1' \to t_2'$: on remplace cette équation par les deux équations $t_1 = t_1'$ et $t_2 = t_2'$ (de même pour les couples, les listes, ...) ;
- deux types de formes différentes, comme `int` = `bool` ou `int` $= t_1 \to t_2$: il n'y a pas de solution, c'est une **erreur de typage** ;
- $\alpha = t$ où α apparait dans t sans être égal à t : pas de solution non plus, car un type est une expression finie.

Exemple 4.2

Déroulons l'inférence sur `let compose f g x = f (g x)`.

- On pose `f` : α, `g` : β, `x` : γ.
- Pour `g x`, on introduit δ et la contrainte $\beta = \gamma \to \delta$.

- Pour `f (g x)`, on introduit ε et la contrainte $\alpha = \delta \to \varepsilon$.

Le type cherché est $\alpha \to \beta \to \gamma \to \varepsilon$. Les deux équations sont de la forme « inconnue = type » sans occurrence de l'inconnue à droite, on obtient la substitution $\beta := \gamma \to \delta$, $\alpha := \delta \to \varepsilon$ et le type

$$(\delta \to \varepsilon) \to (\gamma \to \delta) \to \gamma \to \varepsilon$$

Il reste trois inconnues libres que `OCaml` renomme dans l'ordre d'apparition : δ devient `'a`, ε devient `'b` et γ devient `'c`.

```
### let compose f g x = f (g x);;
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

Exemple 4.3

Pour `let h x = if x then 1 else x`, on pose `x` : α. La condition impose $\alpha =$ `bool` et les deux branches imposent `int` $= \alpha$. Après la substitution $\alpha :=$ `bool`, la seconde équation devient `int` = `bool` : erreur.

```
### let h x = if x then 1 else x;;
Line 1, characters 27-28:
1 | let h x = if x then 1 else x;;
                               ^

Error: The value x has type bool but an expression was expected of type int
```

Le message d'erreur indique l'endroit où l'unification a échoué, qui n'est pas forcément l'endroit où se trouve l'erreur du programmeur : ici, le `then 1` est peut-être le vrai coupable. C'est une source classique d'incompréhension, il faut alors relire tout le code qui a contribué au type.

Exemple 4.4

Pour `let k f = f f`, on pose `f` : α et l'application impose $\alpha = \alpha \to \beta$: α apparait à droite, il n'y a pas de solution.

```
### let k f = f f;;
Line 1, characters 12-13:
1 | let k f = f f;;
                ^

Error: The value f has type 'a -> 'b but an expression was expected of type
         'a
       The type variable 'a occurs inside 'a -> 'b
```

Remarque 4.5 On peut montrer que lorsque les contraintes admettent une solution, cette méthode trouve la plus générale : toute autre solution s'obtient en spécialisant celle-ci (admis). C'est ce qui permet à `OCaml` d'afficher **le** type d'une expression.

4.3.3 Polymorphisme du `let`

Une fois le type d'une définition `let x = e` obtenu, les inconnues qui y restent libres sont **généralisées** : elles deviennent des variables de type `'a`, `'b`, … qu'on pourra spécialiser différemment à chaque utilisation de `x`.

```
### let id x = x in (id 1, id true);;
- : int * bool = (1, true)
```

Ici, `id : 'a -> 'a` est utilisée une fois avec `'a = int` et une fois avec `'a = bool`.

En revanche, un argument de fonction n'est **pas** généralisé : à l'intérieur du corps, son type est une unique inconnue, qui ne peut être unifiée qu'avec un seul type.

```
### (fun id -> (id 1, id true)) (fun x -> x);;
Line 1, characters 21-25:
1 | (fun id -> (id 1, id true)) (fun x -> x);;
                         ^^^^
Error: The constructor true has type bool
       but an expression was expected of type int
```

L'utilisation `id 1` a imposé $\alpha = \texttt{int} \to \beta$, et `id true` impose alors $\texttt{int} = \texttt{bool}$. Pourtant, ce programme ne provoquerait aucune erreur à l'exécution : c'est une limite assumée de l'inférence, qui reste ainsi simple et efficace.

4.3.4 Restriction de valeur

La généralisation n'est pas toujours possible en présence de traits impératifs. Si on écrivait `let r = ref []` avec `r : 'a list ref`, on pourrait faire `r := [1]` puis utiliser `!r` comme une `bool list` : le typage ne garantirait plus rien.

Pour l'éviter, `OCaml` ne généralise que les définitions dont la partie droite est une **valeur** (une constante, une fonction `fun ...`, un constructeur appliqué à des valeurs, ...) et pas un calcul comme une application. Les inconnues non généralisées sont affichées `'_weak1`, `'_weak2`, ... : elles représentent **un** type précis pas encore connu, qui sera fixé à la première utilisation.

```
### let r = ref [];;
val r : '_weak1 list ref = {contents = []}
### r := [1];;
- : unit = ()
### r;;
- : int list ref = {contents = [1]}
```

Cela touche aussi les applications partielles. Dans ce cas, il suffit de rajouter l'argument explicitement pour retrouver une fonction, donc une valeur :

```
### let l = List.map (fun x -> x);;
val l : '_weak2 list -> '_weak2 list = <fun>
### let l2 x = List.map (fun x -> x) x;;
val l2 : 'a list -> 'a list = <fun>
```

TRAVAUX PRATIQUES 5

Rappels et types récursifs

DANS CE CHAPITRE

5.1 Fractions
5.2 Retour sur les listes
5.3 Installer OCaml et son écosystème sur votre ordinateur

5.1 Fractions

On va écrire ici un type et des fonctions pour manipuler des éléments de $\mathbb{Q}$. Pour pouvoir tester directement l'égalité, on va adopter une représentation irréductible des rationnels : pour tout $r \in \mathbb{Q}$, il existe un unique couple $(a, b) \in \mathbb{Z} \times \mathbb{N}^*$ tel que $r = \frac{a}{b}$ et $\text{pgcd}(a, b) = 1$.

Question 1 Écrire une fonction `pgcd : int -> int -> int` qui prend en entrée deux entiers relatifs a, b tels que $(a, b) \neq (0, 0)$ et renvoie leur pgcd (qui est positif).

On pourra utiliser les relations suivantes : $\text{pgcd}(a, 0) = |a|$ et si $b \neq 0$ et $a = bq + r$, on a $\text{pgcd}(a, b) = \text{pgcd}(b, r)$.

Tests

```
let test a b v =
    Printf.printf "Test %d ^ %d : " a b;
    let v' = pgcd a b in
    if v <> v'
    then Printf.printf "Erreur %d <> %d\n" v' v
    else Printf.printf "OK\n" in
test 1 10 1;
test 10 1 1;
test (-3) 0 3;
test 12 16 4;;
```

Corrigé

```
let rec pgcd a b =
    if b = 0
    then if a < 0 then -a else a
    else pgcd b (a mod b)
```

En `OCaml`, `a mod b` peut être négatif quand `a` l'est, mais on a toujours `a = b * (a / b) + a mod b`, ce qui suffit pour appliquer la relation précédente.

On va définir le type `OCaml` suivant :

```
type fraction = {
    num : int;
    den : int
}
```

Pour définir une fraction, il suffit donc d'écrire `{ num = 3; den = 2 }` et si f est une fraction, on peut utiliser `f.num` ou `f.den`.

Question 2 Écrire une fonction `est_irreductible : fraction -> bool` qui renvoie un booléen indiquant si une fraction est bien irréductible, c'est-à-dire $\frac{a}{b}$ avec pgcd $(a, b) = 1$ et $b > 0$.

Tests

```
assert(est_irreductible { num = 8; den = 27 });
assert(not (est_irreductible { num = 1; den = -1 }));
assert(not (est_irreductible { num = 12; den = 16 }));
Printf.printf "est_irreductible : OK\n";;
```

Corrigé

```
let est_irreductible f =
    pgcd f.num f.den = 1 && f.den > 0
```

Question 3 Écrire une fonction `simplifie : fraction -> fraction` qui, étant donnée une fraction $\frac{a}{b}$ (avec $b \neq 0$), renvoie une fraction irréductible $\frac{a'}{b'}$ telle que $\frac{a}{b} = \frac{a'}{b'}$.

Tests

```
let test f v =
    Printf.printf "Test %d/%d : " f.num f.den;
    let v' = simplifie f in
    if v <> v'
    then Printf.printf "Erreur %d/%d <> %d/%d\n"
        v'.num v'.den v.num v.den
    else Printf.printf "OK\n" in
test { num = 12; den = 4 } { num = 3; den = 1 };
test { num = 12; den = -4 } { num = -3; den = 1 };
test { num = 0; den = 3 } { num = 0; den = 1 };
test { num = -3; den = -2 } { num = 3; den = 2 };
test { num = 3; den = -2 } { num = -3; den = 2 };;
```

Corrigé

```
let simplifie f =
    let k = pgcd f.num f.den in
    let s = if f.den / k < 0 then -1 else 1 in
    {
        num = s * f.num / k;
        den = s * f.den / k
    }
```

Question 4 Écrire les fonctions suivantes en supposant que les fractions en entrée sont irréductibles et qui renvoient des fractions irréductibles :

- `frac_som : fraction -> fraction -> fraction` qui renvoie la somme de deux fractions
- `frac_mult : fraction -> fraction -> fraction` qui renvoie le produit de deux fractions
- `frac_neg : fraction -> fraction` qui renvoie l'opposée d'une fraction
- `frac_sub : fraction -> fraction -> fraction` qui renvoie la différence de deux fractions
- `frac_inv : fraction -> fraction` qui renvoie l'inverse d'une fraction non nulle

- `frac_div : fraction -> fraction -> fraction` qui renvoie le quotient de deux fractions

Tests

```
let f1 = { num = 1; den = 3 } in
let f2 = { num = 1; den = 2 } in
Printf.printf "Test frac_som\n";
assert(frac_som f1 f2 = { num = 5; den = 6 });
assert(frac_som f2 f2 = { num = 1; den = 1 });
Printf.printf "Test frac_mult\n";
assert(frac_mult f1 f2 = { num = 1; den = 6 });
Printf.printf "Test frac_neg\n";
assert(frac_neg f1 = { num = -1; den = 3 });
Printf.printf "Test frac_sub\n";
assert(frac_sub f1 f2 = { num = -1; den = 6 });
Printf.printf "Test frac_inv\n";
assert(frac_inv f1 = { num = 3; den = 1 });
Printf.printf "Test frac_div\n";
assert(frac_div f1 f2 = { num = 2; den = 3 });
Printf.printf "OK";;
```

Corrigé

```
let frac_som f1 f2 =
    simplifie {
        num = f1.num * f2.den + f1.den * f2.num;
        den = f1.den * f2.den
    }

let frac_mult f1 f2 =
    simplifie {
        num = f1.num * f2.num;
        den = f1.den * f2.den
    }

let frac_neg f =
    { num = -f.num; den = f.den }

let frac_sub f1 f2 =
    frac_som f1 (frac_neg f2)

let frac_inv f =
    if f.num = 0 then failwith "Division par zéro";
    if f.num > 0
    then { num = f.den; den = f.num }
    else { num = - f.den; den = - f.num }

let frac_div f1 f2 =
    frac_mult f1 (frac_inv f2)
```

Pour `frac_inv`, il faut faire attention au signe : l'inverse de $\frac{-2}{3}$ doit être représenté par $\frac{-3}{2}$ et non par $\frac{3}{-2}$, qui n'est pas irréductible au sens précédent.

5.2 Retour sur les listes

Remarque 5.1 On va ici reprendre le type récursif essentiel en `OCaml` : les listes.

Il s'agit de questions de révision qui reprennent parfois des fonctions déjà traitées dans les TP précédents.

On rappelle que les listes peuvent être vues comme définies par le type :

```
type 'a list = [] | (::) of 'a * 'a list
```

Ainsi, si `t` est une valeur de type `'a` et `q` une valeur de type `'a list`, alors `t::q` est une valeur de type `'a list` qu'on peut voir comme étant la liste qui commence par `t` et se poursuit par `q`.

Attention `::` n'est vraiment pas symétrique ! Comme pour $\in$ en mathématiques, il y a un déséquilibre : `element :: liste`.

Question 5 Écrire une fonction `ajoute_ind_pair : int list -> int` qui prend en entrée une liste d'entiers et renvoie la somme de ses éléments d'indices pairs.

Ainsi, si on passe la liste $[a_0; a_1; ...; a_{n-1}]$ cela va renvoyer $a_0 + a_2 + a_4 + ...$

Tests

```
let test l v =
    Printf.printf "Test [%s] : " (l |> List.map string_of_int |> String.concat ";");
    let v' = ajoute_ind_pair l in
    if v <> v'
    then Printf.printf "Erreur %d <> %d\n" v' v
    else Printf.printf "OK\n" in
test [] 0;
test [1] 1;
test [1;2] 1;
test [1;2;3] 4;
test [1;2;3;4] 4;;
```

Corrigé

```
let rec ajoute_ind_pair l =
    match l with
    | [] -> 0
    | [x] -> x
    | a::b::q -> a + ajoute_ind_pair q
```

Question 6 Écrire une fonction `mex : int list -> int` qui prend en entrée une liste d'entiers relatifs et renvoie le plus petit entier naturel, donc ≥ 0, qui n'est pas présent dans cette liste.

Il est facile de faire une fonction de complexité quadratique, mais essayez d'en faire une ensuite de complexité linéaire.

Tests

```
assert(mex [1;2;3] = 0);
assert(mex [0;1;3] = 2);
assert(mex [0;1;2] = 3);
assert(mex [-1;-4;2] = 0);
assert(mex [-1;-4;0] = 1);
Printf.printf "OK\n";;
```

Corrigé

```
(* Version quadratique *)
let mex l =
    let rec aux n =
        if List.mem n l
        then aux (n+1)
        else n
    in aux 0

(* Version linéaire *)
let mex l =
    let n = List.length l in
```

```
    let a = Array.make (n+1) false in
    List.iter (fun i ->
        if 0 <= i && i <= n
        then a.(i) <- true) l;
    let m = ref 0 in
    while a.(!m) do
        incr m
    done;
    !m
```

La version linéaire repose sur le fait que, si la liste est de longueur n, l'un des entiers $0, 1, \ldots, n$ est absent : le mex est donc au plus n et on peut ignorer les valeurs hors de $[\![0, n]\!]$.

Question 7 Écrire une fonction `coupures : 'a list -> ('a list * 'a list) list` qui, étant donné une liste `l`, renvoie, sous forme d'une liste, la liste de tous les couples `(l1,l2)` tels que `l = l1 @ l2`.

Exemple : `coupures [1;2;3]` renvoie `[ ([], [1;2;3]); ([1], [2;3]); ([1;2], [3]); ([1;2;3], []) ]`.

Corrigé

```
let rec coupures l =
    ([], l) :: match l with
                | [] -> []
                | t::q ->
                    List.map (fun (l1,l2) -> (t::l1, l2))
                      (coupures q)
```

Ici, le `List.map` sert à rajouter `t` en tête de toutes les listes `l1`.

5.3 Installer OCaml et son écosystème sur votre ordinateur

On a vu comment avoir un environnement complet avec la machine virtuelle, mais c'est peu efficace. Le mieux est d'avoir directement `OCaml` (et ça va impliquer d'avoir un compilateur `C` en fait, donc on fait d'une pierre deux coups).

5.3.1 Installer opam

Le plus simple, de très loin, est d'être sur `Linux`. Il suffit alors d'installer `opam` avec votre gestionnaire de paquets.

Sur `macOS`, il faut d'abord installer un gestionnaire de paquets et ensuite, c'est comme si on était sur `Linux`. Le meilleur est brew.

Sur `Windows`, la meilleure option, de très loin, est d'installer WSL qui va, en fait, superposer une installation de `Linux` dans `Windows`. C'est un peu comme si vous aviez un calque `Linux` qui se rajoute au-dessus. Ensuite, on procède exactement comme sur `Linux`. Pour installer `WSL`, vous pouvez suivre ces instructions.

5.3.2 Installer un switch

Une fois `opam` installé, on peut installer sa propre mini-distribution de `OCaml`.

Pour cela, dans un terminal, commencez par lancer `opam init` pour initialiser la configuration d'`opam`, puis on va choisir une version de `OCaml` 5 avec, par exemple, `opam switch create 5.2.0` (la commande `opam switch list-available` donne la liste des versions disponibles).

Répondez `y` à tout. Quand vous relancez un terminal, normalement, vous avez la version de `OCaml` choisie.

6

TRAVAUX PRATIQUES 6

Types enregistrements

DANS CE CHAPITRE

6.1 Rappel sur les enregistrements
6.2 Références et pointeurs
6.3 A link to the future

6.1 Rappel sur les enregistrements

On rappelle qu'on peut définir en `OCaml` des types proches des `struct` du `C`. On parle de **type enregistrement** (ou *record* en anglais) en référence à un parallèle avec une fiche comme celles que l'on vous demande de remplir pour les inscriptions : un certain nombre de *champs* ayant une référence précise comme *adresse* ou *numéro de téléphone* et un type de valeur attendu pour les réponses.

En `OCaml`, pour stocker le nom, l'âge et la moyenne d'informatique d'un élève, on pourrait ainsi imaginer un enregistrement à trois champs et le définir ainsi :

```
type eleve = {
    nom : string;
    age : int;
    moyenne : float
}
```

L'ordre dans lequel les champs sont énumérés n'importe pas. Pour créer une valeur du type `eleve`, il suffit de renseigner les champs, dans l'ordre qu'on veut, comme si on remplissait une fiche `{ nom = "Emile"; age = 18; moyenne = 19.9 }`.

Si on dispose d'une valeur `e` du type `eleve`, on accède aux valeurs de ses champs en utilisant `.` comme en `C` : `e.nom` pour avoir son nom.

Les valeurs des champs sont **immuables** par défaut, cela signifie qu'on ne peut pas changer la moyenne de l'élève ! Comme souvent, travailler avec des données persistantes est plus simple et ce choix par défaut est raisonnable, mais dans de nombreux cas, on aimerait que certaines valeurs puissent changer. À cette fin, on peut déclarer un champ `mutable`. On pourrait ainsi redéfinir `eleve` ainsi :

```
type eleve = {
    nom : string;
    age : int;
    mutable moyenne : float
}
```

On aurait alors à écrire `e.moyenne <- 12.4` pour changer la valeur de la moyenne.

Attention il est interdit d'écrire cela si le champ n'est pas `mutable`.

On rappelle que les références de `OCaml` sont en fait des types enregistrement `'a ref` avec un unique champ `mutable contents : 'a`. On a ainsi une correspondance parfaite entre les notations :

- `ref x` est en fait `{ contents = x }`
- l'évaluation `!x` est `x.contents`
- `x := v` est en fait `x.contents <- v`

Question 1 Écrire une fonction `plus_age : eleve list -> eleve` qui prend en entrée une liste d'élèves **non vide** et renvoie l'élève le plus âgé.

Tests

```
let e1 = { nom = "A"; age = 12; moyenne = 10.0 } in
let e2 = { nom = "A"; age = 15; moyenne = 9.0 } in
let e3 = { nom = "A"; age = 13; moyenne = 11.0 } in
let classe = [ e1; e2; e3 ] in
assert(plus_age classe = e2);
"OK";;
```

Corrigé

```
let rec plus_age l =
    match l with
    | [] -> failwith "Erreur"
    | [e] -> e
    | e::q -> let e' = plus_age q in
        if e.age > e'.age
        then e
        else e'
```

Question 2 Écrire une fonction `moyenne : eleve list -> float` qui prend en entrée une liste d'élèves **non vide** et renvoie la moyenne des moyennes des élèves.

Rappel les flottants en `OCaml` sont un peu pénibles car ils ont des opérateurs dédiés comme `+.` et `/.`. Pour convertir un `int` en `float` on peut utiliser `float_of_int : int -> float`.

Tests

```
let e1 = { nom = "A"; age = 12; moyenne = 10.0 } in
let e2 = { nom = "A"; age = 15; moyenne = 9.0 } in
let e3 = { nom = "A"; age = 13; moyenne = 11.0 } in
let classe = [ e1; e2; e3 ] in
assert(moyenne classe = 10.0);
"OK";;
```

Corrigé

```
let moyenne l =
    let n = List.length l in
    let rec aux l = match l with
        | [] -> 0.0
        | e::q -> e.moyenne +. aux q
    in aux l /. (float_of_int n)
```

Question 3 Écrire une fonction `bulletin : eleve list -> unit` qui rajoute 2 points à la moyenne de tous les élèves de la liste.

Tests

```
let e1 = { nom = "A"; age = 12; moyenne = 10.0 } in
let e2 = { nom = "A"; age = 15; moyenne = 9.0 } in
let e3 = { nom = "A"; age = 13; moyenne = 11.0 } in
let classe = [ e1; e2; e3 ] in
bulletin classe;
assert(e1.moyenne = 12.0);
assert(e2.moyenne = 11.0);
assert(e3.moyenne = 13.0);
"OK";;
```

Corrigé

```
let rec bulletin l =
    match l with
    | [] -> ()
    | e::q ->
        e.moyenne <- 2. +. e.moyenne;
        bulletin q
```

6.2 Références et pointeurs

On va se servir des références pour faire des pointeurs comme en `C`.

On définit le type

```
type 'a ptr = 'a option ref
```

Note on a ici à la fois une référence et un type option, tout l'enjeu de l'exercice est d'apprendre à bien manipuler ce type qu'on retrouvera dans beaucoup de contextes où on a besoin d'une information partielle, comme lorsqu'on veut résoudre petit à petit une grille de *Sudoku*.

On rappelle que le type `'a option` est prédéfini et qu'il correspond au type somme `type 'a option = None | Some of 'a`. La valeur `None` signifie qu'il n'y a pas d'information et `Some v` signifie qu'on a une information de valeur `v`.

Pour utiliser une valeur de type option il est ainsi nécessaire d'effectuer un filtrage :

```
match x with
| None -> (* ce qu'on fait quand il n'y a rien *)
| Some v -> (* ce qu'on fait quand il y a une valeur v *)
```

Pour plus d'informations sur le type `option`, on pourra lire ce chapitre du cours.

Ici, un pointeur sur la valeur `v` sera donc défini par `ref (Some v)`. Le pointeur invalide que l'on note `NULL` en `C` sera défini par `ref None`.

Question 4 Écrire une fonction `null_pointer : unit -> 'a ptr` qui renvoie le pointeur nul.

Tests

```
let p = null_pointer () in
assert(!p = None);
"OK";;
```

Corrigé

```
let null_pointer () =
    ref None
```

Question 5 Écrire une fonction `is_null : 'a ptr -> bool` qui renvoie un booléen indiquant si un pointeur est nul.

Tests

```
assert(is_null (ref None));
assert(not (is_null (ref (Some 3))));
"OK";;
```

Corrigé

```
let is_null p =
    !p = None
```

Question 6 Écrire une fonction `dereference : 'a ptr -> 'a` qui renvoie la valeur pointée par un pointeur.

Cette fonction devra produire une erreur avec un `failwith` si le pointeur est nul.

Tests

```
let p = ref (Some 3) in
assert (dereference p = 3);
"OK";;
```

Corrigé

```
let dereference p =
    match !p with
    | None -> failwith "Null pointer"
    | Some v -> v
```

Question 7 Écrire une fonction `set_pointer : 'a ptr -> 'a -> unit` qui change, ou définit s'il était nul, la valeur pointée par un pointeur.

Tests

```
let p = ref None in
set_pointer p 3;
assert(p = ref (Some 3));
set_pointer p 5;
assert(p = ref (Some 5));
"OK";;
```

Corrigé

```
let set_pointer p v =
    p := Some v
```

Ce n'est pas très pratique d'écrire `dereference p` ou `set_pointer p v`. On peut utiliser des raccourcis avec des opérateurs préfixes et infixes. Ainsi, on peut définir :

```
let (!!) p = dereference p
let (<==) p v = set_pointer p v
```

On peut alors directement utiliser ces symboles ainsi :

```
let p = null_pointer () in
p <== 3; (* p pointe vers 3 *)
!!p (* la valeur pointée, donc 3 *)
```

Pour le moment, l'usage semble un peu limité, on va voir une utilisation possible : initialiser des paramètres dans des boucles.

```
let recherche t x = (* renvoie un indice de x dans t s'il est présent *)
    let p = null_pointer () in
    for i = 0 to Array.length t - 1 do
        if t.(i) = x
        then p <== i
    done;
    !!p
```

Si `x` n'est pas présent dans `t`, l'appel à `!!p` déclenche l'erreur `Failure "Null pointer"`.

6.3 A link to the future

Cette partie prend un peu d'avance sur le cours sur les listes chaînées.

On va considérer maintenant le type

```
type link = {
    mutable value : int;
    next : link ptr
}
```

Les valeurs de ce type seront appelées des **maillons** et leur assemblage constitue une **chaîne**.

Un maillon de type `link` contient ainsi une valeur, qui peut changer, et un pointeur vers le maillon suivant. Quand il n'y a pas de maillon qui fait suite, on placera le pointeur nul.

En suivant les pointeurs `next`, on voit ainsi apparaître une chaîne :

```
<1,|>        |---> <3,NUL>
   |---> <2,|>
```

Cette chaîne est ainsi une autre manière de représenter une liste comme `[1;2;3]`.

Dans les questions suivantes, on utilise le mot *chaîne* quand il s'agit de voir non seulement un maillon, mais l'ensemble de la chaîne qui s'en déduit.

Question 8 Écrire une fonction `single_link : int -> link` qui crée une chaîne comportant un unique maillon contenant une valeur.

Tests

```
let l = single_link 3 in
assert(l = { value = 3; next = ref None });
"OK";;
```

Corrigé

```
let single_link x =
    {
        value = x;
        next = null_pointer ()
    }
```

Question 9 Écrire une fonction `link_connect : link -> link -> unit` telle que `link_connect m1 m2` place `m2` comme maillon suivant de `m1`.

Tests

```
let m1 = { value = 3; next = ref None } in
let m2 = { value = 1; next = ref None } in
link_connect m1 m2;
assert(!!(m1.next) = m2);
"OK";;
```

Corrigé

```
let link_connect m1 m2 =
    m1.next <== m2
```

Question 10 Écrire une fonction `is_last : link -> bool` qui renvoie un booléen indiquant si un maillon est le dernier.

Remarque : il faut utiliser `is_null`.

Tests

```
let m = { value = 3; next = ref None } in
let m' = { value = 3; next = ref (Some m) } in
assert(is_last m);
assert(not (is_last m'));
"OK";;
```

Corrigé

```
let is_last m = is_null m.next
```

Question 11 Écrire une fonction `next_link : link -> link` qui renvoie le maillon suivant après un maillon qui n'est pas le dernier.

Remarque : il faut utiliser `dereference` ou `!!`.

Tests

```
let m1 = { value = 3; next = ref None } in
let m2 = { value = 1; next = ref None } in
link_connect m1 m2;
assert(next_link m1 = m2);
"OK";;
```

Corrigé

```
let next_link m =
    !!(m.next)
```

Question 12 Écrire une fonction `link_of_list : int list -> link` qui crée une chaîne depuis une liste d'entiers **non vide**.

Tests

```
let s = link_of_list [1;2;3] in
let m3 = { value = 3; next = ref None } in
let m2 = { value = 2; next = ref (Some m3) } in
let m1 = { value = 1; next = ref (Some m2) } in
assert(s = m1);
"OK";;
```

Corrigé

```
let rec link_of_list l = match l with
    | [] -> failwith "Empty list"
    | [v] -> single_link v
    | t::q ->
        let m = single_link t in
        let s = link_of_list q in
        link_connect m s;
        m
```

Dans les fonctions suivantes, on fera les parcours de chaîne uniquement en récursif mais sans `match` en utilisant `is_last` et `next_link` à la place.

Question 13 Écrire une fonction `last_link : link -> link` qui renvoie le dernier maillon d'une chaîne en descendant, tant que c'est possible, le long des pointeurs `next`.

Tests

```
let l = link_of_list [1;2;3] in
assert(last_link l = { value = 3; next = ref None });
"OK";;
```

Corrigé

```
let rec last_link m =
    if is_last m
    then m
    else last_link (next_link m)
```

Question 14 Écrire une fonction `print_link : link -> unit` qui affiche les éléments d'une chaîne.

Tests

```
let l = link_of_list [1;2;3] in
print_link l;;
```

Corrigé

```
let rec print_link m =
    print_int m.value;
    if not (is_last m)
    then print_link (next_link m)
```

Question 15 Écrire une fonction `link_to_list : link -> int list` qui renvoie la liste des valeurs dans une chaîne.

Tests

```
let l = link_of_list [3;2;1] in
let ll = link_to_list l in
assert(ll = [3;2;1]);
"OK";;
```

Corrigé

```
let rec link_to_list m =
    if is_last m
    then [ m.value ]
    else m.value :: link_to_list (next_link m)
```

Question 16 En déduire une fonction `append : link -> link -> unit` qui colle la seconde chaîne à la fin de la première.

Tests

```
let m1 = link_of_list [1;2] in
let m2 = link_of_list [3;4] in
append m1 m2;
assert(link_to_list m1 = [1;2;3;4]);
"OK";;
```

Corrigé

```
let append m1 m2 =
    link_connect (last_link m1) m2
```

Attention : `append m m` crée une chaîne cyclique, sur laquelle `last_link` ne termine plus.

CHAPITRE 7

Type option en OCaml

DANS CE CHAPITRE

7.1 Principe
7.2 Syntaxe
7.3 Utilisation concrète

7.1 Principe

Dans de très nombreux contextes, on a besoin de pouvoir exprimer une notion de partialité en programmant. Cela apparait en général dans deux cas :

- On veut réaliser une fonction qui ne va pas pouvoir renvoyer une valeur dans tous les cas. Exemple : renvoyer la tête d'une liste chaînée ne fonctionne pas si on passe la liste vide. La fonction est en fait une fonction partielle au sens mathématique.
- On souhaite construire progressivement une donnée et il faut qu'on puisse avoir une notion de valeurs indéterminées. Exemple : on veut remplir au fur et à mesure une grille de Sudoku, à la fin, c'est une matrice de nombres, mais il faut pouvoir gérer les cases vides de manière intermédiaire.

En `C`, on résout ces questions en utilisant des valeurs comme `-1` ou le pointeur nul `NULL`. En `OCaml`, le système de types nous pousse à chercher une meilleure solution.

7.2 Syntaxe

La solution en `OCaml` est très simple, mais demande de la pratique pour l'utiliser à bon escient. Il s'agit, pour tout type `'a` de définir un type `'a option` permettant de représenter, soit une valeur de type `'a`, soit une absence de valeur.

Ce type est défini ainsi :

```
type 'a option = None | Some of 'a
```

Cela signifie qu'on a remplacé les valeurs `b` par des valeurs `Some b` et que l'absence de valeur est maintenant une vraie valeur `None`.

Aucune difficulté pour définir une valeur de type option, on se contente d'appeler un des deux constructeurs. Ainsi `Some 1` est un `int option` et `Some "test"` un `string option`. La valeur `None` est polymorphe dans le même sens que `[]`.

Pour manipuler une valeur `o` du type `'a option`, on effectue un filtrage comme pour les autres types somme :

```
match o with
| None -> (* ... *)
| Some a -> (* ici on peut accéder au contenu a *)
```

Il est également possible de revenir sur le comportement précédent avec une fonction comme

```
let unwrap o =
    match o with
    | None -> failwith "None"
    | Some a -> a
```

qui permet de déballer une valeur `'a option` en faisant l'hypothèse que ce n'est pas `None`.

Remarque 7.1 Une erreur classique avec le type option, qui est la même qu'avec le type `list`, c'est de se focaliser sur `None` en commençant par écrire :

```
if o = None
then (* .. *)
else (* et ici on est bloqué, car o est emballé *)
```

Donc, une règle : si on a une valeur `option`, on effectue un filtrage !

7.3 Utilisation concrète

7.3.1 Définir une fonction partielle

Il se trouve qu'on a déjà vu un mécanisme permettant de définir une fonction qui peut échouer : les exceptions, notamment avec `failwith`. Si on considère une fonction `f : 'a -> 'b` on a ainsi deux types de valeurs dans le type `'a` :

- Les valeurs `x` qui permettent d'obtenir une valeur `f x` du type `'b`
- Les valeurs `y` pour lesquelles `f y` produit une erreur.

Schématiquement, si on note A les valeurs du type `'a`, on a donc $A = A_s \cup A_e$ où l'union est disjointe et A_s sont les valeurs pour lesquelles `f` ne produit pas d'erreur et A_e les valeurs produisant des erreurs. Si on prend la définition usuelle de fonctions en mathématiques, A_s est le domaine de définition de `f` et on considère en fait une **application** de A_s dans B les valeurs du type `'b`.

L'idée avec le type `option`, c'est de transformer toute fonction partielle $f : A \to B$ en une application $f^\star : A \to B \cup \{\star\}$ où $\star$ est un élément spécial $\star \notin B$ qui correspond à une **valeur** erreur. Cela correspond à une *réification* de l'erreur au rang de valeur. L'application $f^\star$ est ainsi définie :

$$\begin{array}{rccc} f^\star & : & A & \to & B \cup \{\star\} \\ & & a & \mapsto & \begin{cases} f(a) & \text{si } a \in A_s \\ \star & \text{si } a \in A_e \end{cases} \end{array}$$

La valeur $\star$ est ce qu'on appelle un *puits*, c'est ici qu'on redirige toutes les entrées invalides.

Il n'y a pas de type correspondant directement à l'union $B \cup \{\star\}$ en `OCaml` et c'est pour cela qu'on utilise le type `'b option` qui nous force à *emballer* un retour.

Ainsi à chaque fois qu'on aurait renvoyé une valeur `b`, on va renvoyer plutôt une valeur `Some b`, et à la place de produire une erreur, on renvoie `None`.

Par exemple, le code suivant :

```
let tete l =
    match l with
    | t :: q -> t
    | [] -> failwith "Liste vide"
```

va s'écrire

```
let tete_opt l =
    match l with
    | t :: q -> Some t
    | [] -> None
```

7.3.2 Appeler une fonction partielle

Pour appeler une fonction partielle, il n'y a pas de difficulté, il suffit de faire l'appel de fonction et de manipuler ensuite le type option comme vu plus haut. Cela signifie qu'on effectuera en général l'appel directement dans l'expression d'un filtrage :

```
match f a with
| None -> (* ... *)
| Some b -> (* ... *)
```

7.3.3 Données partielles

Pour tout type polymorphe `'a t` on peut en déduire un type partiel `'a option t` dans lequel les valeurs peuvent ne pas être définies. Ainsi, un `int option array` permettra pour chaque case du tableau de :

- soit être définie, et ce sera alors une valeur de la forme `Some k` où `k` est un `int`,
- soit être indéfinie avec la valeur `None`.

Ainsi, une grille de Sudoku qu'on remplirait progressivement aurait le type `int option array array`. Une case valant `None` signifiant qu'elle n'est pas encore remplie. Bien entendu, une fois la grille remplie, toutes les valeurs seraient de la forme `Some k`. On peut ainsi imaginer une transformation de `int option array array` vers `int array array` consistant à passer d'une grille pouvant être partielle, mais pleinement remplie à une grille d'entiers :

```
let grille_complete go =
    let g = Array.make_matrix 9 9 0 in
    for i = 0 to 8 do
        for j = 0 to 8 do
            match go.(i).(j) with
            | None -> failwith "Grille incomplète"
            | Some k -> g.(i).(j) <- k
        done
    done;
    g
```

7.3.4 Chaîne de traitement

On pourrait se dire que le type `option` n'est qu'en fait une version un peu lourde de la production d'erreurs. Mais en fait, la force du type `option` est justement que c'est un type comme un autre. Cela permet d'effectuer de nombreux traitements en manipulant des fonctions partielles pour définir des données partielles et d'attendre uniquement au dernier moment pour déballer les valeurs. L'idée forte est de manipuler la partialité naturellement.

Un exemple peut être d'avoir une série de traitements à appliquer sur une liste, chaque traitement va possiblement faire apparaitre des `None` qu'on va traiter naturellement avec éventuellement une étape finale pour retirer les options :

Fira Sans([a1; a2; a3; a4; a5; ...])
↓
Fira Sans([None; Some b2; Some b3; None; Some b5; ...])
↓
Fira Sans([None; None; Some c3; None; Some c5; ...])
↓
Fira Sans([c3; c5; ...])

On remarque que ce style de programmation est de plus en plus privilégié par la bibliothèque standard de `OCaml` :

- de nombreuses fonctions partielles `f : 'a -> 'b` ont une variante `f_opt : 'a -> 'b option`
- il y a des fonctions manipulant directement des `_opt` comme `List.filter_map` qui effectue un `map` avec une fonction `'a -> 'b option` et filtre les `None` en sortie.
- Le nouveau module `Option` contient beaucoup de fonctions pour se simplifier la vie quand on programme avec les types `option`. **Attention**, cela correspond à une utilisation bien plus pointue de `OCaml` que dans le reste de ce cours. Notamment, cela prend tout son sens avec l'opérateur `|>` qu'on n'utilise pas ici.

CHAPITRE 8

Exceptions en OCaml

DANS CE CHAPITRE

8.1 Syntaxe des exceptions
8.2 Exceptions pour la gestion d'erreurs
8.3 Programmer avec des exceptions

Les exceptions sont un mécanisme présent dans de nombreux langages de programmation afin de permettre à la fois d'indiquer qu'une erreur s'est produite en interrompant l'exécution d'une fonction et aussi de pouvoir, pour l'appelant **rattraper** cette erreur afin de poursuivre l'exécution comme il se doit.

En `OCaml`, ce mécanisme va permettre, de plus, de pouvoir interrompre le flot de contrôle, notamment de retrouver des mécanismes comme `break`, `continue` et surtout `return`.

8.1 Syntaxe des exceptions

Les exceptions forment un type `exn` qui est extensible : il est possible de rajouter un nouveau constructeur pour ce type en écrivant :

```
(* Pour une exception sans paramètre *)
exception NomDeLException

(* Pour une exception avec paramètre de type t*)
exception NomDeLException of t
```

On peut alors **lancer** cette exception à l'aide de `raise` :

```
(* Pour une exception sans paramètre *)
raise NomDeLException

(* Pour une exception avec paramètre de valeur v *)
(* Attention aux parenthèses *)
raise (NomDeLException v)
```

Il est possible d'évaluer une expression en permettant d'évaluer une autre expression en cas d'exception :

```
try
    expression
with ExceptionsARattraper -> expression'
```

En fait, ce `with` est un filtrage, un `match`, sur les valeurs du type `exn` mais sans vérification d'exhaustivité. On peut donc effectuer plusieurs cas :

```
try
    expression
with  motif1 -> expr1
    | motif2 -> expr2
    | ...
```

En cas de `try... with` imbriqués, c'est le `try` le plus proche de l'exception qui la rattrape. Si une exception n'est pas rattrapée, elle va produire une erreur qui stoppera l'exécution d'un programme. On peut très bien lancer une exception depuis un `with .. ->`.

8.2 Exceptions pour la gestion d'erreurs

Il s'agit ici de l'utilisation la plus *logique* des exceptions.

On a déjà pu rencontrer des exceptions en `OCaml` :

```
### let t = [|1;2|];;
val t : int array = [|1; 2|]
### t.(2);;
Exception: Invalid_argument "index out of bounds".
### List.hd [];;
Exception: Failure "hd".
```

Dans les deux cas, ces exceptions sont plutôt le signe d'une erreur de programmation qu'il faut corriger qu'un comportement limite qu'il faudrait prendre en compte. Dans le cas d'une fonction comme `List.hd`, on a également vu qu'il était préférable d'utiliser un type option pour pouvoir gérer l'erreur derrière avec :

```
let hd_opt l =
    match l with
    | [] -> None
    | t :: _ -> Some t
```

Pour autant, il y a des erreurs importantes à gérer en `OCaml`, ce sont celles qui sont inévitables quand il n'est pas possible de prévoir si une opération va réussir. Citons deux cas :

- la lecture d'un fichier va produire une exception `End_of_file` une fois atteinte la fin de celui-ci ;
- des structures de données qui ne fournissent comme seul moyen efficace de savoir si elles sont vides que d'essayer d'en extraire un élément et d'échouer.

Ainsi, dans le premier cas, on pourra écrire une fonction comme :

```
let rec input_all_lines ic =
    try
        let l = input_line ic in
        l :: input_all_lines ic
    with End_of_file -> []
```

Cette fonction prend un canal d'entrée (de type `in_channel`) et renvoie toutes les lignes qu'il contient sous la forme d'une liste. Le rattrapage d'exception est crucial, c'est lui qui permet d'avoir un cas de base pour la récurrence.

Remarque 8.1 **Attention**, on peut être tenté d'écrire :

```
let rec input_all_lines ic =
    try
        input_line ic :: input_all_lines ic
    with End_of_file -> []
```

Or, il se trouve que `OCaml` évalue tous les arguments du constructeur `::` avant de construire la liste et, en pratique (l'ordre n'est pas spécifié par le langage), il le fait de la droite vers la gauche. Le code précédent

va donc effectuer des appels récursifs sans jamais lire une seule ligne, jusqu'au débordement de pile (`Stack_overflow`).

8.3 Programmer avec des exceptions

8.3.1 Retour prématuré

Si on considère le programme `C` suivant :

```c
int recherche_element(int t[],
    unsigned int nb_elts, int x)
{
    for(int i = 0; i < nb_elts; i++)
        if(t[i] == x)
            return i;
    return -1;
}
```

On peut l'écrire sous la forme suivante en `OCaml` :

```
let recherche_element t x =
    let indice = ref None in
    for i = 0 to Array.length t - 1 do
        if !indice = None && t.(i) = x
        then indice := Some i
    done;
    !indice
```

Cela correspond à ce qu'on aurait fait en `C` pour se passer du `return` dans la boucle mais cela présente plusieurs problèmes :

- on inhibe les itérations d'une boucle explicitement ;
- avec plusieurs boucles imbriquées, c'est peu lisible.

Notons que cette interruption du flot de contrôle est très naturelle dans un langage utilisant beaucoup la récursivité. Le code récursif suivant a la même structure que le code `C` :

```
let recherche_element t x =
    let rec rech_aux i =
        if i = Array.length t
        then None
        else if t.(i) = x then Some i
        else rech_aux (i+1)
    in rech_aux 0
```

En effet, on remarque deux feuilles possibles pour l'arbre d'appels récursifs : soit `None` en fin de tableau, soit `Some i` quand on a trouvé l'élément.

Traduire des programmes impératifs sous cette forme est également peu satisfaisant.

À l'aide des exceptions, on peut retrouver exactement la structure du code `C` initial ainsi :

```
exception Trouve of int

let recherche_element t x =
    try
        for i = 0 to Array.length t - 1 do
            if t.(i) = x
            then raise (Trouve i)
        done;
        None
    with Trouve i -> Some i
```

Ou même, en utilisant l'exception `Not_found` prédéfinie (pour signaler l'absence de l'élément) :

```
let recherche_element t x =
    try
        for i = 0 to Array.length t - 1 do
            if t.(i) = x
            then raise (Trouve i)
        done;
        raise Not_found
    with Trouve i -> i
```

L'expression `raise (Trouve i)` est ainsi exactement le pendant du `return i;`.

Plus généralement, avec une exception `exception Return of t` (où `t` est le type de la valeur renvoyée), au lieu d'écrire (en pseudo-code)

```
...
return v
...
```

on écrit

```
try
    ...
    raise (Return v)
    ...
    failwith "Unreachable"
with Return v -> v
```

8.3.2 `break` et `continue`

On a déjà vu que `break` permettait de sortir d'une boucle et `continue` de passer à l'itération suivante.

Retrouver ces mécanismes avec `OCaml` est plus anecdotique mais on peut le faire. On définit tout d'abord deux exceptions :

```
exception Break
exception Continue
```

Pour un `break` dans une boucle,

au lieu d'écrire (en pseudo-code)

```
for ... do
    ...
    break
    ...
done
```

on écrit

```
try
    for ... do
        ....
        raise Break (* sortie *)
        ...
    done
with Break -> ()
```

Et pour un `continue`,

au lieu d'écrire (en pseudo-code)

```
for ... do
    ...
    continue
```

```
    ...
done
```

on écrit

```
for ... do
    try
        ...
        raise Continue
        ...
    with Continue -> ()
done
```

CHAPITRE 9

Entrées-Sorties en Python

9.1 Entrée standard

Pour gérer l'entrée d'information d'un utilisateur en `Python`, le plus simple est de passer par l'entrée standard. Il s'agit d'une sorte de fichier ouvert où on lit des informations ligne par ligne.

Quand on lance le programme, soit l'utilisateur va effectivement taper des entrées ligne par ligne, soit l'entrée va provenir d'un fichier que l'on va rediriger sur l'entrée standard.

On utilise ainsi la fonction `input()` qui renvoie une ligne lue et passe à la suite.

Par exemple, dans le problème HIGHWAYS on nous donne un format de la forme :

```
2
4 2 1 4
1 2 5
3 4 5
4 4 1 4
1 2 5
2 3 5
3 4 5
4 2 6
```

où on nous dit que la première ligne est le nombre d'instances de problèmes à résoudre. On peut le stocker dans une variable `n` entière ainsi :

```
n = int(input())
```

Ensuite, on trouve n problèmes comportant une ligne : `sommets aretes x y` de quatre entiers suivis de `aretes` lignes de la forme `a b w` indiquant une arête de poids `w` entre `a` et `b`. Les sommets sont numérotés de 1 à `sommets`.

Pour lire trois entiers (x, y, z) écrits sous la forme `x y z` on va utiliser `split` :

```
x, y, z = map(int, input().split())
```

Donc ici, on pourrait faire une fonction `distance(g, x, y)` qui prend un graphe et deux sommets et renvoie la plus petite distance de `x` à `y`.

Pour lire et afficher le résultat, il suffit alors d'écrire :

```
n = int(input())

for _ in range(n):
    sommets, aretes, x, y = map(int, input().split())
    g = [ [] for _ in range(sommets + 1) ] # sommets numérotés à partir de 1
    for _ in range(aretes):
        a, b, w = map(int, input().split())
        g[a].append( (b, w) )
```

```
        g[b].append( (a, w) )

    d = distance(g, x, y)
    if d == float('inf'):
        print('NONE')
    else:
        print(d)
```

PARTIE II

Programmation en C

10 TP Travaux Pratiques d'apprentissage du C 95
11 TP Introduction au C (version notebook) 129
12 TP Tableaux, chaînes et automates de Wolfram (version notebook) 145
13 TP C avancé 155
14 TP DS3 - Examen pratique de C 179

Travaux Pratiques d'apprentissage du C

DANS CE CHAPITRE

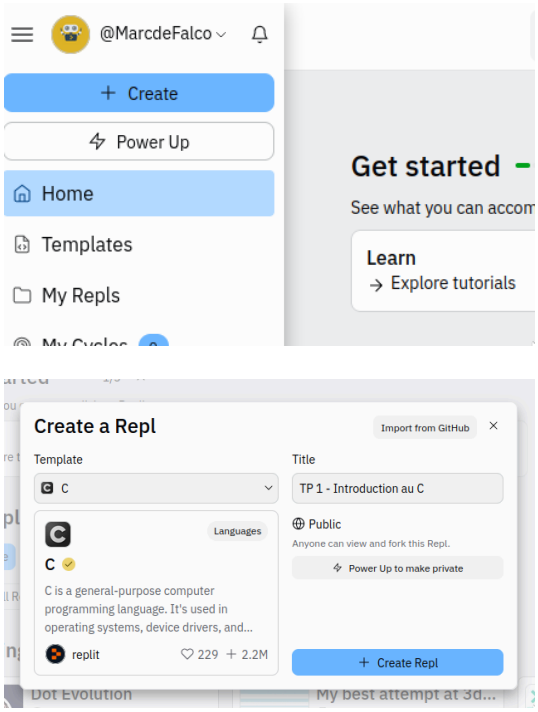
10.1 Cadre des TP
10.2 TP1 - Introduction au C
10.3 TP2 - Les tableaux
10.4 TP3 - Renforcement
10.5 TP4 - Automates de Wolfram

10.1 Cadre des TP

10.1.1 Utilisation de `Repl.it`

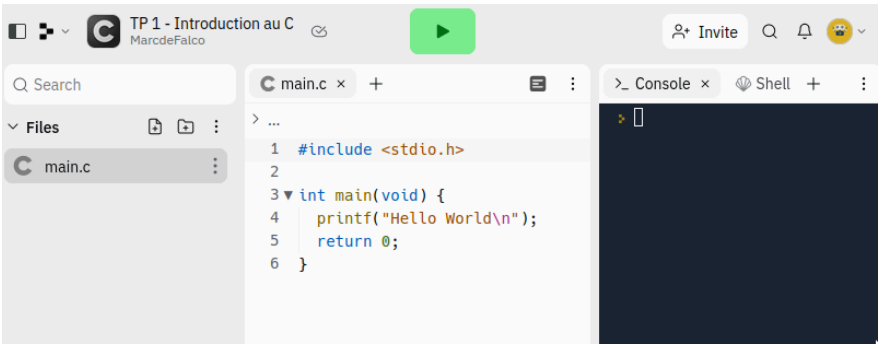
On va utiliser pour commencer le site https://replit.com.

Pour chaque TP, on commence ainsi par créer un Repl avec le bouton `+ Create` et on sélectionne le langage `C` (et pas `C++`, car cela fera vite une différence) :



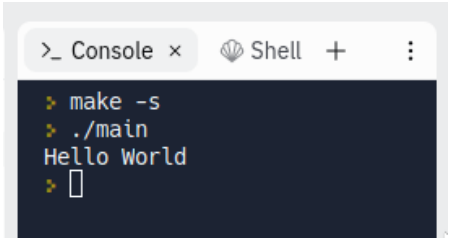
L'interface est assez simple :

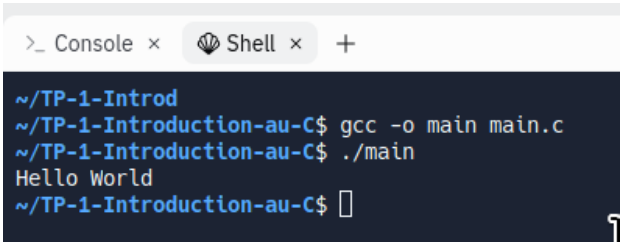
- la colonne de gauche où on peut voir les fichiers édités et en changer (pour le moment on aura juste un seul fichier) et des outils ;
- la colonne centrale où on édite un fichier ;
- la colonne de droite où on a la console dans laquelle s'effectuent les entrées et sorties textuelles et le shell qui fournit une invite de commande dans un répertoire où sont situés les fichiers.



Pour compiler et lancer son programme, on peut appuyer sur le gros bouton vert et on verra l'affichage (et les éventuelles erreurs) dans la console, on peut aussi utiliser le shell pour compiler manuellement avec le compilateur `gcc`.

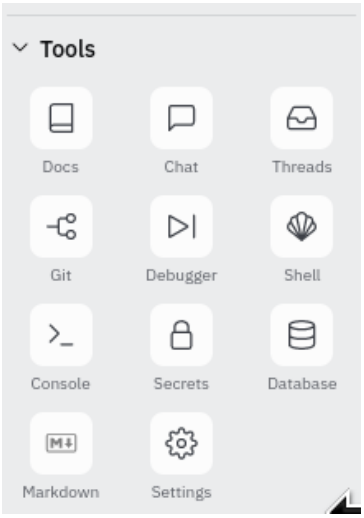
La commande est `gcc -o nomduprogramme fichier.c`. Pour le lancer, on va alors utiliser la commande `./nomduprogramme`. On reviendra sur ces notions, mais `.` signifie le répertoire courant et pour des raisons de sécurité, il faut donner le chemin d'accès à un programme qui n'est pas dans l'emplacement usuel des exécutables.

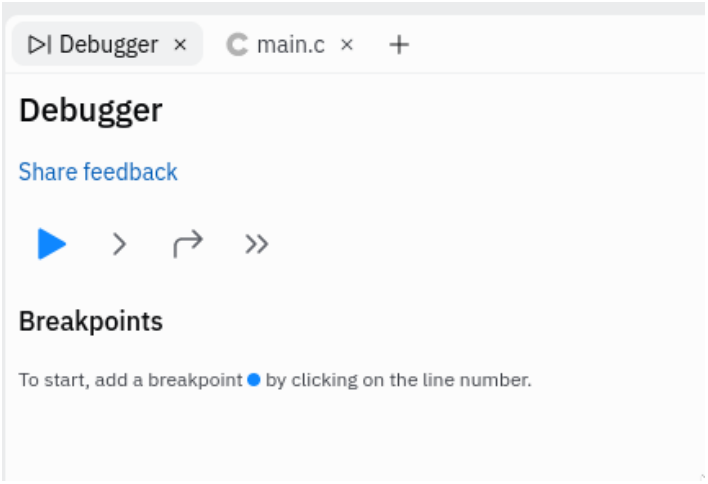




> **Remarque 10.1** Il est vivement conseillé de compiler avec les options `gcc -Wall -Wextra -g -fsanitize=address,undefined -o nomduprogramme fichier.c` : les deux premières activent les avertissements du compilateur, les suivantes ajoutent des vérifications à l'exécution qui détectent la plupart des erreurs d'accès mémoire (voir le chapitre Gestion de la mémoire).

Une autre possibilité qui sera très pratique pour vous est d'utiliser le débogueur (*debugger*) visuel intégré. On y accède en cliquant sur le bouton `Debugger` de `Tools`. Il s'ouvre par défaut dans les onglets de l'éditeur.

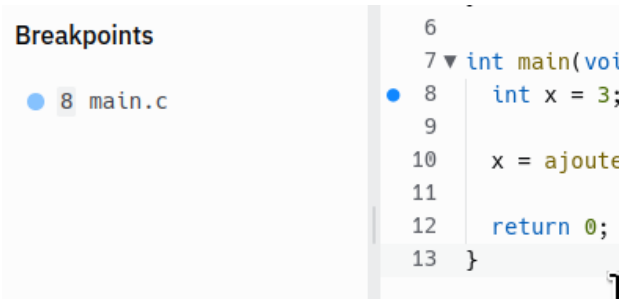




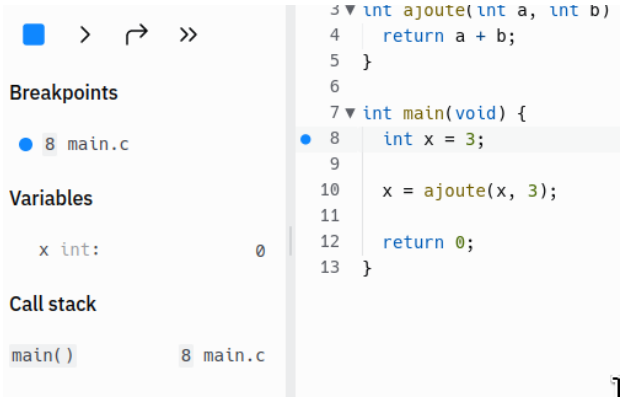
Ce n'est pas très pratique, je vous recommande de faire glisser l'onglet à côté.



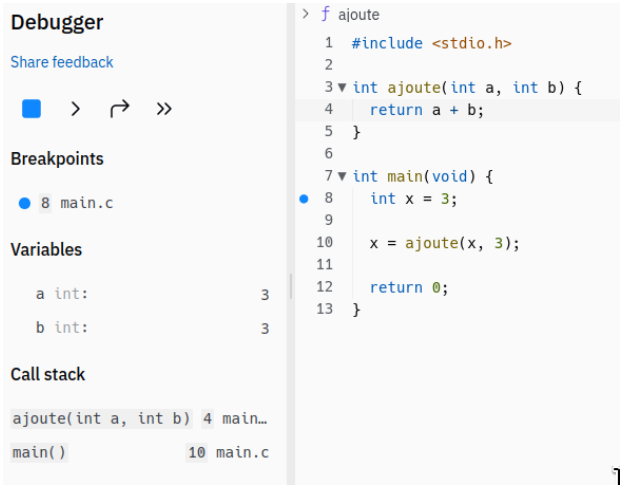
L'avantage du débogueur, c'est qu'on peut mettre des points d'arrêt (*breakpoints*) où le programme va s'arrêter et attendre quand il atteint ces points.



Pour lancer le débogage, il faut cliquer sur le bouton démarrer de la partie `Debugger`. Ça prend un peu de temps et on se retrouve sur le premier accès à un point d'arrêt avec l'état complet du programme.



On peut cliquer sur > à droite du bouton démarrer pour avancer d'un pas dans le programme (on dit *step* en anglais). On va ainsi passer dans les fonctions par les appels et on peut observer le contenu des paramètres.



En l'absence d'interprète, il va être nécessaire de faire des affichages complets pour observer des valeurs. C'est assez fastidieux. C'est là que le débogueur (ou l'observateur mémoire https://pythontutor.com/c.html) va permettre de récupérer la souplesse qu'on connaît en `Python` ou en `OCaml`.

Mais ne nous trompons pas : **c'est beaucoup plus délicat de programmer ainsi**. On va donc aller doucement.

On verra aussi que la mise en place de tests va permettre de vérifier que tout fonctionne sans intervention humaine.

10.1.2 Structure des questions

Les questions du TP vous demandent d'écrire des fonctions. Pour cela, on vous donne leur **prototype** (aussi appelé une *déclaration*), c'est-à-dire une ligne de la forme :

```
typeret nomfonction(typearg1 arg1, ..., typeargn argn);
```

Par exemple :

```
int ajoute(int x, int y);
```

On vous décrit également ce que doit faire votre fonction. **À vous de vous assurer qu'elle fonctionne.** Avec des tests, des affichages, le débogueur. Savoir vérifier qu'une fonction fait ce qu'on attend est quelque chose qu'on va apprendre progressivement.

Quand vous répondez à la question, vous allez donner une **définition** de la fonction, c'est-à-dire que la première ligne sera identique au prototype, sans le `;` et avec à la place des instructions entre `{` et `}`.

10.2 TP1 - Introduction au C

Normalement un programme en C est composé d'un fichier `.c` (le plus souvent plusieurs qui sont réunis) contenant une fonction `main` qui est le point d'entrée du programme.

Voici un exemple de contenu d'un tel fichier :

```c
// déclaration de fonctions d'impression pour pouvoir les utiliser
#include <stdio.h>

int x = 3; // une variable globale entière x pour tout le programme

// déclaration d'une fonction qui prend deux entiers en entrée et renvoie un entier
int ajoute(int a, int b)
{
    return a+b;
}

// la fonction principale

int main(void)
{
    int a = 2; // déclaration de variables locales
    int b; // non initialisée ici

    b = 3; // nécessite que b soit déclarée.

    printf("La somme de %d + %d = %d\n",
           b, x, ajoute(b, x)); // la fonction principale d'affichage, véritable couteau-suisse
                        // ici %d signifie un entier qui figure dans la liste juste après

    // On retrouve des boucles for
    a = 0;
    for (int i = 0; i < 10; i++) // mais on doit gérer notre compteur nous-même
    {
        a = a + i;
    }
    printf("La somme des entiers de 0 à 9 vaut %d\n", a);

    return 0; // pour main signifie que tout va bien
}
```

Question 1

Écrire une fonction de prototype :

```c
int multiplie(int a, int b);
```

qui renvoie le produit de `a` et de `b`.

Corrigé

```c
int multiplie(int a, int b)
{
    return a * b;
}
```

10.2.1 Les bases du C

Types et variables

Mis à part l'ajout des points-virgules à la fin des instructions, la différence principale entre `OCaml`, `Python` et `C` dans ce qu'on vient de voir se situe au niveau des instructions explicites de type et donc de la notion de déclaration de variable.

On a en `C` des types de base sur lesquels on reviendra dans les exercices : `int`, `float`, `bool`, mais une variable doit être **déclarée** comme ayant un type donné et, bien qu'on puisse en changer la valeur, on ne pourra pas changer le type des données qu'elle contient.

Pour déclarer une variable `x` ayant le type `t` on écrit `t x;`, par exemple `int b;` ici. Une fois une variable déclarée, on peut l'utiliser en lecture et en écriture. Mais il faut écrire dedans AVANT d'y lire quoi que ce soit. Pour cela, on pourra écrire `x = val;` pour que la variable `x` prenne la valeur `val`. Il est possible de déclarer et d'initialiser une variable, comme dans `int x = val;`.

ATTENTION : si on écrit juste `int x;` sans valeur, on ne peut pas supposer que `x` contienne une valeur spécifique comme `0`. On peut juste dire que `x` contient un entier quelconque qu'on se gardera bien d'utiliser dans un calcul.

Ainsi, comme `OCaml`, `C` est un langage typé statiquement : le type de chaque variable est fixé à la compilation. Mais ici les types sont explicites.

```c
int x; // x n'est pas initialisé mais c'est un entier
float y = 3.2; // y est un flottant initialisé à 3.2
bool b = true; // b est un booléen qui vaut true (nécessite stdbool.h, voir TP2)
char c = 'a'; // le type des caractères
```

Un type propre à `C` est le type de rien, qui se nomme `void`. Cela correspond au type `unit` de `OCaml`.

> **Remarque 10.2** Le typage du C est cependant beaucoup moins strict que celui d'`OCaml` : de nombreuses conversions sont implicites, par exemple `int n = 3.7;` est accepté et donne `n = 3`, et `3 / 2` vaut `1` alors que `3 / 2.0` vaut `1.5`.

Blocs et portée

En `C` on a une notion de bloc : ce sont des instructions ou des déclarations de variables entourées d'accolades, comme dans :

```c
{
    int x = 3;
    printf("x vaut %d\n", x);
}
```

Une variable déclarée dans un bloc n'existe que dans celui-ci. Dès qu'on sort du bloc on ne peut plus y accéder. Les fonctions, les instructions conditionnelles ou les boucles utilisent des blocs.

```c
// exemple avec deux blocs imbriqués où on retrouve la valeur précédente de x
// qui a juste été masquée dans le sous-bloc
{
    int x = 3;
    printf("x vaut %d\n", x);

    {
        int x = 4;
        printf("x vaut %d\n", x);
    }

    printf("x vaut %d\n", x);
}
```

qui va afficher :

```
x vaut 3
x vaut 4
x vaut 3
```

Attention, par contre, le code suivant utilise `x` dans le bloc pour le changer de valeur et comme c'est le `x` du bloc principal, ça le modifie :

```c
{
    int x = 3;
    printf("x vaut %d\n", x);

    {
        x = 4;
        printf("x vaut %d\n", x);
    }

    printf("x vaut %d\n", x);
}
```

```
x vaut 3
x vaut 4
x vaut 4
```

Fonctions

Une fonction en `C` a un type de retour et chacun des arguments est donné avec son type. S'ensuit un bloc d'instructions qui peut comporter une ou plusieurs instructions `return v;` qui vont renvoyer la valeur `v`. Pour une fonction de type de retour `void`, il n'y a pas de valeur à renvoyer : on peut omettre le `return` ou écrire `return;` sans valeur pour sortir de la fonction.

On définit une fonction ainsi :

```c
type_retour nom_fonction(type_arg1 nom_arg1, ..., type_argn nom_argn)
{
    // bloc d'instructions qui constitue le corps
    // de la fonction
}
```

On est donc très proche des fonctions `Python` mais avec cette contrainte de type.

```c
int f(int a, char c)
{
    printf("a=%d et c=%c\n", a, c);
    return a*a;
}
```

On appelle alors la fonction comme en `Python` avec `nom_fonction(val1, ..., valn)`. Ainsi, l'instruction

```c
int r = f(3, 'z');
```

affiche

```
a=3 et c=z
```

et place dans `r` la valeur `9`.

On voit que la valeur de sortie de la fonction est bien le résultat du `return`, le reste c'est juste de l'affichage. Si on écrit simplement `f(3, 'z');`, l'affichage a lieu mais la valeur renvoyée est perdue.

Les arguments sont des variables qu'on peut utiliser directement : cela ne changera pas la valeur passée en paramètre si on les redéfinit. Une fonction peut s'appeler elle-même, avec toutes les réserves vues quant à l'utilisation de la pile d'exécution.

Instructions conditionnelles

Pour effectuer une **instruction** conditionnelle, on écrit

```
if (condition)
{
    // bloc d'instructions à exécuter quand la condition est vraie
}
else
{
    // bloc d'instructions à exécuter quand la condition est fausse
}
```

Par exemple, on pourra écrire :

```
if (n % 2 == 0)
{
    printf("n est pair\n");
}
else
{
    printf("n est impair\n");
}
```

Ici, comme il n'y a qu'une instruction, on pourrait ne pas faire de bloc et écrire juste l'instruction. Mais on se gardera de faire ça pour le moment, car c'est une source classique d'erreurs.

On note qu'il n'est pas nécessaire d'avoir un `else`. Ici, comme en `Python` et contrairement à `OCaml`, on n'a pas de notion de type associé aux branches, ce sont juste des suites d'instructions.

On peut ainsi écrire

```
if (condition)
{
    // instructions si condition est vérifiée
}
```

Notamment : en `C`, ce `if` est une **instruction** et pas une expression. Si on peut écrire

```
let x = if n mod 2 = 0 then 1 else 2
```

en `OCaml`, en `C`, on devra écrire :

```
int x;

if (n % 2 == 0)
{
    x = 1;
}
else
{
    x = 2;
}
```

Au passage, remarquons qu'on utilise `==` comme en `Python`. Par contre, contrairement à `Python`, comme les déclarations ne sortent pas des blocs, on ne peut pas transposer

```
if n % 2 == 0:
    x = 1
else:
    x = 2
print("x vaut ", x)
```

car en `C` le programme suivant est erroné :

```c
if (n % 2 == 0)
{
    int x = 1;
}
else
{
    int x = 2;
}
printf("x vaut %d\n", x);
```

En effet, `x` n'est pas définie en dehors des blocs.

Exercice

Pour la question suivante, on rappelle la définition de la suite de Syracuse.

Étant donné $a \in \mathbb{N}^*$, on appelle suite de Syracuse de premier terme a la suite $(u_n)_{n\in\mathbb{N}}$ définie par $u_0 = a$ et pour $n \in \mathbb{N}$:

$$u_{n+1} = \begin{cases} \frac{u_n}{2} & \text{si } u_n \text{ est pair} \\ 3u_n + 1 & \text{sinon.} \end{cases}$$

Indications :

- pour déterminer si un nombre `a` est pair, on peut utiliser le modulo et tester `a % 2 == 0` comme en `Python` ;
- une fonction peut avoir plusieurs `return`, donc quand vous avez calculé une valeur, il suffit de la renvoyer.

Question 2

Écrire une fonction de prototype

```c
int suivant(int u);
```

qui renvoie la valeur du terme qui suit `u` dans la suite.

Corrigé

```c
int suivant(int u)
{
    if (u % 2 == 0)
    {
        return u/2;
    }
    else
    {
        return 3*u+1;
    }
}
```

10.2.2 Boucles conditionnelles

Pour effectuer une boucle conditionnelle, on écrit

```c
while(condition)
{
    // instructions à réaliser
}
```

Ce sont, comme en `Python`, les boucles les plus simples mais elles demandent de gérer plus de choses qu'une boucle inconditionnelle. Par exemple, pour calculer la somme $\sum_{k=0}^{n} k$, on doit nécessairement gérer les indices à la main :

```c
int somme_entiers(int n)
{
    int k = 0; // l'indice
    int s = 0;  // la somme courante

    while (k <= n)
    {
        s = s + k;
        k = k+1;
    }

    return s;
}
```

Le cas le plus approprié d'une telle boucle est quand on ne peut maîtriser le nombre d'itérations comme dans la fonction suivante.

Question 3

Écrire une fonction de prototype

```c
int sup_log2(int n);
```

qui renvoie, pour $n \geq 0$, le plus petit entier naturel d tel que $n < 2^d$ (attention, cette puissance n'existe pas en C).

Corrigé

```c
int sup_log2(int n)
{
    int d = 0;
    int p = 1;
    while (p <= n)
    {
        p = p * 2;
        d = d + 1;
    }

    return d;
}
```

Invariant : $p = 2^d$ et $2^{d-1} \leq n$ dès que $d \geq 1$. On a ainsi `sup_log2(0) = 0`, `sup_log2(1) = 1` et `sup_log2(15) = 4`.

10.2.3 Boucles `for`

Contrairement à `Python`, il n'y a aucun mécanisme permettant de parcourir des valeurs. Mais contrairement aux boucles `for` de `OCaml`, il ne s'agit pas simplement de boucles inconditionnelles. Le mécanisme est plus complexe.

Commençons par un exemple en reprenant la fonction `somme_entiers` :

```c
int somme_entiers(int n)
{
    int s = 0;

    for(int k = 0; k <= n; k++)
    {
        s = s + k;
    }

    return s;
}
```

Déjà, précisons que `k++` est une manière d'écrire `k = k+1`. On parle d'incrémentation de `k`.

Ensuite, on peut voir qu'il y a trois parties dans la boucle `for` :

```
for (initialisation; condition; miseajour)
{
    // instructions
}
```

L'initialisation est la plupart du temps la déclaration d'une variable de boucle. Ici, on peut utiliser le raccourci permettant de déclarer et définir la variable directement. Par opposition au fait d'écrire :

```
int k;
for (k = 0; k <= n; k++)
{
    s = s+k;
}
```

Ensuite, vient la condition d'arrêt qui est très proche de celle d'un `while`.

Enfin, la mise à jour est une instruction qui sera effectuée à la fin de chaque itération. Finalement, une boucle `for` en `C` n'est qu'un `while` déguisé :

```
initialisation;

while(condition)
{
    // instructions

    miseajour;
}
```

Pour autant, la lisibilité d'une boucle `for` est bien meilleure que celle d'un `while` car elle structure plus en mettant en valeur les **vraies** instructions du corps de boucle. On ne sera pas surpris de trouver ainsi des `for` dès que c'est possible.

Question 4

Écrire une fonction de prototype

```
int somme_carres(int n);
```

qui renvoie la somme des carrés des entiers de 0 à n : $0^2 + 1^2 + \ldots + n^2$.

Corrigé

```
int somme_carres(int n)
{
    int s = 0;
    for (int k = 0; k <= n; k++)
    {
        s = s + k*k;
    }
    return s;
}
```

Par exemple, `somme_carres(13)` vaut $\frac{13\times14\times27}{6} = 819$.

10.2.4 J'aimerais tant (re)voir Syracuse

Question 5

Écrire une fonction de prototype

```
int syracuse(int a, int n);
```

qui renvoie le n-ième terme u_n de la suite de Syracuse de premier terme a.

Attention : on utilisera `suivant` et on ne fera pas de copier-coller de code.

Corrigé

```
int syracuse(int a, int n)
{
    int u = a;
    for (int k = 1; k <= n; k++)
    {
        u = suivant(u); // on calcule u(k) ici
    }
    return u;
}
```

Question 6

Écrire une fonction de prototype

```
int temps_de_vol(int a);
```

qui renvoie le temps de vol pour a, c'est-à-dire le plus petit entier n tel que $u_n = 1$ quand $u_0 = a$.

Corrigé

```
int temps_de_vol(int a)
{
    int n = 0;
    int u = a;
    while (u != 1)
    {
        u = suivant(u);
        n = n+1;
    }
    return n;
}
```

Par exemple `temps_de_vol(1)` vaut 0 et `temps_de_vol(14)` vaut 17. La terminaison de cette boucle pour tout a est exactement la conjecture de Syracuse, qui n'est pas démontrée !

Question 7

Écrire une fonction de prototype

```
int altitude_maximale(int a);
```

qui renvoie l'altitude maximale de a, c'est-à-dire la plus grande valeur prise par la suite quand $u_0 = a$.

Corrigé

Une fois arrivée à 1, la suite boucle sur 4, 2, 1 : l'altitude maximale est donc au moins 4, et il ne faut pas oublier le premier terme $u_0 = a$ (sinon on obtiendrait 8 au lieu de 16 pour $a = 16$).

```c
int altitude_maximale(int a)
{
    int m = 4; // la suite finit par boucler sur 4, 2, 1
    if (a > m)
    {
        m = a;
    }
    int u = a;
    while (u != 1)
    {
        u = suivant(u);
        if (u > m)
        {
            m = u;
        }
    }
    return m;
}
```

Par exemple, `altitude_maximale(1)` et `altitude_maximale(2)` valent 4, et `altitude_maximale(14)` vaut 52.

10.3 TP2 - Les tableaux

10.3.1 Déclaration et initialisation

La structure de données la plus primitive en `C` est celle des tableaux. Attention, ce sont des tableaux de taille fixe comme les `'a array` de `OCaml`. Pour déclarer un tableau `t` de `n` valeurs (**où `n` est une CONSTANTE et pas une variable**) d'un type `T`, on écrit

```c
T t[n];
```

Par exemple `int a[10];` déclare `a` comme étant un tableau de 10 entiers.

> **Remarque 10.3** Depuis C99, il est possible de déclarer un tableau dont la taille est une variable (*variable length array*). On ne le fera pas : un tel tableau est placé sur la pile et une taille trop grande provoque un dépassement de pile. On verra plus tard comment allouer un tableau de taille quelconque avec `malloc`.

On peut alors accéder en lecture et en écriture à la i-ème case du tableau en écrivant `t[i]`. Ici tout se passe comme si `t` était une collection de variables accessibles par des indices. De plus, on a l'assurance que les cases se suivent linéairement en mémoire, ce qui sera utile dans la suite.

REMARQUE : `C` est un langage sans garde-fou, on peut tout à fait écrire `t[100]` pour un tableau de taille 3 et ainsi aller écrire n'importe où en mémoire et…



Explosion d'Ariane 5 au premier lancement (1996), suite à une erreur de conversion d'un flottant vers un entier 16 bits trop petit, non rattrapée, qui a conduit à l'autodestruction du lanceur.

Attention : dans presque tous les cas, les valeurs ne sont pas initialisées, et comme pour les variables, il faut attribuer des valeurs avant de travailler sur le tableau. Notamment, les cases ne sont pas initialisées à zéro.

Il est possible d'initialiser un tableau au moment de sa déclaration. Pour cela, on met les valeurs entre accolades comme ici :

```
int t[3] = { 1, 3, 5 };
```

Ce qui équivaut à l'écriture de :

```
int t[3];
t[0] = 1;
t[1] = 3;
t[2] = 5;
```

Si jamais on donne moins de `n` valeurs pour `t[n]`, on va compléter par des valeurs nulles. Cela permet d'utiliser le cas spécial `{ 0 }` pour tout initialiser à zéro. Les trois définitions suivantes sont alors équivalentes :

```
int t[3] = { 0, 0, 0 };
int t[3] = { 0, 0 };
int t[3] = { 0 };
```

> **Remarque 10.4** On rencontre aussi `int t[3] = {};` : c'est une extension de `gcc`, acceptée par la norme seulement depuis C23.

Quand on donne une succession complète de valeurs, on connaît déjà le nombre d'éléments du tableau. Il est alors assez redondant de le donner une seconde fois dans la déclaration. On dispose ainsi d'un mécanisme automatique en écrivant juste `T t[] = ...;`. On peut reprendre la déclaration précédente ainsi :

```
int t[] = { 1, 3, 5 };
```

10.3.2 Tableaux en arguments

Pour passer un tableau en paramètre à une fonction on pourra utiliser, entre autres, `int t[]` ou `int t[10]` avec une taille précise, mais **il n'y a aucune vérification de taille en C**. On prendra donc l'habitude de donner en arguments le nombre d'éléments du tableau.

Ainsi la fonction suivante calcule la somme des éléments d'un tableau.

```c
int somme_tableau(int t[], int nb_elements)
{
    int s = 0;

    for (int i = 0; i < nb_elements; i++)
    {
        s = s + t[i];
    }

    return s;
}
```

Qu'on peut tester ainsi :

```c
int t[] = { 0, 1, 2, 3 };
printf("%d\n", somme_tableau(t, 4)); // affiche 6
```

Mais on peut très bien appeler la fonction sur le même tableau avec une taille plus petite :

```c
printf("%d\n", somme_tableau(t, 3)); // affiche 3
```

Ou une taille plus grande, mais alors rappelez-vous de la fusée au-dessus…

```c
printf("%d\n", somme_tableau(t, 1000)); // comportement indéterminé
```

Remarque 10.5 En fait, le tableau n'est pas copié lors de l'appel : la fonction reçoit l'adresse de sa première case. C'est pourquoi elle ne peut pas connaître sa taille, et pourquoi elle peut modifier ses cases. On reviendra sur ce point avec les pointeurs (voir Pointeurs et tableaux).

Question 1

Écrire une fonction

```c
int recherche(int t[], int nb_elements, int x);
```

qui renvoie le plus petit indice de `x` dans le tableau `t` ; si `x` n'est pas présent, on renvoie `-1` (qui est toujours un indice invalide).

Rappel : on peut faire des `return` quand on veut.

Corrigé

```c
int recherche(int t[], int nb_elements, int x)
{
    for (int i = 0; i < nb_elements; i++)
    {
        if (t[i] == x)
        {
            return i;
        }
    }

    return -1;
}
```

Question 2

Écrire une fonction

```c
int maximum(int t[], int nb_elements);
```

qui renvoie la valeur du maximum du tableau `t`.

On considère que `nb_elements` > 0, donc il en existe toujours un.

Corrigé

```c
int maximum(int t[], int nb_elements)
{
    int m = t[0];
    for (int i = 1; i < nb_elements; i++)
    {
        if (m < t[i])
        {
            m = t[i];
        }
    }

    return m;
}
```

Question 3

En déduire une fonction `arg_max` qui renvoie le plus petit indice du maximum de `t` où `nb_elements` > 0.

Attention : on ne fera pas de boucles ici.

Corrigé

```c
int arg_max(int t[], int nb_elements)
{
    return recherche(t, nb_elements, maximum(t, nb_elements));
}
```

Cette solution parcourt deux fois le tableau, ce qui reste linéaire.

10.3.3 Les booléens

Les booléens ne sont pas définis par défaut, il faut pour cela charger un fichier d'en-tête particulier, `stdbool.h`. Pour cela, on rajoute en tête de fichier la ligne

```c
#include <stdbool.h>
```

On dispose alors d'un type `bool` ayant deux valeurs `true` et `false`.

Il s'agit en fait d'un enrobage autour des entiers `1` et `0`, mais c'est beaucoup plus sûr de passer par le type `bool`.

Question 4

Écrire une fonction de prototype

```c
bool est_trie(int t[], int nb_elements);
```

qui renvoie un booléen indiquant si un tableau d'entiers est trié dans l'ordre croissant.

Note : on pourra interrompre l'exécution pour `return false`, pas besoin de calculer explicitement ce booléen dans une variable.

Corrigé

```c
bool est_trie(int t[], int nb_elements) {
    for(int i = 0; i < nb_elements-1; i++) {
        if (t[i] > t[i+1])
            return false;
    }
    return true;
}
```

Un tableau vide ou à un élément est trié : la boucle ne fait alors aucune itération.

10.3.4 Des tableaux particuliers : les chaînes de caractères

Présentation du type

Une chaîne de caractères en `C` est juste un tableau de `char` dont on impose que la dernière case soit nulle. Dans le contexte des caractères on note `'\0'` le caractère nul, mais cela correspond à la valeur 0.

Ainsi :

```c
char chaine[] = { 'B', 'o', 'n', 'j', 'o', 'u', 'r', '\0' };
```

est la chaîne contenant `Bonjour`. Comme écrire ainsi les chaînes serait très fastidieux, on utilise la notation usuelle :

```c
char chaine[] = "Bonjour";
```

et tout se passe comme dans la définition ci-dessus : le tableau a 8 cases.

Pour afficher une chaîne, on peut utiliser `printf`. Attention à s'assurer d'avoir la ligne

```c
#include <stdio.h>
```

en tête de votre fichier.

Pour afficher la chaîne `s` puis revenir à la ligne, on écrira :

```c
printf("%s\n", s);
```

Attention, les caractères en C sont limités : un `char` occupe un seul octet et ne peut contenir que les caractères de la table ASCII (lettres non accentuées, chiffres, ponctuation…). On peut tout de même écrire des caractères accentués, de l'arabe ou du japonais dans une chaîne : ils sont alors codés (en UTF-8 sur nos machines) sur plusieurs octets, donc plusieurs `char`, ce qui fausse par exemple le calcul de la longueur de la chaîne. Il y a un certain nombre de caractères spéciaux comme `\n` pour effectuer un saut de ligne. On reverra ce codage plus tard.

Parcourir une chaîne

La valeur finale `'\0'` est ce qu'on appelle en informatique une **sentinelle** : elle permet de s'assurer que le tableau est fini et de ne pas avoir à donner sa longueur en plus. Une alternative pourrait être de mettre la longueur du tableau en première case pour un tableau d'entiers, mais c'est problématique quand on a un tableau de valeurs non entières ou qui peuvent dépasser cela.

On peut donc parcourir une chaîne avec une boucle `while` pour s'assurer qu'on est sur un caractère autre que `'\0'`.

Question 5

Écrire une fonction de prototype

```c
int longueur(char s[]);
```

qui renvoie le nombre de caractères de `s`, sans compter le `'\0'`.

Corrigé

```c
int longueur(char s[]) {
    int n = 0;
    while (s[n] != '\0') {
        n = n+1;
    }
    return n;
}
```

Cette fonction est si courante qu'elle existe déjà en `C` avec la fonction `strlen` présente dans l'en-tête

```c
#include <string.h>
```

On écrira donc souvent

```c
int n = strlen(s);
for(int i = 0; i < n; i++)
{
    // parcours de la chaîne
}
```

Attention, si on effectue le test `i < strlen(s)` avec des compilateurs pas malins, l'appel à `strlen` est effectué à chaque itération et c'est très coûteux : on obtient une boucle de complexité quadratique en la longueur de la chaîne.

Premières fonctions sur les chaînes

Un caractère est un petit entier sur un octet, entre `-128` et `127` (ou entre `0` et `255` selon les compilateurs). Il n'y a pas de conversion à faire : en `C`, la valeur `'a'` est un entier valant 97, son code ASCII.

Les lettres de l'alphabet se suivent dans ce codage, ainsi une lettre minuscule est entre `'a'` et `'z'`.

Question 6

Écrire une fonction de prototype

```c
bool est_minuscule(char s[]);
```

qui renvoie un booléen indiquant si tous les caractères de `s` sont des lettres minuscules.

On pourra simplement tester la fonction en appelant `est_minuscule("Toto")`, donc en donnant une chaîne constante directement, sans définir de variable.

Corrigé

```c
bool est_minuscule(char s[]) {
    int n = strlen(s);
    for(int i = 0; i < n; i++) {
        if (s[i] < 'a' || s[i] > 'z')
            return false;
    }

    return true;
}
```

Contrairement à `OCaml` ou `Python`, on peut modifier les caractères d'une chaîne.

Ainsi :

```
s[1] = 'c';
```

remplace le second caractère de `s` par `'c'`.

Question 7

Écrire une fonction de prototype

```
void capitalise(char s[]);
```

qui remplace toutes les lettres minuscules par la majuscule correspondante.

Attention, pour tester, on ne peut pas écrire

```
capitalise("Toto");
```

car `"Toto"` est en lecture seule. Il faut écrire

```
char s[] = "Toto";
capitalise(s);
```

Corrigé

```
void capitalise(char s[]) {
    int n = strlen(s);
    for(int i = 0; i < n; i++) {
        if (s[i] >= 'a' && s[i] <= 'z') {
            s[i] = s[i] + 'A' - 'a';
        }
    }
}
```

Question 8

Écrire une fonction de prototype

```
bool palindrome(char s[]);
```

qui vérifie si une chaîne est un palindrome, c'est-à-dire si on peut la lire de la même manière de gauche à droite et de droite à gauche.

Corrigé

On compare les caractères deux à deux en partant des deux extrémités, sans oublier de faire avancer les deux indices.

```
bool palindrome(char s[]) {
    int i = 0;
    int j = strlen(s) - 1;
    while (i < j) {
        if (s[i] != s[j]) {
            return false;
        }
        i = i + 1;
        j = j - 1;
    }
    return true;
}
```

La chaîne vide (`j = -1`) et les chaînes d'un caractère sont des palindromes.

10.3.5 La fonction `printf`

On va maintenant revenir sur la fonction `printf` vue précédemment. C'est une fonction spéciale qui peut prendre un nombre variable d'arguments mais qui en aura toujours au moins un : `printf(format, ...)` où `format` est une chaîne de caractères. Cette chaîne peut comporter des symboles spéciaux commençant par `%` et affichant, dans l'ordre de gauche à droite, les valeurs passées en argument après `format`.

On a par exemple `%d` pour afficher un entier signé, `%s` pour une chaîne de caractères ou `%c` pour un caractère. Exemple :

```
char s[] = "Test";
int i = 2;
printf("Le %deme caractère de la chaine %s est %c\n",
     i, s, s[i]); // oui on commence à compter à 0
```

Le nombre renvoyé par `printf` est le nombre de caractères (plus précisément d'octets) affichés après ces substitutions.

On pourra utiliser la fonction suivante pour afficher un tableau :

```
void print_tableau(int t[], int nb_elements)
{
    for(int i = 0; i < nb_elements; i++)
    {
        char c = ',';
        if (i == nb_elements - 1)
            c = '\n'; // fin de ligne pour le dernier élément.
        printf("%d%c", t[i], c);
    }
}
```

qu'on appelle ainsi :

```
int t[] = {1, 2, 3};
print_tableau(t, 3); // affiche 1,2,3
```

Question 9

Écrire une fonction de prototype

```
void print_matrix(int t[], int n, int p);
```

qui prend un tableau de $n \times p$ entiers représentant une matrice à n lignes et p colonnes comme la suite de ses coefficients ligne après ligne, et affiche cette matrice sous un format lisible, avec une ligne d'affichage par ligne de la matrice.

Par exemple, la matrice

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

sera représentée par le tableau `{ 1, 2, 3, 4, 5, 6 }`. On l'affichera ainsi :

```
1 2 3
4 5 6
```

✍ **Corrigé**

Le coefficient de la ligne i et de la colonne j se trouve à l'indice $i \times p + j$ du tableau.

```c
void print_matrix(int t[], int n, int p)
{
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < p; j++)
        {
            printf("%d", t[i * p + j]);
            if (j < p - 1)
            {
                printf(" ");
            }
        }
        printf("\n");
    }
}
```

10.4 TP3 - Renforcement

Vous pouvez directement récupérer le fichier TP_C_3.c pour le compléter : il contient les squelettes des fonctions (à la place des `// FIXME`) et des tests automatiques, lancés par la fonction `main` qui affiche `VALIDE` ou `INVALIDE` pour chaque question.

10.4.1 Manipulations de tableaux

Question 1

Écrire une fonction `swap` qui prend en entrée, dans cet ordre, un tableau d'entiers, deux indices `i` et `j` et permute les valeurs du tableau situées à ces indices.

```c
void swap(int tab[], int i, int j);
```

Corrigé

```c
void swap(int tab[], int i, int j)
{
    int temp = tab[i];
    tab[i] = tab[j];
    tab[j] = temp;
}
```

Question 2

Écrire une fonction `renverse` qui prend en entrée, dans cet ordre, un tableau d'entiers puis un entier indiquant le nombre d'éléments de ce tableau, et permute ses éléments de manière à renverser le tableau : `1,2,3` devient `3,2,1`.

```c
void renverse(int tab[], int nb);
```

Corrigé

```c
void renverse(int tab[], int nb)
{
    for (int i = 0; i < nb/2; i++)
    {
        swap(tab, i, nb-1-i);
    }
}
```

Question 3

Prouver la correction de `renverse`.

Indication : on pourra noter $a_0, \ldots, a_{n-1}$ les valeurs initiales du tableau et $t_0, \ldots, t_{n-1}$ les valeurs à un instant donné de l'exécution, et exprimer l'invariant en liant ces valeurs et l'indice de boucle.

Corrigé

Tout d'abord, on remarque qu'il n'est pas nécessaire de prouver la terminaison vu qu'il s'agit d'une boucle `for` effectuant exactement $\lfloor n/2 \rfloor$ itérations.

On va prouver l'invariant suivant, vérifié au début de l'itération d'indice i :

$$I(i) : (\forall k \in [\![0, i-1]\!], t_k = a_{n-1-k} \wedge t_{n-1-k} = a_k) \wedge (\forall k \in [\![i, n-1-i]\!], t_k = a_k).$$

- Initialisation : on a directement $I(0)$ car $[\![0, -1]\!] = \emptyset$ et aucune case n'a été modifiée.
- Hérédité : si on a $I(i)$ en début d'itération, avec $i < \lfloor n/2 \rfloor$, alors $i < n-1-i$ et les indices i et $n-1-i$ sont dans $[\![i, n-1-i]\!]$. On va donc réaliser l'échange entre $t_i = a_i$ et $t_{n-1-i} = a_{n-1-i}$ (par l'invariant). On va ainsi avoir ensuite $t_i = a_{n-1-i}$ et $t_{n-1-i} = a_i$, ce qui permet de compléter la première propriété de l'invariant pour $k = i$. Comme on n'a pas touché aux autres éléments, la seconde propriété reste vérifiée sur $[\![i+1, n-2-i]\!]$. On a bien $I(i+1)$ en sortie.

Conclusion : on a $I(\lfloor n/2 \rfloor)$ vérifié en sortie de boucle, ainsi $\forall k \in [\![0, \lfloor n/2 \rfloor - 1]\!], t_k = a_{n-1-k} \wedge t_{n-1-k} = a_k$. Quand $n = 2p$ est pair, les indices k et $n-1-k$ pour $k \in [\![0, p-1]\!]$ couvrent tout le tableau. Il n'y a qu'un élément potentiellement non couvert, quand $n = 2p+1$ est impair : c'est l'élément central d'indice p. Or, la seconde partie de l'invariant devient alors $\forall k \in [\![p, 2p+1-1-p]\!] = \{p\}, t_k = a_k$. Cet élément n'a donc pas bougé, et $t_p = a_p = a_{n-1-p}$. On a bien renversé le tableau dans tous les cas.

Question 4

Écrire une fonction qui, étant donné un tableau d'entiers et sa taille, effectue un traitement en place sur le tableau en sorte que le tableau commence par les nombres pairs et termine par les impairs, en respectant l'ordre dans lequel ils étaient dans le tableau initialement.

```
void pairs_devant(int tab[], int nb);
```

Exemples :

- `1, 2, 3, 4` devient `2, 4, 1, 3` ;
- `1, 3` reste `1, 3` ;
- `5, 4, 1, 2, 3` devient `4, 2, 5, 1, 3`.

On pourra commencer par écrire deux fonctions : `cherche_pair` qui cherche l'indice du premier nombre pair à partir d'un indice `i` (et `-1` s'il n'y en a pas), et `decale` qui prend deux indices `i` et `j` et effectue les affectations dans le tableau ainsi : `j-1` → `j`, `j-2` → `j-1`, ..., `i` → `i+1`, pour pouvoir placer un élément en `i`.

Corrigé

Ici, les deux fonctions auxiliaires ont été écrites directement dans la boucle : à l'étape `i`, on cherche le premier pair d'indice `r >= i`, on décale d'un cran vers la droite les cases d'indices `i` à `r-1` (qui sont toutes impaires) et on place le pair trouvé en `i`.

```
void pairs_devant(int tab[], int nb)
{
    for(int i = 0; i < nb; i++)
    {
        int r = -1;
```

```
        for (int j = i; j < nb; j++)
        {
            if(tab[j] % 2 == 0)
            {
                r = j;
                break;
            }
        }
        if (r == -1) break;
        int p = tab[r];
        for (int k = r-1; k >= i; k--)
        {
            tab[k+1] = tab[k];
        }
        tab[i] = p;
    }
}
```

Dans le pire cas, cet algorithme est en $O(n^2)$.

10.4.2 Le chiffre de Playfair

Le chiffre de Playfair est une technique manuelle de chiffrement symétrique et a été historiquement la première technique de chiffrement par substitution de digrammes.

Ce chiffre utilise une table de 5 par 5 contenant chaque lettre de l'alphabet latin exactement une fois, hormis la lettre Q. La table constitue la clé de chiffrement.

Par exemple, on peut considérer la table suivante :

P	L	A	Y	F
I	R	E	X	M
B	C	D	G	H
J	K	N	O	S
T	U	V	W	Z

Pour chiffrer un message, on retire tous les espaces et on coupe le message en digrammes (groupes de deux lettres) de telle sorte que, par exemple, `Hello World` devienne `HE LL OW OR LD`. Ensuite on applique les règles suivantes pour chaque digramme afin de produire le message chiffré.

- Si les deux lettres sont les mêmes (ou qu'il n'y a plus qu'une lettre comme à la fin d'un message de longueur impaire), on remplace la seconde lettre par un `X` et on chiffre le digramme obtenu.
- Si les lettres sont sur la même rangée de la table, on les chiffre avec les lettres situées immédiatement à leur droite (si on sort de la rangée ainsi, on retourne à la première lettre de celle-ci). Avec la table au-dessus, `CH` devient `DB`.
- Si les lettres sont sur la même colonne, on les chiffre par les lettres situées immédiatement en dessous dans la table, en revenant aussi à la première en cas de dépassement. Ainsi `VA` devient `AE`.
- Si les lettres ne sont pas sur la même rangée ou la même colonne, elles sont les sommets d'un rectangle dans la table, on les chiffre alors avec les autres sommets du rectangle. Comme l'ordre importe, on choisit de chiffrer une lettre par le sommet qui est sur la même rangée. Ainsi `KM` devient `SR`.

Remarque 10.6 Dans le vrai chiffre de Playfair, la première règle *insère* un `X` entre les deux lettres identiques, la seconde lettre étant reportée dans le digramme suivant : `TREES` se découpe en `TR EX ES`. On a simplifié ici en remplaçant la seconde lettre, ce qui garantit que le message chiffré a la même longueur que le message (arrondie au nombre pair supérieur), au prix de la perte d'une lettre : le déchiffrement donnera `TREXES`… à moins de ne jamais avoir deux lettres identiques dans un digramme.

La table précédente pourra être stockée sous la forme de la chaîne `"PLAYFIREXMBCDGHJKNOSTUVWZ"` : la lettre de la rangée `row` et de la colonne `col` est à l'indice `5 * row + col`.

On ne considère ici que des messages en majuscules, sans espace et sans la lettre Q. De plus, on va supposer qu'on n'a jamais deux `X` consécutifs ni un `X` en dernière position (avec le vrai chiffre, cela ferait boucler la première règle).

Question 5

Écrire deux fonctions qui prennent un caractère `c` présent dans la clé et renvoient respectivement son numéro de rangée et son numéro de colonne dans la table.

```
int car_vers_row(char cle[], char c);
int car_vers_col(char cle[], char c);
```

Avec la table précédente, `K` est à la rangée 3 et à la colonne 1.

Corrigé

```
int car_vers_row(char cle[], char c)
{
    int i;
    for (i = 0; i < 25; i++)
    {
        if (cle[i] == c)
            break;
    }
    return i / 5;
}

int car_vers_col(char cle[], char c)
{
    int i;
    for (i = 0; i < 25; i++)
    {
        if (cle[i] == c)
            break;
    }
    return i % 5;
}
```

Question 6

Écrire une fonction qui renvoie le caractère situé dans la table `cle` sur la rangée `row` et la colonne `col`.

```
char position_vers_car(char cle[], int row, int col);
```

Corrigé

```
char position_vers_car(char cle[], int row, int col)
{
    return cle[5 * row + col];
}
```

Question 7

Écrire une fonction qui réalise le chiffrement de `s` avec la clé `cle` sous forme d'une chaîne comme ci-dessus.

```
void chiffre(char cle[], char s[], char sortie[]);
```

La chaîne `sortie` contient assez de place pour stocker le message chiffré. Vous devrez bien rajouter un `'\0'` à la fin.

Avec la table précédente, `HIDETHEGOLDINTHETREEESTUMP` devient `BMNDZBXDKYBEJVDMUIXMMNUVIF`.

Corrigé

```c
void chiffre(char cle[], char s[], char sortie[])
{
    int n = strlen(s);
    for (int i = 0; i < n; i += 2)
    {
        char c1 = s[i];
        char c2;
        if (i+1 < n) c2 = s[i+1];
        else c2 = 'X';
        if (c1 == c2) c2 = 'X';

        int x1 = car_vers_row(cle, c1);
        int y1 = car_vers_col(cle, c1);
        int x2 = car_vers_row(cle, c2);
        int y2 = car_vers_col(cle, c2);

        if (x1 == x2)
        {
            c1 = position_vers_car(cle, x1, (y1+1)%5);
            c2 = position_vers_car(cle, x2, (y2+1)%5);
        }
        else if (y1 == y2)
        {
            c1 = position_vers_car(cle, (x1+1)%5, y1);
            c2 = position_vers_car(cle, (x2+1)%5, y2);
        }
        else
        {
            c1 = position_vers_car(cle, x1, y2);
            c2 = position_vers_car(cle, x2, y1);
        }
        sortie[i] = c1;
        sortie[i+1] = c2;
    }
    if (n%2 == 0) sortie[n] = '\0';
    else sortie[n+1] = '\0';
}
```

La chaîne `sortie` doit donc pouvoir contenir $n+2$ caractères quand n est impair, et $n+1$ sinon.

10.5 TP4 - Automates de Wolfram

Vous pouvez directement récupérer le fichier TP_C_4.c pour le compléter. Il contient des tests automatiques de vos fonctions.

10.5.1 Conversions binaires

On considère l'écriture binaire $n = \sum_{k=0}^{+\infty} b_k 2^k$ d'un entier naturel n, où les $b_k \in \{0, 1\}$ sont presque tous nuls. En pratique, on ne considère que des nombres dont la taille est majorée, donc de la forme $\sum_{k=0}^{N-1} b_k 2^k$. N est le nombre de bits sur lequel on a écrit l'entier n. Le bit b_{N-1} peut être nul, c'est-à-dire qu'on n'impose pas que $b_{N-1} \neq 0$; par contre, on a forcément $n < 2^N$.

Les différents types de valeurs entières en `C` sont de la forme `t` ou `unsigned t` où :

- `unsigned` signifie qu'on ne peut stocker que des valeurs positives ; si on l'omet on peut avoir des valeurs négatives, sachant qu'il faut un bit pour coder l'information du signe : un même type peut prendre des valeurs deux fois plus grandes en `unsigned`, de 0 à $2^N - 1$;

- `t` peut valoir (types principaux) **ATTENTION : c'est dépendant du compilateur, on donne ici les valeurs usuelles pour un compilateur moderne et un processeur 64 bits !** :
  - `char` : pour des valeurs sur $N = 8$ bits, donc 1 octet, donc de -128 à 127 ;
  - `short` : pour des valeurs sur $N = 16$ bits, 2 octets, donc de -2^{15} à $2^{15} - 1$;
  - `int` : pour des valeurs sur $N = 32$ bits, donc 4 octets, donc de -2^{31} à $2^{31} - 1$;
  - `long` : pour des valeurs sur $N = 64$ bits, donc 8 octets, donc de -2^{63} à $2^{63} - 1$ (mais seulement 4 octets sous Windows).

Dans le doute, on peut connaître la taille en nombre d'octets en utilisant `sizeof` :

```
printf("%zu\n", sizeof(long));
```

va afficher `8`.

Il est aussi possible d'utiliser des types explicites comme `int32_t` ou `int64_t`, définis dans `stdint.h`, qui ont une taille indépendante de l'architecture de compilation.

Question 1

Écrire une fonction de prototype

```
void vers_binaire(unsigned int x, unsigned char b[]);
```

où on considère ici que `b` est un tableau de 32 `unsigned char`. Cette fonction renvoie dans `b` la représentation en binaire de `x` de sorte que `b[i]` contienne b_i de l'expression au-dessus (comme toujours, voir les tests en cas de doute).

Corrigé

```
void vers_binaire(unsigned int x, unsigned char b[])
{
    for (int i = 0; i < 32; i++)
    {
        b[i] = x % 2;
        x = x / 2;
    }
}
```

Question 2

Écrire une fonction de prototype

```
unsigned int depuis_binaire(unsigned char b[]);
```

qui renvoie l'entier représenté par le tableau de 32 bits `b`. On fera en sorte que `depuis_binaire` et `vers_binaire` soient réciproques l'une de l'autre.

Corrigé

```
unsigned int depuis_binaire(unsigned char b[])
{
    unsigned int x = 0;
    unsigned int p = 1;
    for (int i = 0; i < 32; i++)
    {
        if (b[i] == 1)
        {
            x = x + p;
        }
```

```
        p = 2*p;
    }
    return x;
}
```

Au dernier tour, `p = 2*p` dépasse $2^{32} - 1$: pour un type `unsigned`, le calcul est fait modulo 2^{32} (ici `p` revient à 0), ce qui est sans conséquence. Avec un `int`, ce dépassement serait un comportement indéterminé.

10.5.2 Présentation et premier automate de Wolfram

On va s'intéresser ici à des automates cellulaires unidimensionnels. On considère ainsi une ligne infinie de cellules. Chaque cellule a exactement deux cellules voisines, une à sa droite et une à sa gauche. Les cellules contiennent un état. Pour simplifier ici, on ne va considérer que deux états 1 et 0, noir ou blanc, ou encore `'#'` et `' '` dans la suite.

Afin de représenter informatiquement de telles lignes infinies, on va considérer un nombre fini de cellules en supposant que toutes les autres sont dans l'état 0.



Figure 10.1 — Une ligne finie de cellules (bleu = 1, blanc = 0), toutes les autres étant à 0.

Un automate de Wolfram est la donnée d'un ensemble de règles de la forme



Figure 10.2 — Les huit règles $(g, c, d) \to c'$: en haut le voisinage, en bas le nouvel état (bleu = 1).

ce qui peut se lire sous la forme $(g, c, d) \to c'$: connaissant l'état g de la cellule de gauche, c de la cellule courante et d de la cellule de droite, le nouvel état de la cellule courante est c'.

Exemple : dans le cas du jeu de règles au-dessus, une cellule à 1 dont la voisine de gauche est à 1 et la voisine de droite à 0 va passer à 0.

Question 3

Écrire une fonction de prototype

```
int regle_vois(int g, int c, int d);
```

qui prend un triplet (g, c, d) sous la forme de valeurs 0 ou 1 et renvoie c' selon la règle au-dessus en image, selon la correspondance case pleine (bleue) = 1, case vide = 0.

Indication : oui, il faut faire un paquet de `if` sans aucune finesse.

Corrigé

```
int regle_vois(int g, int c, int d)
{
    if (g == 1 && c == 1 && d == 1)
        return 0;
    if (g == 1 && c == 1 && d == 0)
        return 0;
    if (g == 1 && c == 0 && d == 1)
        return 0;
    if (g == 1 && c == 0 && d == 0)
        return 1;
    if (g == 0 && c == 1 && d == 1)
```

```
        return 1;
    if (g == 0 && c == 1 && d == 0)
        return 1;
    if (g == 0 && c == 0 && d == 1)
        return 1;
    return 0;
}
```

On voit dans l'exemple précédent qu'il y a 8 cas de figure ; on peut les numéroter de 0 à 7. En fait, on peut considérer l'entier $4g + 2c + d$ pour associer à un tel triplet ce numéro. Par exemple, la configuration $(1, 1, 0)$ correspond à $4 + 2 = 6$.

On a ici rajouté ces numéros sur la règle précédente :

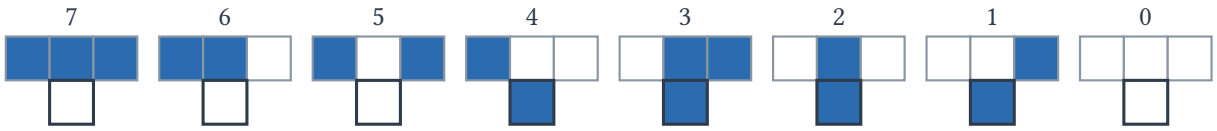


Figure 10.3 — Les mêmes règles, numérotées par $4g + 2c + d$.

Question 4

Écrire une fonction de prototype

```
int regle(int config);
```

qui se comporte comme la fonction précédente mais prend le triplet sous la forme d'un numéro comme décrit au-dessus et renvoie la nouvelle valeur.

On ne fera PAS appel à la fonction `regle_vois` précédente.

Corrigé

```
int regle(int config)
{
    if (config == 0 || config == 5 || config == 6 || config == 7)
        return 0;
    return 1;
}
```

On va considérer que la configuration est donnée sous la forme d'une chaîne, par exemple `"__##__#"` (on a écrit _ pour les espaces ici pour qu'ils soient visibles), où les `'#'` correspondent à noir, donc à 1, et les `' '` à blanc, donc à 0. On suppose qu'à gauche et à droite de cette configuration, il n'y a que des `' '`.

Si on applique la règle précédente sur cette configuration, on va obtenir `"_##_###"`. On remarque que la cellule juste à droite de la chaîne devrait devenir noire en appliquant la règle (sa configuration est $(1, 0, 0)$), mais pour simplifier, on ignore cette cellule, ce qui va, dans la suite, revenir à la maintenir blanche.

Question 5

Écrire une fonction de prototype

```
void suivant(char in[], char out[]);
```

qui prend une configuration `in` de cellules SOUS LA FORME D'UNE CHAÎNE (donc terminée par `'\0'` et dont on peut obtenir la taille avec `strlen`), applique la règle précédente sur chacune des cellules et stocke la nouvelle configuration dans la chaîne `out` DE MÊME LONGUEUR.

ATTENTION : on utilise à la place de 0 le caractère `' '` et à la place de 1 le caractère `'#'`.

ATTENTION : en dehors du tableau, on considère que les cellules sont à `' '`.

Corrigé

```
void suivant(char in[], char out[])
{
    int n = strlen(in);
    for(int i = 0; i < n; i++)
    {
        int config = 0;
        if (i > 0 && in[i-1] == '#')
        {
            config += 4;
        }
        if (in[i] == '#')
        {
            config += 2;
        }
        if (i < n-1 && in[i+1] == '#')
        {
            config += 1;
        }
        if (regle(config) == 1)
        {
            out[i] = '#';
        }
        else
        {
            out[i] = ' ';
        }
    }
}
```

Il est indispensable d'écrire dans une autre chaîne que `in` : sinon, la cellule `i+1` serait calculée à partir de la **nouvelle** valeur de la cellule `i`.

Question 6

Écrire une fonction de prototype

```
void copie_chaine(char tgt[], char src[]);
```

qui prend en entrée deux chaînes `src` et `tgt` de même longueur et copie la chaîne `src` dans la chaîne `tgt`.

Corrigé

```
void copie_chaine(char tgt[], char src[])
{
    int n = strlen(src);
    for (int i = 0; i < n; i++)
    {
        tgt[i] = src[i];
    }
}
```

Comme les deux chaînes ont la même longueur, le `'\0'` de `tgt` est déjà à la bonne place. La fonction `strcpy` de `string.h` fait la même chose en recopiant aussi le `'\0'`.

Question 7

Écrire une fonction de prototype

```
void itere_regle(char depart[], char temp[], char sortie[], int iterations);
```

qui applique `iterations` fois la fonction `suivant` en partant de la configuration `depart` et place la configuration finale dans `sortie`.

Comme on ne sait pas encore allouer une telle chaîne dynamiquement, on vous donne aussi une chaîne `temp` comme espace de travail.

ATTENTION : les chaînes `depart`, `temp` et `sortie` sont de même longueur.

ATTENTION : il ne faut pas modifier la chaîne `depart`.

Corrigé

```
void itere_regle(char depart[], char temp[], char sortie[], int iterations)
{
    copie_chaine(sortie, depart);
    copie_chaine(temp, depart);
    for (int i = 0; i < iterations; i++)
    {
        suivant(temp, sortie);
        copie_chaine(temp, sortie);
    }
}
```

La suite d'instructions suivante devrait vous permettre de **voir** les 30 premières itérations.

```
char init[81];
for(int i = 0; i < 80; i++)
{
    init[i] = ' ';
}
init[80] = '\0';
init[40] = '#';

char temp[81], sortie[81];
strcpy(temp, init);
strcpy(sortie, init);

for(int i = 0; i < 30; i++)
{
    itere_regle(init, temp, sortie, i);
    printf("%s\n", sortie);
}
```

Les premières lignes affichées sont :

```
                                        #
                                       ###
                                      ##  #
                                     ## ####
                                    ##  #   #
                                   ## #### ###
                                  ##  #    #  #
                                 ## ####  ######
```

Les 256 règles

Il se trouve que la règle qu'on vient d'implémenter s'appelle la règle 30. En effet, si on décore la règle précédente :

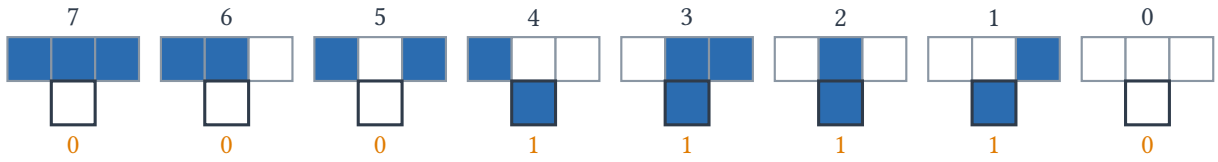


Figure 10.4 — Les sorties lues de 7 à 0 forment le nombre binaire 00011110 = 30.

on peut lire le nombre binaire `00011110` qui correspond à $0 \times 128 + 0 \times 64 + 0 \times 32 + 1 \times 16 + 1 \times 8 + 1 \times 4 + 1 \times 2 + 0 \times 1 = 16 + 8 + 4 + 2 = 30$.

Autrement dit, la valeur associée à la configuration numéro k est le bit b_k de l'écriture binaire du numéro de la règle.

On va maintenant définir une fonction permettant d'obtenir la règle associée à un nombre entre 0 et 255. Par exemple, comme 111 s'écrit 01101111 en binaire, cela signifie qu'on associe la valeur 0 aux configurations de numéro 7 et 4 (i.e. (g, c, d) valant $(1, 1, 1)$ et $(1, 0, 0)$) et la valeur 1 à toutes les autres.

Question 8

Écrire une fonction de prototype

```
int regle_gen(int gen, int config);
```

qui renvoie la valeur obtenue en appliquant la règle `gen` sur la configuration de numéro `config`.

On pourra utiliser `vers_binaire`, même si, sur 32 bits, beaucoup de valeurs seront à 0.

Corrigé

```
int regle_gen(int gen, int config)
{
    unsigned char r[32];
    vers_binaire(gen, r);
    return r[config];
}
```

On peut aussi éviter le tableau en écrivant directement `return (gen >> config) % 2;` où `>>` est le décalage à droite des bits.

Question 9

Écrire une fonction de prototype

```
void suivant_gen(int gen, char in[], char out[]);
```

qui prend une configuration `in` de cellules SOUS LA FORME D'UNE CHAÎNE (donc terminée par `'\0'` et dont on peut obtenir la taille avec `strlen`), applique la règle `gen` sur chacune et stocke la nouvelle configuration dans la chaîne `out` DE MÊME LONGUEUR.

ATTENTION : on utilise à la place de 0 le caractère `' '` et à la place de 1 le caractère `'#'`.

ATTENTION : en dehors du tableau, on considère que les cellules sont à `' '`.

REMARQUE : avec un COPIER-COLLER de ce que vous avez fait pour `suivant`, ça prend une seconde.

Corrigé

```
void suivant_gen(int gen, char in[], char out[])
{
```

```
    int n = strlen(in);
    for(int i = 0; i < n; i++)
    {
        int config = 0;
        if (i > 0 && in[i-1] == '#')
        {
            config += 4;
        }
        if (in[i] == '#')
        {
            config += 2;
        }
        if (i < n-1 && in[i+1] == '#')
        {
            config += 1;
        }
        if (regle_gen(gen, config) == 1)
        {
            out[i] = '#';
        }
        else
        {
            out[i] = ' ';
        }
    }
}
```

Question 10

Écrire une fonction de prototype

```
void itere_regle_gen(int gen, char depart[], char temp[], char sortie[], int iterations);
```

qui applique `iterations` fois la fonction `suivant_gen` en partant de la configuration `depart` et place la configuration finale dans `sortie`.

Comme on ne sait pas encore allouer une telle chaîne dynamiquement, on vous donne aussi une chaîne `temp` comme espace de travail.

ATTENTION : les chaînes `depart`, `temp` et `sortie` sont de même longueur.

REMARQUE : avec un COPIER-COLLER de ce que vous avez fait pour `itere_regle`, ça prend une seconde.

Corrigé

```
void itere_regle_gen(int gen, char depart[], char temp[], char sortie[], int iterations)
{
    copie_chaine(sortie, depart);
    copie_chaine(temp, depart);
    for (int i = 0; i < iterations; i++)
    {
        suivant_gen(gen, temp, sortie);
        copie_chaine(temp, sortie);
    }
}
```

Question 11

En vous inspirant des 30 itérations calculées par le code donné plus haut, affichez les 30 premières itérations des 256 règles en partant d'une même configuration.

✍

Corrigé

Il suffit de rajouter une boucle sur les numéros de règle, de 0 à 255 inclus. Plutôt que de tout recalculer à chaque ligne avec `itere_regle_gen`, on peut appliquer une fois `suivant_gen` par ligne.

```
char init[81];
for(int i = 0; i < 80; i++)
{
    init[i] = ' ';
}
init[80] = '\0';
init[40] = '#';

char temp[81], sortie[81];
for (int gen = 0; gen < 256; gen++)
{
    printf("Règle %d\n", gen);
    strcpy(sortie, init);
    for(int j = 0; j < 30; j++)
    {
        printf("%s\n", sortie);
        strcpy(temp, sortie);
        suivant_gen(gen, temp, sortie);
    }
    printf("\n");
}
```

11

TRAVAUX PRATIQUES 11

Introduction au C (version notebook)

DANS CE CHAPITRE

11.1 Cadre de ce TP
11.2 Les bases du C
11.3 J'aimerais tant (re)voir Syracuse
11.4 Les tableaux
11.5 Tri par sélection

11.1 Cadre de ce TP

Ce TP a d'abord été proposé dans un notebook Jupyter muni d'un noyau interactif permettant d'écrire du C++ : 98 % de la syntaxe du C est directement utilisable en C++, donc ça ne change pas grand-chose. Ici, on compile de vrais programmes C.

Normalement un programme en C est composé d'un fichier `.c` (le plus souvent plusieurs qui sont réunis) contenant une fonction `main` qui est le point d'entrée du programme.

Voici un exemple de contenu d'un tel fichier :

```c
// déclaration de fonctions d'impression pour pouvoir les utiliser
#include <stdio.h>

int x = 3; // une variable globale entière x pour tout le programme

// déclaration d'une fonction qui prend deux entiers en entrée et renvoie un entier
int ajoute(int a, int b)
{
    return a+b;
}

// la fonction principale

int main(void)
{
    int a = 2; // déclaration de variables locales
    int b; // non initialisée ici

    b = 3; // nécessite que b soit déclarée.

    printf("La somme de %d + %d = %d\n",
           b, x, ajoute(b, x)); // la fonction principale d'affichage, véritable couteau-suisse
                                // ici %d signifie un entier qui figure dans la liste juste après

    // On retrouve des boucles for
    a = 0;
```

```
    for (int i = 0; i < 10; i++) // mais on doit gérer notre compteur nous-même
    {
        a = a + i;
    }
    printf("La somme des entiers de 0 à 9 vaut %d\n", a);

    return 0; // pour main signifie que tout va bien
}
```

qui affiche

```
La somme de 3 + 3 = 6
La somme des entiers de 0 à 9 vaut 45
```

On procédera comme en `OCaml` pour ce TP : on vous demandera d'écrire le corps de fonctions, puis une série de tests sera effectuée pour vous assurer que vous avez réussi. Les tests sont écrits avec `assert` (il faut `#include <assert.h>` en tête du fichier) : `assert(c);` interrompt le programme avec un message d'erreur si la condition `c` est fausse. On les place dans la fonction `main`.

Question 1

Écrire une fonction `int multiplie(int a, int b)` qui renvoie le produit de `a` et de `b` (allez voir le corps de `ajoute` au-dessus et adaptez).

Test :

```
assert(multiplie(2, 3) == 6);
```

Corrigé

```
int multiplie(int a, int b)
{
    return a * b;
}
```

11.2 Les bases du C

11.2.1 Types et variables

Mis à part l'ajout des points-virgules à la fin des instructions, la différence principale entre `OCaml`, `Python` et `C` dans ce qu'on vient de voir se situe au niveau des instructions explicites de type et donc de la notion de déclaration de variable.

On a en `C` des types de base sur lesquels on reviendra dans les exercices : `int`, `float`, `bool`, mais une variable doit être **déclarée** comme ayant un type donné et, bien qu'on puisse en changer la valeur, on ne pourra pas changer le type des données qu'elle contient.

Pour déclarer une variable `x` ayant le type `t` on écrit `t x;`, par exemple `int b;` ici. Une fois une variable déclarée, on peut l'utiliser en lecture et en écriture. Mais il faut écrire dedans AVANT d'y lire quoi que ce soit. Pour cela, on pourra écrire `x = val;` pour que la variable `x` prenne la valeur `val`. Il est possible de déclarer et d'initialiser une variable, comme dans `int x = val;`.

ATTENTION : si on écrit juste `int x;` sans valeur, on ne peut pas supposer que `x` contienne une valeur spécifique comme `0`. On peut juste dire que `x` contient un entier quelconque qu'on se gardera bien d'utiliser dans un calcul.

Ainsi, comme `OCaml`, `C` est un langage typé statiquement, mais dont les types sont explicites.

```c
int x; // x n'est pas initialisé mais c'est un entier
float y = 3.2; // y est un flottant initialisé à 3.2
bool b = true; // b est un booléen qui vaut true (nécessite stdbool.h)
char c = 'a'; // le type des caractères
```

Un type propre à `C` est le type de rien, qui se nomme `void`. Cela correspond au type `unit` de `OCaml`.

11.2.2 Blocs et portée

En `C` on a une notion de bloc : ce sont des instructions ou des déclarations de variables entourées d'accolades, comme dans :

```c
{
    int x = 3;
    printf("x vaut %d\n", x);
}
```

Une variable déclarée dans un bloc n'existe que dans celui-ci. Dès qu'on sort du bloc on ne peut plus y accéder. Les fonctions, les instructions conditionnelles ou les boucles utilisent des blocs.

```c
// exemple avec deux blocs imbriqués où on retrouve la valeur précédente de x
// qui a juste été masquée dans le sous-bloc
{
    int x = 3;
    printf("x vaut %d\n", x);

    {
        int x = 4;
        printf("x vaut %d\n", x);
    }

    printf("x vaut %d\n", x);
}
```

```
x vaut 3
x vaut 4
x vaut 3
```

```c
// ATTENTION par contre, le code suivant utilise x dans le bloc pour le changer de
// valeur et comme c'est le x du bloc principal, ça le modifie
{
    int x = 3;
    printf("x vaut %d\n", x);

    {
        x = 4;
        printf("x vaut %d\n", x);
    }

    printf("x vaut %d\n", x);
}
```

```
x vaut 3
x vaut 4
x vaut 4
```

11.2.3 Fonctions

Une fonction en `C` a un type de retour et chacun des arguments est donné avec son type. S'ensuit un bloc d'instructions qui peut comporter une ou plusieurs instructions `return v;` qui vont renvoyer la valeur `v`. Pour une fonction de type de retour `void`, on peut omettre le `return` ou écrire `return;` sans valeur.

On définit une fonction ainsi :

```
type_retour nom_fonction(type_arg1 nom_arg1, ..., type_argn nom_argn)
{
    // bloc d'instructions qui constitue le corps
    // de la fonction
}
```

On est donc très proche des fonctions `Python` mais avec cette contrainte de type.

```
int f(int a, char c)
{
    printf("a=%d et c=%c\n", a, c);
    return a*a;
}
```

On appelle alors la fonction comme en `Python` avec `nom_fonction(val1, ..., valn)` : `int r = f(3, 'z');` affiche `a=3 et c=z` et place la valeur `9` dans `r`.

On voit que la valeur de sortie de la fonction est bien le résultat du `return`, le reste c'est juste de l'affichage.

Les arguments sont des variables qu'on peut utiliser directement : cela ne changera pas la valeur passée en paramètre si on les redéfinit. Une fonction peut s'appeler elle-même, avec toutes les réserves vues quant à l'utilisation de la pile.

11.2.4 Instructions conditionnelles

Pour effectuer une **instruction** conditionnelle, on écrit

```
if (condition)
{
    // bloc d'instructions à exécuter quand la condition est vraie
}
else
{
    // bloc d'instructions à exécuter quand la condition est fausse
}
```

Par exemple, on pourra écrire :

```
if (n % 2 == 0)
{
    printf("n est pair\n");
}
else
{
    printf("n est impair\n");
}
```

Ici, comme il n'y a qu'une instruction, on pourrait ne pas faire de bloc et écrire juste l'instruction. Mais on se gardera de faire ça pour le moment, car c'est une source classique d'erreurs.

On note qu'il n'est pas nécessaire d'avoir un `else`. Ici, comme en `Python` et contrairement à `OCaml`, on n'a pas de notion de type associé aux branches, ce sont juste des suites d'instructions.

Notamment : en `C`, ce `if` est une **instruction** et pas une expression. Si on peut écrire

```
let x = if n mod 2 = 0 then 1 else 2
```

en `OCaml`, en `C`, on devra écrire :

```
int x;

if (n % 2 == 0)
{
    x = 1;
```

```
}
else
{
    x = 2;
}
```

Au passage, remarquons qu'on utilise `==` comme en `Python`. Par contre, contrairement à `Python`, comme les déclarations ne sortent pas des blocs, on ne peut pas transposer

```
if n % 2 == 0:
    x = 1
else:
    x = 2
print("x vaut ", x)
```

car en `C` le programme suivant est erroné

```
if (n % 2 == 0)
{
    int x = 1;
}
else
{
    int x = 2;
}
printf("x vaut %d\n", x);
```

puisque `x` n'est pas définie en dehors des blocs.

Exercice

Pour la question suivante, on rappelle la définition de la suite de Syracuse :

Étant donné $a \in \mathbb{N}^*$, on appelle suite de Syracuse de premier terme a la suite $(u_n)_{n\in\mathbb{N}}$ définie par $u_0 = a$ et pour $n \in \mathbb{N}$:

$$u_{n+1} = \begin{cases} \frac{u_n}{2} & \text{si } u_n \text{ est pair} \\ 3u_n + 1 & \text{sinon.} \end{cases}$$

Indications :

- pour déterminer si un nombre `a` est pair on peut utiliser le modulo et tester `a % 2 == 0` comme en `Python` ;
- une fonction peut avoir plusieurs `return`, donc quand vous avez calculé une valeur, il suffit de la renvoyer.

Question 2

Écrire une fonction `int suivant(int u)` qui renvoie la valeur du terme qui suit `u` dans la suite.

Tests :

```
assert(suivant(14) == 7);
assert(suivant(7) == 22);
```

Corrigé

```
int suivant(int u)
{
    if (u % 2 == 0)
    {
        return u/2;
    }
    else
    {
```

```
        return 3*u+1;
    }
}
```

11.2.5 Boucles conditionnelles

Pour effectuer une boucle conditionnelle, on écrit

```
while(condition)
{
    // instructions à réaliser
}
```

Ce sont, comme en `Python`, les boucles les plus simples mais elles demandent de gérer plus de choses qu'une boucle inconditionnelle. Par exemple, pour calculer la somme $\sum_{k=0}^{n} k$, on doit nécessairement gérer les indices à la main :

```
int somme_entiers(int n)
{
    int k = 0; // l'indice
    int s = 0;  // la somme courante

    while (k <= n)
    {
        s = s + k;
        k = k+1;
    }

    return s;
}
```

Le cas le plus approprié d'une telle boucle est quand on ne peut maîtriser le nombre d'itérations, comme dans la fonction suivante.

Question 3

Écrire une fonction `int sup_log2(int n)` qui renvoie, pour $n \geq 0$, le plus petit entier naturel d tel que $n < 2^d$ (attention, cette puissance n'existe pas en C).

Tests :

```
assert(sup_log2(0) == 0);
assert(sup_log2(1) == 1);
assert(sup_log2(15) == 4);
```

Corrigé

```
int sup_log2(int n)
{
    int d = 0;
    int p = 1;
    while (p <= n)
    {
        p = p * 2;
        d = d + 1;
    }

    return d;
}
```

11.2.6 Boucles `for`

Contrairement à `Python`, il n'y a aucun mécanisme permettant de parcourir des valeurs. Mais contrairement aux boucles `for` de `OCaml`, il ne s'agit pas simplement de boucles inconditionnelles. Le mécanisme est plus complexe.

Commençons par un exemple en reprenant la fonction `somme_entiers` :

```
int somme_entiers(int n)
{
    int s = 0;

    for(int k = 0; k <= n; k++)
    {
        s = s + k;
    }

    return s;
}
```

Déjà, précisons que `k++` est une manière d'écrire `k = k+1`. On parle d'incrémentation de `k`.

Ensuite, on peut voir qu'il y a trois parties dans la boucle `for` :

```
for (initialisation; condition; miseajour)
{
    // instructions
}
```

L'initialisation est la plupart du temps la déclaration d'une variable de boucle. Ici, on peut utiliser le raccourci permettant de déclarer et définir la variable directement. Par opposition au fait d'écrire :

```
int k;
for (k = 0; k <= n; k++)
{
    s = s+k;
}
```

Ensuite, vient la condition d'arrêt qui est très proche de celle d'un `while`.

Enfin, la mise à jour est une instruction qui sera effectuée à la fin de chaque itération. Finalement, une boucle `for` en `C` n'est qu'un `while` déguisé :

```
initialisation;

while(condition)
{
    // instructions

    miseajour;
}
```

Pour autant, la lisibilité d'une boucle `for` est bien meilleure que celle d'un `while` car elle structure plus en mettant en valeur les **vraies** instructions du corps de boucle. On ne sera pas surpris de trouver ainsi des `for` dès que c'est possible.

Question 4

Écrire une fonction `int somme_carres(int n)` qui renvoie la somme des carrés des entiers de 0 à n : $0^2 + 1^2 + \ldots + n^2$.

Tests :

```
assert(somme_carres(13) == 819);
assert(somme_carres(0) == 0);
assert(somme_carres(1) == 1);
```

Corrigé

```
int somme_carres(int n)
{
    int s = 0;
    for (int k = 0; k <= n; k++)
    {
        s = s + k*k;
    }
    return s;
}
```

11.3 J'aimerais tant (re)voir Syracuse

Question 5

Écrire une fonction `int syracuse(int a, int n)` qui renvoie le n-ième terme de la suite de Syracuse de premier terme a.

ATTENTION : on utilisera `suivant` et on ne fera pas de copier-coller de code.

Tests :

```
assert(syracuse(3, 5) == 4);
assert(syracuse(1, 0) == 1);
assert(syracuse(1, 1) == 4);
assert(syracuse(1, 2) == 2);
```

Corrigé

```
int syracuse(int a, int n)
{
    int u = a;
    for (int k = 1; k <= n; k++)
    {
        u = suivant(u); // on calcule u(k) ici
    }
    return u;
}
```

Question 6

Écrire une fonction `int temps_de_vol(int a)` qui renvoie le temps de vol pour a, c'est-à-dire le plus petit entier n tel que $u_n = 1$ quand $u_0 = a$.

Tests :

```
assert(temps_de_vol(1) == 0);
assert(temps_de_vol(14) == 17);
```

Corrigé

```
int temps_de_vol(int a)
{
```

```
    int n = 0;
    int u = a;
    while (u != 1)
    {
        u = suivant(u);
        n = n+1;
    }
    return n;
}
```

Question 7

Écrire une fonction `int altitude_maximale(int a)` qui renvoie l'altitude maximale de a, c'est-à-dire la plus grande valeur prise par la suite quand $u_0 = a$.

Tests :

```
assert(altitude_maximale(1) == 4);
assert(altitude_maximale(2) == 4);
assert(altitude_maximale(14) == 52);
assert(altitude_maximale(16) == 16);
```

Corrigé

La suite finit par boucler sur 4, 2, 1, donc l'altitude maximale vaut au moins 4, et il ne faut pas oublier le premier terme $u_0 = a$.

```
int altitude_maximale(int a)
{
    int m = 4; // la suite finit par boucler sur 4, 2, 1
    if (a > m)
    {
        m = a;
    }
    int u = a;
    while (u != 1)
    {
        u = suivant(u);
        if (u > m)
        {
            m = u;
        }
    }
    return m;
}
```

11.4 Les tableaux

11.4.1 Déclaration et initialisation

La structure de données la plus primitive en `C` est celle des tableaux. Attention, ce sont des tableaux de taille fixe comme les `'a array` de `OCaml`. Pour déclarer un tableau `t` de `n` valeurs (**où `n` est une CONSTANTE et pas une variable**) d'un type `T` on écrit

```
T t[n];
```

Par exemple `int a[10];` déclare `a` comme étant un tableau de 10 entiers.

On peut alors accéder en lecture et en écriture à la i-ème case du tableau en écrivant `t[i]`. Ici tout se passe comme si `t` était une collection de variables accessibles par des indices. De plus, on a l'assurance que les cases se suivent linéairement en mémoire, ce qui sera utile dans la suite.

REMARQUE : C est un langage sans garde-fou, on peut tout à fait écrire `t[100]` pour un tableau de taille 3 et ainsi aller écrire n'importe où en mémoire (voir la page Travaux Pratiques d'apprentissage du C et l'explosion d'Ariane 5).

ATTENTION : dans presque tous les cas, les valeurs ne sont pas initialisées, et comme pour les variables, il faut attribuer des valeurs avant de travailler sur le tableau. Notamment, les cases ne sont pas initialisées à zéro.

Il est possible d'initialiser un tableau au moment de sa déclaration. Pour cela, on met les valeurs entre accolades comme ici :

```
int t[3] = { 1, 3, 5 };
```

Ce qui équivaut à l'écriture de :

```
int t[3];
t[0] = 1;
t[1] = 3;
t[2] = 5;
```

Si jamais on donne moins de `n` valeurs pour `t[n]`, on va compléter par des valeurs nulles. Cela permet d'utiliser le cas spécial `{ 0 }` pour tout initialiser à zéro. Les trois définitions suivantes sont alors équivalentes :

```
int t[3] = { 0, 0, 0 };
int t[3] = { 0, 0 };
int t[3] = { 0 };
```

(La forme `{}` est une extension de `gcc`, normalisée seulement en C23.)

Quand on donne une succession complète de valeurs, on connaît déjà le nombre d'éléments du tableau. Il est alors assez redondant de le donner une seconde fois dans la déclaration. On dispose ainsi d'un mécanisme automatique en écrivant juste `T t[] = ...;`. On peut reprendre la déclaration précédente ainsi :

```
int t[] = { 1, 3, 5 };
```

11.4.2 Tableaux en arguments

Pour passer un tableau en paramètre à une fonction on pourra utiliser, entre autres, `int t[]` ou `int t[10]` avec une taille précise, mais **il n'y a aucune vérification de taille en C**. On prendra donc l'habitude de donner en arguments le nombre d'éléments du tableau.

Ainsi la fonction suivante calcule la somme des éléments d'un tableau.

```
int somme_tableau(int t[], int nb_elements)
{
    int s = 0;

    for (int i = 0; i < nb_elements; i++)
    {
        s = s + t[i];
    }

    return s;
}
```

```
int t[] = { 0, 1, 2, 3 };
printf("%d\n", somme_tableau(t, 4)); // 6
// mais on peut très bien appeler la fonction sur le même tableau avec une taille plus petite
printf("%d\n", somme_tableau(t, 3)); // 3
// ou une taille plus grande mais alors rappelez-vous de la fusée...
printf("%d\n", somme_tableau(t, 1000)); // valeur absurde, ou plantage
```

Question 8

Écrire une fonction `int recherche(int t[], int nb_elements, int x)` qui renvoie le plus petit indice de `x` dans le tableau `t` ; si `x` n'est pas présent, on renvoie `-1` (qui est toujours un indice invalide).

Rappel : on peut faire des `return` quand on veut.

Tests :

```
int t[] = { 0, 2, 2, 6 };
assert(recherche(t, 4, 2) == 1);
assert(recherche(t, 1, 2) == -1);
assert(recherche(t, 0, 2) == -1);
assert(recherche(t, 1, 0) == 0);
assert(recherche(t, 4, 7) == -1);
```

Corrigé

```
int recherche(int t[], int nb_elements, int x)
{
    for (int i = 0; i < nb_elements; i++)
    {
        if (t[i] == x)
        {
            return i;
        }
    }

    return -1;
}
```

Question 9

Écrire une fonction `int maximum(int t[], int nb_elements)` qui renvoie la valeur du maximum du tableau `t`. On considère que `nb_elements` > 0, donc il existe toujours.

Tests :

```
int t[] = { 5, 0, 2, 1, 6 };
assert(maximum(t, 4) == 5);
assert(maximum(t, 5) == 6);
```

Corrigé

```
int maximum(int t[], int nb_elements)
{
    int m = t[0];
    for (int i = 1; i < nb_elements; i++)
    {
        if (m < t[i])
        {
            m = t[i];
        }
    }

    return m;
}
```

Question 10

En déduire une fonction `int arg_max(int t[], int nb_elements)` qui renvoie le plus petit indice du maximum de `t` où `nb_elements` > 0. ON NE FERA PAS DE BOUCLES ICI.

Tests :

```
int t[] = { 5, 0, 5, 1, 6 };
assert(arg_max(t, 4) == 0);
assert(arg_max(t, 5) == 4);
```

Corrigé

```
int arg_max(int t[], int nb_elements)
{
    return recherche(t, nb_elements, maximum(t, nb_elements));
}
```

11.4.3 Des tableaux particuliers : les chaînes de caractères

Une chaîne de caractères en C est juste un tableau de `char` dont on impose que la dernière case soit nulle. Dans le contexte des caractères on note `'\0'` le caractère nul, mais cela correspond à la valeur 0.

Ainsi :

```
char chaine[] = { 'B', 'o', 'n', 'j', 'o', 'u', 'r', '\0' };
```

est la chaîne contenant `Bonjour`. Comme écrire ainsi les chaînes serait très fastidieux, on utilise la notation usuelle :

```
char chaine[] = "Bonjour";
```

et tout se passe comme dans la définition ci-dessus : `chaine[7]` vaut bien `'\0'`.

Attention, les caractères en C sont limités : un `char` est un octet, qui ne peut contenir qu'un caractère ASCII. On peut écrire « 歩こう ! » ou des caractères accentués dans une chaîne, mais chacun de ces caractères occupe alors plusieurs `char` (codage UTF-8). Il y a un certain nombre de caractères spéciaux comme `\n` pour effectuer un saut de ligne.

On va maintenant revenir sur la fonction `printf` vue précédemment. C'est une fonction spéciale qui peut prendre un nombre variable d'arguments mais qui en aura toujours au moins un : `printf(format, ...)` où `format` est une chaîne de caractères. Cette chaîne peut comporter des symboles spéciaux commençant par `%` et affichant, dans l'ordre de gauche à droite, les valeurs passées en argument après `format`.

On a par exemple `%d` pour afficher un entier signé, `%s` pour une chaîne de caractères ou `%c` pour un caractère. Exemple :

```
char s[] = "Test";
int i = 2;
int r = printf("Le %deme caractère de la chaine %s est %c\n",
     i, s, s[i]); // oui on commence à compter à 0
```

affiche `Le 2eme caractère de la chaine Test est s` et `r` vaut 43.

Le nombre renvoyé par `printf` est le nombre de caractères (plus précisément d'octets : le `è` en compte deux) affichés après ces substitutions.

On pourra utiliser la fonction suivante pour afficher un tableau :

```
void print_tableau(int t[], int nb_elements)
{
    for(int i = 0; i < nb_elements; i++)
    {
        char c = ',';
        if (i == nb_elements - 1)
```

```
            c = '\n'; // fin de ligne pour le dernier élément.
        printf("%d%c", t[i], c);
    }
}
```

Ainsi, `int t[] = {1, 2, 3}; print_tableau(t, 3);` affiche `1,2,3`.

11.4.4 Tableaux en valeur de retour ?

On ne peut pas renvoyer un tableau défini ainsi en C. En effet, il correspond à une zone mémoire locale à la fonction qui ne peut en sortir. Comme souvent en `C`, on peut l'écrire, mais ça va avoir un comportement erroné pouvant aboutir à une catastrophe (voir les *pointeurs pendants* dans le chapitre Gestion de la mémoire).

La bonne manière d'avoir un tableau en valeur de retour, c'est paradoxalement de le passer en paramètre et d'écrire dedans.

La fonction suivante, par exemple, va renvoyer un tableau **à l'envers**. Pour bien identifier le tableau en entrée et le tableau en sortie, on les a nommés `in` et `out`.

```
void renverse_tableau(int in[], int out[], int nb_elts)
{
    for(int i = 0; i < nb_elts; i++)
    {
        out[i] = in[nb_elts-1-i];
    }
}
```

```
int t[] = {1,2,3};
int q[3];
renverse_tableau(t, q, 3);
print_tableau(t, 3); // 1,2,3
print_tableau(q, 3); // 3,2,1
```

11.5 Tri par sélection

On va maintenant revenir sur un problème classique d'algorithmique : la notion de **tri**. On considère des tableaux et on va étudier différentes manières de les trier.

On a des méthodes de tri dites **en place** où on trie un tableau directement en changeant ses valeurs, et des méthodes **non en place** qui vont placer dans un autre tableau le résultat. Pour effectuer des tris en place, on va considérer une fonction permettant d'échanger deux valeurs dans un tableau.

Question 11

Écrire une fonction `void swap(int t[], int i, int j)` qui échange dans `t` les valeurs des cases d'indices `i` et `j`.

Test :

```
int t[] = { 1, 2, 3 };
swap(t, 0, 2);
assert(t[0] == 3 && t[2] == 1);
```

Corrigé

```
void swap(int t[], int i, int j)
{
    int temp = t[i];
    t[i] = t[j];
    t[j] = temp;
}
```

Le tri par sélection est un tri qui va constituer un sous-tableau trié de plus en plus grand en tête du tableau. C'est une méthode en place. À l'étape i, on considère que les cases d'indices 0 à $i-1$ sont triées et contiennent les i plus petites valeurs du tableau.

- Si $i = n$, où n est la taille du tableau, alors le sous-tableau est en fait le tableau complet : il est trié.
- Sinon, on va chercher le minimum du tableau entre les indices i et $n-1$ et on le place à l'indice i. On passe alors à l'étape $i+1$.

Question 12

Écrire une fonction `int indice_minimum_sous_tableau(int t[], int i, int nb_elts)` qui renvoie l'indice du plus petit élément du tableau `t` à partir de l'indice `i`. On appellera toujours cette fonction avec `i < nb_elts`.

Tests :

```
int t[] = { 4, 5, 2, 1, 6 };
assert(indice_minimum_sous_tableau(t, 0, 5) == 3);
assert(indice_minimum_sous_tableau(t, 2, 5) == 3);
assert(indice_minimum_sous_tableau(t, 4, 5) == 4);
```

Corrigé

```
int indice_minimum_sous_tableau(int t[], int i, int nb_elts)
{
    int val_min = t[i];
    int indice_min = i;

    for (int j = i+1; j < nb_elts; j++)
    {
        if (val_min > t[j])
        {
            val_min = t[j];
            indice_min = j;
        }
    }

    return indice_min;
}
```

Question 13

Écrire une fonction `void tri_par_selection(int t[], int nb_elts)` qui trie `t` par sélection à l'aide de la fonction précédente.

Test :

```
int t[] = { 4, 5, 2, 1, 6 };
int q[] = { 1, 2, 4, 5, 6 };
tri_par_selection(t, 5);
for (int i = 0; i < 5; i++)
{
    assert(t[i] == q[i]);
}
```

Corrigé

```
void tri_par_selection(int t[], int nb_elts)
{
    for (int i = 0; i < nb_elts; i++)
    {
        int j = indice_minimum_sous_tableau(t, i, nb_elts);
```

```
            swap(t, i, j);
        }
    }
```

L'invariant décrit plus haut est maintenu : après l'itération i, les cases 0 à i contiennent les $i+1$ plus petites valeurs, triées. Le nombre de comparaisons est $\sum_{i=0}^{n-1}(n-1-i) = \frac{n(n-1)}{2}$, donc le tri est en $O(n^2)$ dans tous les cas.

12

TRAVAUX PRATIQUES 12

Tableaux, chaînes et automates de Wolfram (version notebook)

DANS CE CHAPITRE

12.1 Conversions binaires
12.2 Retour sur les chaînes
12.3 Automates de Wolfram

Les tests sont écrits avec `assert` (il faut `#include <assert.h>`) et sont à placer dans la fonction `main`. On aura aussi besoin de `stdio.h` et de `string.h`.

12.1 Conversions binaires

On considère l'écriture binaire $n = \sum_{k=0}^{+\infty} b_k 2^k$ d'un entier naturel n, où les $b_k \in \{0, 1\}$ sont presque tous nuls. En pratique, on ne considère que des nombres dont la taille est majorée, donc de la forme $\sum_{k=0}^{N-1} b_k 2^k$. N est le nombre de bits sur lequel on a écrit l'entier n. Le bit b_{N-1} peut être nul, c'est-à-dire qu'on n'impose pas que $b_{N-1} \neq 0$; par contre, on a forcément $n < 2^N$.

Les différents types de valeurs entières en `C` sont de la forme `t` ou `unsigned t` où :

- `unsigned` signifie qu'on ne peut stocker que des valeurs positives ; si on l'omet on peut avoir des valeurs négatives, sachant qu'il faut un bit pour coder l'information du signe : un même type peut prendre des valeurs deux fois plus grandes en `unsigned` ;
- `t` peut valoir (types principaux) **ATTENTION : c'est dépendant du compilateur, on donne ici les valeurs usuelles pour un compilateur moderne et un processeur 64 bits !** :
  - `char` : pour des valeurs sur $N = 8$ bits, donc 1 octet, donc de 0 à 255 en `unsigned` (de -128 à 127 sinon) ;
  - `short` : pour des valeurs sur $N = 16$ bits, 2 octets, donc de 0 à $2^{16} - 1$ en `unsigned` ;
  - `int` : pour des valeurs sur $N = 32$ bits, donc 4 octets, donc de 0 à $2^{32} - 1$ en `unsigned` ;
  - `long` : pour des valeurs sur $N = 64$ bits, donc 8 octets, donc de 0 à $2^{64} - 1$ en `unsigned`.

  En version signée, un type sur N bits va de -2^{N-1} à $2^{N-1} - 1$.

Dans le doute, on peut connaître la taille en nombre d'octets en utilisant `sizeof` : `sizeof(long)` vaut `8`.

Il est aussi possible d'utiliser des types explicites comme `int32_t` ou `int64_t` (dans `stdint.h`).

Question 1

Écrire une fonction `void vers_binaire(unsigned int x, unsigned char b[])`. On considère ici que `b` est un tableau de 32 `unsigned char` ; cette fonction renvoie dans `b` la représentation en binaire de `x` de sorte que `b[i]` contienne b_i de l'expression au-dessus (comme toujours, voir les tests en cas de doute).

Tests :

```
unsigned char b[32];
unsigned char out[] = { 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1,
                        1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
vers_binaire(123457, b);
for(int i = 0; i < 32; i++)
{
    assert(b[i] == out[i]);
}
vers_binaire(4294967295, b);
for(int i = 0; i < 32; i++)
{
    assert(b[i] == 1);
}
```

Corrigé

```
void vers_binaire(unsigned int x, unsigned char b[])
{
    for (int i = 0; i < 32; i++)
    {
        b[i] = x % 2;
        x = x / 2;
    }
}
```

Question 2

Écrire une fonction `unsigned int depuis_binaire(unsigned char b[])` qui renvoie l'entier représenté par le tableau de 32 bits `b`. On fera en sorte que `depuis_binaire` et `vers_binaire` soient réciproques l'une de l'autre.

Tests :

```
unsigned char b[] = { 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1,
                      1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
assert(depuis_binaire(b) == 123457);
unsigned char c[32];
for (int i = 0; i < 32; i++) c[i] = 1;
assert(depuis_binaire(c) == 4294967295);
```

Corrigé

```
unsigned int depuis_binaire(unsigned char b[])
{
    unsigned int x = 0;
    unsigned int p = 1;
    for (int i = 0; i < 32; i++)
    {
        if (b[i] == 1)
        {
            x = x + p;
        }
        p = 2*p;
    }
    return x;
}
```

12.2 Retour sur les chaînes

On rappelle qu'une chaîne de caractères en C est juste un tableau de `char`. Les `char` sont des valeurs entières sur un octet (8 bits). En pratique, on ne s'intéressera pas à la valeur entière mais uniquement à leur représentation sous forme de caractère, donc on écrira `'a'` et pas `97` bien que ce soit la même valeur : `'a' == 97` est vrai.

On ne considère ici que les caractères ASCII dont la valeur est comprise entre 0 et 127. **ATTENTION** : comme les `char` sont limités, on ne peut pas écrire `for(char i = 0; i < 128; i++)` car, sur la plupart des machines où `char` va de −128 à 127, la comparaison avec `128` est toujours vraie : la boucle ne termine pas.

Les chaînes finissent toutes par `'\0'` et donc on peut passer des chaînes en argument sans se soucier de donner leur taille grâce à ce caractère spécial. On appelle ça une **sentinelle**.

Pour faire une boucle dans un tableau de taille `n`, on écrit `for(i = 0; i < n; i++)` mais ici, comme on ne connaît pas `n`, il faut utiliser la sentinelle, par exemple en écrivant `for(i = 0; s[i] != '\0'; i++)`.

Question 3

Écrire une fonction `int longueur_chaine(char s[])` qui renvoie la longueur de la chaîne `s`.

Tests :

```
assert(longueur_chaine("test") == 4);
assert(longueur_chaine("") == 0);
```

Corrigé

```
int longueur_chaine(char s[])
{
    int i;
    for (i = 0; s[i] != '\0'; i++)
    {
    }
    return i;
}
```

Cette fonction, comme d'autres sur les chaînes, est définie directement dans la bibliothèque standard du C (en-tête `string.h`) : c'est `strlen`. On pourra donc l'utiliser dans la suite pour pouvoir faire des boucles avec la taille comme dans un tableau usuel.

Il est probable que pour écrire la fonction suivante vous souhaitiez pouvoir sortir prématurément d'une boucle comme en `Python`. C'est également possible en `C`, on peut utiliser :

- `break;` pour sortir de la boucle ;
- `continue;` pour finir l'itération et passer à l'itération suivante.

Question 4

Écrire une fonction `int recherche_motif(char s[], char m[])` qui renvoie l'indice de début de la première occurrence du motif `m` dans `s` s'il est présent, ou `-1` sinon.

Tests :

```
char s[] = "tetututo";
assert(recherche_motif(s, "tu") == 2);
assert(recherche_motif(s, "to") == 6);
assert(recherche_motif(s, "ti") == -1);
```

Corrigé

```
int recherche_motif(char s[], char m[])
{
    int n = strlen(s);
    int p = strlen(m);
    for (int i = 0; i <= n-p; i++)
    {
        int j;
```

```
        for (j = 0; j < p; j++)
        {
            if (s[i+j] != m[j])
            {
                break;
            }
        }
        if (j == p)
        {
            return i;
        }
    }

    return -1;
}
```

En sortie de la boucle sur `j`, on a `j == p` exactement quand on n'est pas sorti par le `break`, c'est-à-dire quand le motif est présent à l'indice `i`. Si le motif est plus long que la chaîne, `n-p < 0` et on renvoie directement `-1`. Cet algorithme naïf est en $O(np)$ dans le pire cas.

Question 5

Écrire une fonction `void occurrences(char s[], int occ[])` qui renvoie dans le tableau `occ`, contenant 128 valeurs NON INITIALISÉES, les nombres d'occurrences de chaque caractère dans la chaîne `s`.

Tests :

```
int occ[128];
occurrences("test", occ);
for(int i = 0; i < 128; i++)
{
    if (i == 't') assert(occ[i] == 2);
    else if (i == 'e' || i == 's') assert(occ[i] == 1);
    else assert(occ[i] == 0);
}
```

Corrigé

```
void occurrences(char s[], int occ[])
{
    for (int i = 0; i < 128; i++)
    {
        occ[i] = 0;
    }
    for (int i = 0; s[i] != '\0'; i++)
    {
        occ[(unsigned char) s[i]]++;
    }
}
```

La conversion en `unsigned char` évite un indice négatif si `char` est signé ; on suppose de plus que la chaîne ne contient que des caractères ASCII, sinon l'indice peut dépasser 127.

12.3 Automates de Wolfram

12.3.1 Présentation et premier automate

On va s'intéresser ici à des automates cellulaires unidimensionnels. On considère ainsi une ligne infinie de cellules. Chaque cellule a exactement deux cellules voisines, une à sa droite et une à sa gauche. Les cellules contiennent un état. Pour simplifier ici, on ne va considérer que deux états 1 et 0, noir ou blanc, ou encore `'#'` et `' '` dans la suite.

Afin de représenter informatiquement de telles lignes infinies, on va considérer un nombre fini de cellules en supposant que toutes les autres sont dans l'état 0.

Un automate de Wolfram est la donnée d'un ensemble de règles de la forme



Figure 12.1 — Les huit règles $(g, c, d) \to c'$: en haut le voisinage, en bas le nouvel état (bleu = 1).

ce qui peut se lire sous la forme $(g, c, d) \to c'$: connaissant l'état g de la cellule de gauche, c de la cellule courante et d de la cellule de droite, le nouvel état de la cellule courante est c'.

Question 6

Écrire une fonction `int regle_vois(int g, int c, int d)` qui prend un triplet (g, c, d) sous la forme de valeurs 0 ou 1 et renvoie c' selon la règle au-dessus en image, selon la correspondance case pleine (bleue) = 1, case vide = 0.

Tests :

```
assert(regle_vois(0,0,0) == 0);
assert(regle_vois(0,0,1) == 1);
assert(regle_vois(0,1,0) == 1);
assert(regle_vois(0,1,1) == 1);
assert(regle_vois(1,0,0) == 1);
assert(regle_vois(1,0,1) == 0);
assert(regle_vois(1,1,0) == 0);
assert(regle_vois(1,1,1) == 0);
```

Corrigé

```
int regle_vois(int g, int c, int d)
{
    if (g == 1 && c == 1 && d == 1)
        return 0;
    if (g == 1 && c == 1 && d == 0)
        return 0;
    if (g == 1 && c == 0 && d == 1)
        return 0;
    if (g == 1 && c == 0 && d == 0)
        return 1;
    if (g == 0 && c == 1 && d == 1)
        return 1;
    if (g == 0 && c == 1 && d == 0)
        return 1;
    if (g == 0 && c == 0 && d == 1)
        return 1;
    return 0;
}
```

On voit dans l'exemple précédent qu'il y a 8 cas de figure ; on peut les numéroter de 0 à 7. En fait, on peut considérer l'entier $4g + 2c + d$ pour associer à un tel triplet ce numéro. Par exemple, la configuration $(1, 1, 0)$ correspond à $4 + 2 = 6$.

Question 7

Écrire une fonction `int regle(int config)` qui prend le triplet sous la forme d'un numéro comme décrit au-dessus et renvoie la nouvelle valeur. On ne fera PAS appel à la fonction `regle_vois` précédente.

Tests :

```c
int attendu[8] = { 0, 1, 1, 1, 1, 0, 0, 0 };
for (int k = 0; k < 8; k++)
{
    assert(regle(k) == attendu[k]);
}
```

Corrigé

```c
int regle(int config)
{
    if (config == 0 || config == 5 || config == 6 || config == 7)
        return 0;
    return 1;
}
```

Question 8

Écrire une fonction `void suivant(char in[], char out[])` qui prend une configuration `in` de cellules SOUS LA FORME D'UNE CHAÎNE (donc terminée par `'\0'` et dont on peut obtenir la taille avec `strlen`), applique la règle précédente sur chacune et stocke la nouvelle configuration dans la chaîne `out` DE MÊME LONGUEUR.

ATTENTION : on utilise à la place de 0 le caractère `' '` et à la place de 1 le caractère `'#'`.

ATTENTION : en dehors du tableau, on considère que les cellules sont à `' '`.

Test :

```c
char in[] =  "    ##   ## ##     #  # # #";
char out[] = "                           "; // de même longueur
suivant(in, out);
assert(strcmp(out, "   ## # ##  # #   ##### # #") == 0);
```

Corrigé

```c
void suivant(char in[], char out[])
{
    int n = strlen(in);
    for(int i = 0; i < n; i++)
    {
        int config = 0;
        if (i > 0 && in[i-1] == '#')
        {
            config += 4;
        }
        if (in[i] == '#')
        {
            config += 2;
        }
        if (i < n-1 && in[i+1] == '#')
        {
            config += 1;
        }
        if (regle(config) == 1)
        {
            out[i] = '#';
        }
        else
        {
            out[i] = ' ';
        }
```

```
    }
}
```

Question 9

Écrire une fonction `void copie_chaine(char tgt[], char src[])` qui prend en entrée deux chaînes `src` et `tgt` de même longueur et copie la chaîne `src` dans la chaîne `tgt`.

Test :

```
char src[] = "toto";
char tgt[] = "abcd";
copie_chaine(tgt, src);
assert(strcmp(src, tgt) == 0);
```

Corrigé

```
void copie_chaine(char tgt[], char src[])
{
    int n = strlen(src);
    for (int i = 0; i < n; i++)
    {
        tgt[i] = src[i];
    }
}
```

Question 10

Écrire une fonction `void itere_regle(char depart[], char temp[], char sortie[], int iterations)` qui applique `iterations` fois la fonction `suivant` en partant de la configuration `depart` et place la configuration finale dans `sortie`. Comme on ne sait pas encore allouer une telle chaîne dynamiquement, on vous donne aussi une chaîne `temp` comme espace de travail.

ATTENTION : les chaînes `depart`, `temp` et `sortie` sont de même longueur.

Test :

```
char init[81];
for(int i = 0; i < 80; i++)
{
    init[i] = ' ';
}
init[80] = '\0';
char temp[81], sortie[81];
strcpy(temp, init);
strcpy(sortie, init);
init[40] = '#';
itere_regle(init, temp, sortie, 30);
assert(strcmp(sortie, "          ##  #    # ###  # ## # #  ##### #  ######  #   # ##   #  #  #
") == 0);
```

Corrigé

```
void itere_regle(char depart[], char temp[], char sortie[], int iterations)
{
    copie_chaine(sortie, depart);
    copie_chaine(temp, depart);
    for (int i = 0; i < iterations; i++)
    {
        suivant(temp, sortie);
```

```
        copie_chaine(temp, sortie);
    }
}
```

Vous pouvez maintenant profiter de l'affichage des 30 premières itérations :

```
char init[81];
for(int i = 0; i < 80; i++)
{
    init[i] = ' ';
}
init[80] = '\0';
init[40] = '#';

char temp[81], sortie[81];
strcpy(temp, init);
strcpy(sortie, init);
for(int i = 0; i < 30; i++)
{
    itere_regle(init, temp, sortie, i);
    printf("%s\n", sortie);
}
```

```
                                        #
                                       ###
                                      ##  #
                                     ## ####
                                    ##  #   #
                                   ## #### ###
                                  ##  #    #  #
                                 ## ####  ######
                                ##  #   ###     #
                               ## #### ##  #   ###
                              ##  #    # #### ##  #
                             ## ####  ## #    # ####
                            ##  #   ###  ##  ## #   #
                           ## #### ##  ### ###  ## ###
                          ##  #    # ###   #  ###  #  #
                         ## ####  ## #  # #####  #######
```

Il se trouve que la règle qu'on vient d'implémenter s'appelle la règle 30. En effet, si on construit l'entier $\sum_{i=0}^{7} \text{regle}(i)\, 2^i$, où $\text{regle}(i)$ a été définie au-dessus, on obtient l'entier :

$$0 \times 128 + 0 \times 64 + 0 \times 32 + 1 \times 16 + 1 \times 8 + 1 \times 4 + 1 \times 2 + 0 \times 1 = 30$$

On va maintenant définir une fonction permettant d'obtenir la règle associée à un nombre entre 0 et 255. Par exemple, comme 111 s'écrit 01101111 en binaire, cela signifie qu'on associe la valeur 0 aux configurations de numéro 7 et 4 (i.e. (g, c, d) valant $(1, 1, 1)$ et $(1, 0, 0)$) et la valeur 1 à toutes les autres.

Question 11

Écrire une fonction `int regle_gen(int gen, int config)` qui renvoie la valeur obtenue en appliquant la règle `gen` sur la configuration de numéro `config`. On pourra utiliser `vers_binaire`, même si, sur 32 bits, beaucoup de valeurs seront à 0.

Tests :

```
assert(regle_gen(30, 2) == 1);
assert(regle_gen(188, 6) == 0);
assert(regle_gen(188, 7) == 1);
assert(regle_gen(188, 2) == 1);
assert(regle_gen(188, 3) == 1);
```

Corrigé

```c
int regle_gen(int gen, int config)
{
    unsigned char r[32];
    vers_binaire(gen, r);
    return r[config];
}
```

Question 12

Écrire une fonction `void suivant_gen(int gen, char in[], char out[])` qui fait comme `suivant` mais en appliquant la règle `gen`.

REMARQUE : AVEC UN COPIER-COLLER de ce que vous avez fait pour `suivant`, ça prend une seconde.

Tests :

```c
char in[] =  "    ##   ## ##     #  # # #";
char out[] = "                           "; // de même longueur
suivant_gen(30, in, out);
assert(strcmp(out, "   ## # ##  # #   ##### # #") == 0);
suivant_gen(188, in, out);
assert(strcmp(out, "    # #  # ## #    ## #####") == 0);
```

Corrigé

```c
void suivant_gen(int gen, char in[], char out[])
{
    int n = strlen(in);
    for(int i = 0; i < n; i++)
    {
        int config = 0;
        if (i > 0 && in[i-1] == '#')
        {
            config += 4;
        }
        if (in[i] == '#')
        {
            config += 2;
        }
        if (i < n-1 && in[i+1] == '#')
        {
            config += 1;
        }
        if (regle_gen(gen, config) == 1)
        {
            out[i] = '#';
        }
        else
        {
            out[i] = ' ';
        }
    }
}
```

Question 13

Écrire une fonction `void itere_regle_gen(int gen, char depart[], char temp[], char sortie[], int iterations)` qui fait comme `itere_regle` mais en appliquant `suivant_gen`.

Tests (avec `init`, `temp` et `sortie` comme dans le test de la question 10) :

```
itere_regle_gen(30, init, temp, sortie, 30);
assert(strcmp(sortie, "          ##  #    # ###  # ## # #  ##### #  ######  #   # ##   #  #  #
") == 0);
itere_regle_gen(188, init, temp, sortie, 30);
assert(strcmp(sortie, "                                          # ### ### ### ### ### ### ### #
") == 0);
```

Corrigé

```
void itere_regle_gen(int gen, char depart[], char temp[], char sortie[], int iterations)
{
    copie_chaine(sortie, depart);
    copie_chaine(temp, depart);
    for (int i = 0; i < iterations; i++)
    {
        suivant_gen(gen, temp, sortie);
        copie_chaine(temp, sortie);
    }
}
```

Dernière étape : savourez en affichant les 256 règles sur 30 itérations.

```
char init[81];
for(int i = 0; i < 80; i++)
{
    init[i] = ' ';
}
init[80] = '\0';
init[40] = '#';
char temp[81], sortie[81];
strcpy(temp, init);
strcpy(sortie, init);

for (int i = 0; i < 256; i++)
{
    printf("Règle %d\n", i);
    for(int j = 0; j < 30; j++)
    {
        itere_regle_gen(i, init, temp, sortie, j);
        printf("%s\n", sortie);
    }
    printf("\n");
}
```

13

TRAVAUX PRATIQUES 13

C avancé

DANS CE CHAPITRE

13.1 Pointeurs, structures
13.2 Listes chaînées

13.1 Pointeurs, structures

13.1.1 Les structures

Présentation

Il est possible de définir ses propres structures en C qui agrègent d'autres types. Ainsi, on pourra définir une structure `point` permettant de représenter des points du plan :

```
struct point {
    double x;
    double y;
};
```

On peut alors définir un élément de cette structure avec le type `struct point` et on peut accéder à ses champs par l'utilisation de `.`. Pour obtenir le point $(3, 4)$ on écrit :

```
struct point p;
p.x = 3;
p.y = 4;
```

On peut passer en paramètre une telle structure et la renvoyer en valeur de retour comme ici :

```
struct point cree_point(double x, double y)
{
    struct point p;
    p.x = x;
    p.y = y;
    return p;
}

void affiche_point(struct point p)
{
    printf("(%f,%f)\n", p.x, p.y);
}

struct point milieu(struct point a, struct point b)
{
    struct point m;
    m.x = (a.x + b.x)/2;
```

```
    m.y = (a.y + b.y)/2;
    return m;
}

// exemple, dans main
struct point p1 = cree_point(0, 10);
struct point p2 = cree_point(6, 4);
affiche_point(milieu(p1, p2)); // (3.000000,7.000000)
```

Lors de ces passages et de ces retours, la structure est **copiée** champ par champ, comme le serait un `int`.

On remarque que c'est un peu pénible d'écrire `struct point` partout ; il est possible d'utiliser un `typedef` pour définir un type synonyme. Si jamais on a une structure `struct t_s`, on peut écrire `typedef struct t_s t;` pour pouvoir utiliser `t` à la place. On peut même avoir le même nom, comme ici :

```
typedef struct point point;

point p;
p.x = 2;
p.y = 3;
affiche_point(p);
```

Quand on sépare en `.c` et `.h`, il faut mettre les types et les déclarations (ou prototypes) dans le `.h` et les définitions dans le `.c`. Par exemple, on pourrait écrire un fichier `point.h` contenant

```
#ifndef POINT_H
#define POINT_H

struct point {
    double x;
    double y;
};

typedef struct point point;

point cree_point(double x, double y);

#endif
```

puis un fichier `point.c` contenant

```
#include "point.h"

point cree_point(double x, double y)
{
    point p;
    p.x = x;
    p.y = y;
    return p;
}
```

> **Remarque 13.1** Les trois lignes `#ifndef POINT_H`, `#define POINT_H` et `#endif` sont une **garde d'inclusion** : si le fichier est inclus deux fois dans un même `.c` (par exemple directement et via un autre `.h` qui l'inclut), son contenu n'est pris en compte que la première fois. Sans elle, la structure serait définie deux fois, ce qui est une erreur de compilation.

Dans les questions qui suivent, on vous demande d'écrire dans des fichiers. Votre programme sera donc réparti sur plusieurs fichiers. On suppose que votre fonction `main` ainsi que les tests de bon fonctionnement de vos fonctions se trouvent dans le fichier `main.c`. Pour compiler la prochaine question, on pourra donc écrire :

```
gcc -o main main.c vecteur.c -lm
```

Note : le `-lm` est nécessaire pour les fonctions mathématiques.

Attention : si on compile avec plusieurs fichiers `.c` comme ici, il faut placer les bibliothèques à la fin.

Sur le modèle précédent, on va définir une structure `vecteur` qui contient les trois coordonnées d'un vecteur dans l'espace sous la forme de nombres `double`.

> **Remarque 13.2** Pour travailler avec plusieurs fichiers, le plus simple est de créer un répertoire `tp_structures` sur votre ordinateur où vous avez accès à un compilateur `C` et à l'invite de commande. Vous placez vos fichiers dans ce répertoire et vous compilez.
>
> Cependant, si vous n'avez pas installé ça ou si vous êtes sur l'ordinateur de l'établissement, vous pouvez créer des fichiers différents sur `repl.it`.

Question 1

Écrire un fichier `vecteur.h` qui contient la déclaration de la structure et le `typedef` correspondant. Ce fichier contiendra également tous les prototypes des fonctions définies dans `vecteur.c`.

Rappel : le prototype, c'est la ligne comme au-dessus dans `point.h` avec le nom de la fonction et les types de retour et de ses paramètres.

> **Indication** Il s'agit juste de copier-coller ce qui a été fait dans `point.h` en rajoutant `double z` et en changeant le nom de la structure.

Corrigé

```c
#ifndef VECTEUR_H
#define VECTEUR_H

struct vecteur {
    double x;
    double y;
    double z;
};

typedef struct vecteur vecteur;

vecteur vecteur_cree(double x, double y, double z);
vecteur vecteur_nul(void);
double vecteur_norme(vecteur v);
vecteur vecteur_ajoute(vecteur v1, vecteur v2);

#endif
```

En C, une fonction sans paramètre se déclare avec `(void)` : avant C23, `vecteur vecteur_nul();` signifie « nombre de paramètres non précisé ».

Question 2

Écrire un fichier `vecteur.c` qui contient les fonctions suivantes :

1. `vecteur vecteur_cree(double x, double y, double z);` qui renvoie un nouveau vecteur ;
2. `vecteur vecteur_nul(void);` qui renvoie le vecteur nul ;
3. `double vecteur_norme(vecteur v);` qui renvoie la norme du vecteur `v`.

   Note : la fonction `sqrt` se trouve dans `math.h` et il faut compiler en rajoutant le paramètre `-lm` ;
4. `vecteur vecteur_ajoute(vecteur v1, vecteur v2);` qui renvoie un nouveau vecteur, somme des vecteurs `v1` et `v2`.

Indication On rappelle qu'on peut renvoyer une valeur `vecteur` directement et que tout va bien se passer. En fait, on va avoir des copies automatiques.

Corrigé

```c
#include <math.h>
#include "vecteur.h"

vecteur vecteur_cree(double x, double y, double z)
{
    vecteur v;

    v.x = x;
    v.y = y;
    v.z = z;

    return v;
}

vecteur vecteur_nul(void)
{
    return vecteur_cree(0., 0., 0.);
}

double vecteur_norme(vecteur v)
{
    return sqrt(v.x*v.x + v.y*v.y + v.z*v.z);
}

vecteur vecteur_ajoute(vecteur v1, vecteur v2)
{
    return vecteur_cree(
        v1.x+v2.x,
        v1.y+v2.y,
        v1.z+v2.z);
}
```

On va maintenant définir des pavés droits ; on parle d'**axis aligned bounding boxes** en anglais, ce qui est communément abrégé en **AABB**. Un pavé droit est l'extension naturelle à l'espace des rectangles alignés sur les axes ; on peut les définir par les coordonnées $(x_{min}, y_{min}, z_{min})$ et $(x_{max}, y_{max}, z_{max})$ et ce sont les points de coordonnées (x, y, z) où $x_{min} \leq x \leq x_{max}$, $y_{min} \leq y \leq y_{max}$ et $z_{min} \leq z \leq z_{max}$.

On peut ainsi définir la structure :

```c
struct aabb {
    vecteur v_min;
    vecteur v_max;
};
```

Question 3

Définir sur le modèle précédent des fichiers `aabb.h` et `aabb.c`, avec :

1. `aabb aabb_create(vecteur v_min, vecteur v_max);` qui crée une `aabb` ;
2. `aabb aabb_union(aabb b1, aabb b2);` qui renvoie la plus petite boîte contenant les deux boîtes.

Indication Pour 2, il faudra prendre les maximums des maximums et les minimums des minimums. Faites un dessin.

Corrigé

`aabb.h` :

```c
#ifndef AABB_H
#define AABB_H

#include "vecteur.h"

struct aabb {
    vecteur v_min;
    vecteur v_max;
};

typedef struct aabb aabb;

aabb aabb_create(vecteur v_min, vecteur v_max);
aabb aabb_union(aabb b1, aabb b2);

#endif
```

aabb.c :

```c
#include "aabb.h"

aabb aabb_create(vecteur v_min, vecteur v_max)
{
    aabb box;
    box.v_min = v_min;
    box.v_max = v_max;
    return box;
}

static double min(double a, double b)
{
    if (a < b)
        return a;
    else
        return b;
}

static double max(double a, double b)
{
    if (a > b)
        return a;
    else
        return b;
}

aabb aabb_union(aabb b1, aabb b2)
{
    return aabb_create(
        vecteur_cree(
            min(b1.v_min.x, b2.v_min.x),
            min(b1.v_min.y, b2.v_min.y),
            min(b1.v_min.z, b2.v_min.z)),
        vecteur_cree(
            max(b1.v_max.x, b2.v_max.x),
            max(b1.v_max.y, b2.v_max.y),
            max(b1.v_max.z, b2.v_max.z)));
}
```

Le mot-clé `static` rend `min` et `max` locales au fichier `aabb.c` : elles n'entrent pas en conflit avec d'autres fonctions de même nom dans le programme. Le programme se compile avec `gcc -o main main.c vecteur.c aabb.c -lm`.

Au sens strict, la réunion de deux pavés n'est pas un pavé : on renvoie ici la plus petite boîte englobante, ce qui est l'usage (en lancer de rayons par exemple).

Remarque 13.3 On pourrait continuer en faisant la différence, qui est complexe car elle va créer plusieurs boîtes. Cela pose pas mal de questions sur le prototype d'une telle fonction. On pourrait par exemple définir :

```
aabb *aabb_difference(aabb b1, aabb b2, int *nb_boites);
```

qui renvoie un tableau alloué dynamiquement de boîtes et place dans l'entier `nb_boites`, passé par pointeur, le nombre de boîtes.

Structure plus riche

La structure suivante permet de définir une structure `eleve` permettant de contenir un nom, qui est une chaîne de longueur au plus 9, et un âge :

```
struct eleve {
    char nom[10]; // prend donc 10 octets en mémoire
    int age; // prend 4 octets sur 64 bits
}; // donc prend au moins 14 octets
typedef struct eleve eleve;
```

Remarque 13.4 En pratique, `sizeof(eleve)` vaut 16 et pas 14 : le compilateur ajoute du remplissage (*padding*) pour que le champ `age` commence à une adresse multiple de 4.

On aura besoin de comparer deux chaînes ; on utilise pour cela

```
int strncmp(const char *s1, const char *s2, size_t n);
```

depuis `string.h`, qui permet de comparer au plus `n` caractères pour décider si les chaînes sont égales et renvoie 0 à ce moment-là :

- `strncmp("toto", "toti", 4);` ne renvoie pas 0 ;
- `strncmp("toto", "toto", 5);` renvoie 0 (pas de problème de dépassement, la comparaison s'arrête au `'\0'`) ;
- `strncmp("toto", "toti", 3);` renvoie 0.

Le qualificatif `const` présent ici signifie que `strncmp` ne modifiera pas les chaînes (ce qui est assez rassurant pour une fonction de comparaison).

Ici, on va donc utiliser `10` pour `n` pour ne pas sortir du `nom`.

On va considérer une classe qui est un tableau d'`eleve` et on pourra directement accéder aux champs des cases avec `t[i].nom` ou `t[i].age`.

Question 4

Écrire une fonction :

```
/*
 * age : renvoie l'âge d'un élève dans une classe en fonction de son nom
 *       s'il n'est pas présent dans la classe, renvoie -1
 *
 * classe est un tableau d'eleve
 * nb_eleves est le nombre d'éléments du tableau
 * nom est le nom de l'élève dont on cherche l'âge
 */
int age(eleve classe[], int nb_eleves, char *nom);
```

Indication Il s'agit juste de parcourir le tableau et de faire le test avec `strncmp`. C'est exactement comme la `recherche` d'un élément dans un tableau.

Du code pour tester votre fonction :

```c
eleve classe[4];
strcpy(classe[0].nom, "Riri");
classe[0].age = 9;
strcpy(classe[1].nom, "Fifi");
classe[1].age = 10;
strcpy(classe[2].nom, "Loulou");
classe[2].age = 11; // le premier qui dit que ce sont des triplés a un devoir supplémentaire
strcpy(classe[3].nom, "Donald");
classe[3].age =  42; // il est en 43/2

if (age(classe, 4, "Donald") != 42)
    printf("Erreur 1\n");
if (age(classe, 3, "Donald") != -1)
    printf("Erreur 2\n");
if (age(classe, 4, "Loulou") != 11)
    printf("Erreur 3\n");
```

Corrigé

```c
int age(eleve classe[], int nb_eleves, char *nom)
{
    for(int i = 0; i < nb_eleves; i++) {
        if (strncmp(classe[i].nom, nom, 10) == 0)
            return classe[i].age;
    }

    return -1;
}
```

13.1.2 Retours d'arguments avec des pointeurs

Question 5

On a vu jusqu'ici comment renvoyer le minimum d'un tableau, comment renvoyer son indice, mais pas encore comment renvoyer les deux en même temps. Pour cela, on va écrire une fonction qui renvoie le minimum directement mais utilise un pointeur pour renvoyer son indice (voir le passage par adresse). On suppose que `nb` ≥ 1.

```c
int cherche_minimum(int *t, int nb, int *indice);
```

Remarque 13.5 Ici `*` est utilisé de deux façons différentes :

- dans `int *t`, `t` est l'adresse de la première case d'un tableau d'entiers (un tableau passé en paramètre est converti en un tel pointeur, voir Pointeurs et tableaux) ;
- dans `int *indice`, il s'agit d'un passage par adresse de la variable `indice` de l'appelant.

C'est vrai qu'il peut être plus clair d'utiliser `int t[]` dans ce cadre-là pour distinguer les deux usages : pour un paramètre, les deux écritures sont équivalentes.

Indication On rappelle qu'il suffit d'écrire `*indice = 4;` pour changer cette valeur.

Exemple d'utilisation :

```c
int t[] = { 1, 3, 0, 2, 4 };
int indice;
int min = cherche_minimum(t, 5, &indice); // on passe un pointeur sur indice
printf("Le minimum est %d et il est à l'indice %d\n", min, indice);
```

Corrigé

```c
int cherche_minimum(int *t, int nb, int *indice)
{
    int mini = 0;

    for(int i = 1; i < nb; i++) {
        if(t[i] < t[mini])
            mini = i;
    }

    *indice = mini;
    return t[mini];
}
```

Question 6

Pour la fonction suivante, on vous demande de calculer, pour un tableau `t` contenant au moins deux éléments, le couple (i, j) où $i < j$ et pour lequel $|\, t[i] - t[j] \,|$ est minimal. C'est-à-dire la plus petite distance entre deux éléments d'indices distincts dans le tableau. On pourra utiliser `int abs(int x);` pour calculer $|\, x \,|$ (qui se trouve dans `stdlib.h`).

```c
/*
 * t : tableau d'entiers
 * nb : nombre d'éléments du tableau
 *
 * i, j : valeurs de retour passées par pointeurs
 *
 * Renvoie dans (i,j) le couple tel que |t[i]-t[j]| soit minimal
 * et i < j. En cas de plusieurs couples valides, renvoie le couple
 * dont la première coordonnée est minimale.
 */
void plus_petite_distance(int *t, int nb, int *i, int *j);
```

Indication C'est un calcul de minimum mais avec deux indices au lieu d'un. C'est sûrement plus clair ici de placer la distance minimale courante dans une variable plutôt que de la calculer à la volée.

Pour les indices du minimum courant, vous avez le choix entre utiliser `*i` et `*j` ou passer par des variables locales et attendre la fin de la fonction pour affecter `*i` et `*j`. La deuxième version est sûrement plus claire.

```c
// Exemple de test (il faut #include <assert.h>)
int t[] = { 5, 2, 7, 10 };
int i, j;
plus_petite_distance(t, 4, &i, &j);
assert( i == 0 && j == 2 );
```

Corrigé

```c
void plus_petite_distance(int *t, int nb, int *i, int *j)
{
    int mini = 0;
    int minj = 1;

    for(int k = 0; k < nb; k++) {
        for(int l = k+1; l < nb; l++) {
            if(abs(t[k]-t[l]) < abs(t[mini]-t[minj])) {
                mini = k;
                minj = l;
            }
        }
    }

    *i = mini;
```

```
    *j = minj;
}
```

Les couples sont parcourus dans l'ordre lexicographique et la comparaison est stricte : en cas d'égalité, on garde le premier couple rencontré, de première coordonnée minimale (puis de seconde coordonnée minimale). Cette solution est en $O(n^2)$; en triant une copie du tableau, on pourrait trouver la plus petite distance en $O(n \log n)$.

13.1.3 Tableaux redimensionnables

On va utiliser ici une structure pour emballer les tableaux, qui permet de transmettre directement le tableau et sa taille :

```
struct array_t {
    int *elements;
    int nelements;
};
typedef struct array_t *array;
```

Le pointeur `elements` va être alloué dynamiquement. Ici, `array` est un pointeur sur la structure. C'est très important car cela va permettre de modifier les valeurs `elements` ou `nelements` dans les fonctions.

Si on a un `array a`, c'est un pointeur `struct array_t *a`, donc pour accéder à ses membres, il faut écrire par exemple `(*a).elements`. Ce n'est pas très agréable et comme ce cas est très présent, il y a un raccourci : `a->elements`.

Remarque 13.6 En résumé :

- `x.v` si `x` est une valeur de type `struct t` ;
- `x->v` si `x` est une valeur de type `struct t *`.

Note : compilez avec `-g -fsanitize=address` pour être sûr de ne pas faire n'importe quoi avec la mémoire (voir Erreurs classiques).

Question 7

Écrire une fonction

```
array array_alloc(int n);
```

qui alloue et renvoie un tableau de `n` entiers.

Indication Il faut faire deux `malloc` ici (un pour le `array` et un pour son champ `elements`), et il faut bien faire attention à ne pas oublier de changer la valeur du champ `nelements` aussi.

Corrigé

```
array array_alloc(int n)
{
    array t = malloc(sizeof(struct array_t));
    t->elements = malloc(n * sizeof(int));
    t->nelements = n;
    return t;
}
```

Attention à `sizeof(struct array_t)` : `sizeof(array)` ne serait que la taille d'un pointeur.

Question 8

Écrire une fonction

```
void array_free(array t);
```

qui libère le tableau d'éléments.

Indication Attention à bien libérer à la fois `t->elements` et `t` lui-même.

Corrigé

```
void array_free(array t)
{
    free(t->elements);
    free(t);
}
```

L'ordre compte : après `free(t)`, on n'a plus le droit de lire `t->elements`.

Question 9

Écrire une fonction

```
int array_get(array t, int i);
```

qui renvoie l'entier à l'indice `i` si c'est un indice valide et fait une erreur, en déclenchant un `assert`, sinon.

Indication Il faut inclure `assert.h` pour avoir `assert`.

Corrigé

```
int array_get(array t, int i)
{
    assert(0 <= i && i < t->nelements);
    return t->elements[i];
}
```

Question 10

Écrire une fonction

```
void array_set(array t, int i, int v);
```

qui place la valeur `v` à l'indice `i`. On gérera les erreurs comme dans la fonction précédente.

Corrigé

```
void array_set(array t, int i, int v)
{
    assert(0 <= i && i < t->nelements);
    t->elements[i] = v;
}
```

Question 11

Écrire une fonction

```
void array_append(array t, int v);
```

qui va :

- allouer un tableau `nt` de `t->nelements + 1` éléments ;
- recopier `t->elements` dans `nt->elements` (avec une boucle `for`) ;
- placer `v` à la fin du nouveau tableau ;
- libérer `t->elements` ;
- changer `t->elements` et `t->nelements` avec les valeurs dans `nt` ;
- libérer `nt` (attention, ça ne va pas libérer `nt->elements` et c'est ce qu'on veut).

Corrigé

```
void array_append(array t, int v)
{
    array nt = array_alloc(t->nelements + 1);
    for(int i = 0; i < t->nelements; i++) {
        nt->elements[i] = t->elements[i];
    }
    nt->elements[t->nelements] = v;
    free(t->elements);
    t->elements = nt->elements;
    t->nelements += 1;
    free(nt);
}
```

Question 12

Écrire une fonction

```
int array_pop(array t);
```

qui va, pour un tableau `t` non vide :

- allouer un tableau `nt` de `t->nelements - 1` éléments ;
- recopier `t->elements`, sauf son dernier élément, dans ce tableau ;
- stocker le dernier élément de `t->elements` ;
- libérer `t->elements` ;
- changer `t->elements` et `t->nelements` ;
- libérer `nt` (attention, ça ne va pas libérer `nt->elements` et c'est ce qu'on veut) ;
- renvoyer le dernier élément.

Corrigé

```
int array_pop(array t)
{
    assert(t->nelements > 0);
    array nt = array_alloc(t->nelements - 1);
    for(int i = 0; i < t->nelements-1; i++) {
        nt->elements[i] = t->elements[i];
    }
    int v = t->elements[t->nelements-1];
    free(t->elements);
    t->elements = nt->elements;
    t->nelements -= 1;
    free(nt);
    return v;
}
```

Exemple d'utilisation, qui affiche `3`, `2` puis `1` :

```
array l = array_alloc(0);
array_append(l, 1);
array_append(l, 2);
array_append(l, 3);
for(int i = 0; i < 3; i++)
    printf("%d\n", array_pop(l));
array_free(l);
```

Chaque ajout ou retrait recopie tout le tableau : n ajouts successifs coûtent $\sum_{k=0}^{n-1} k = O(n^2)$ copies. C'est l'objet de la partie suivante.

Tableaux dynamiques plus efficaces

Question 13

Reprendre ce qui a été fait en `OCaml` avec des tableaux dynamiques qui doublent de taille en cas d'ajout.

Corrigé

On distingue le nombre de cases utilisées (`nelements`) du nombre de cases allouées (`capacite`). On ne réalloue que lorsque le tableau est plein, en doublant sa capacité : n ajouts coûtent alors $O(n)$ copies au total, donc $O(1)$ en coût amorti par ajout.

```
struct tabdyn_t {
    int *elements;
    int nelements; // nombre de cases utilisées
    int capacite;  // nombre de cases allouées
};
typedef struct tabdyn_t *tabdyn;

tabdyn tabdyn_vide(void)
{
    tabdyn t = malloc(sizeof(struct tabdyn_t));
    t->capacite = 1;
    t->nelements = 0;
    t->elements = malloc(t->capacite * sizeof(int));
    return t;
}

void tabdyn_free(tabdyn t)
{
    free(t->elements);
    free(t);
}

int tabdyn_get(tabdyn t, int i)
{
    assert(0 <= i && i < t->nelements);
    return t->elements[i];
}

void tabdyn_set(tabdyn t, int i, int v)
{
    assert(0 <= i && i < t->nelements);
    t->elements[i] = v;
}

void tabdyn_append(tabdyn t, int v)
{
    if (t->nelements == t->capacite)
    {
        // plein : on double la capacité
        int *nouveau = malloc(2 * t->capacite * sizeof(int));
        for (int i = 0; i < t->nelements; i++)
```

```
        {
            nouveau[i] = t->elements[i];
        }
        free(t->elements);
        t->elements = nouveau;
        t->capacite = 2 * t->capacite;
    }
    t->elements[t->nelements] = v;
    t->nelements += 1;
}

int tabdyn_pop(tabdyn t)
{
    assert(t->nelements > 0);
    t->nelements -= 1;
    return t->elements[t->nelements];
}
```

La fonction `realloc` de `stdlib.h` permet d'agrandir une zone allouée en évitant d'écrire la boucle de recopie.

13.2 Listes chaînées

13.2.1 Listes simplement chaînées

Remarque 13.7 Il faut compiler tout ce TP avec `-fsanitize=address -Wall` pour ne pas rater d'erreurs de gestion de la mémoire. Il faudra aussi effectuer des tests de vos fonctions.

Voici un exemple de script de test :

```
int main(void)
{
    liste *l = liste_vide();
    liste_affiche(l);
    liste_ajout_en_tete(l, 3);
    liste_ajout_en_tete(l, 2);
    liste_ajout_en_tete(l, 1);
    liste_affiche(l);
    liste_ajout_en_fin(l, 5);
    liste_affiche(l);
    maillon *m = liste_cherche_maillon(l, 3);
    liste_ajout_maillon_apres(l, m, 4);
    liste_affiche(l);
    liste_suppr_maillon(l, m);
    liste_affiche(l);
    liste_detruire(l);
}
```

qui doit afficher `[]`, `[1,2,3]`, `[1,2,3,5]`, `[1,2,3,4,5]` puis `[1,2,4,5]`. Pensez aussi à tester les cas limites : liste vide, liste à un seul maillon, opérations sur le premier ou le dernier maillon.

On pourra également faire des tests dans PythonTutor pour s'assurer que la mémoire correspond à ce que l'on a en tête.

On va définir des fonctions sur les listes chaînées en utilisant le type suivant :

```
struct maillon {
    int valeur;
    struct maillon *suivant;
};
typedef struct maillon maillon;

struct liste {
```

```
    maillon *premier;
    maillon *dernier;
};
typedef struct liste liste;
```

Remarque 13.8 **Invariants à maintenir**

- Dans un maillon, `suivant` est soit `NULL` si c'est le dernier maillon de la chaîne, soit un pointeur vers le maillon qui le suit.
- Si la liste est vide, alors `premier` et `dernier` valent `NULL`. Sinon, ils pointent respectivement sur le premier et le dernier maillon de la chaîne.

Chaque fonction qui modifie la liste doit rétablir ces invariants, en particulier le champ `dernier`, qu'on oublie facilement.

Question 14

Écrire une fonction de prototype

```
liste *liste_creer(maillon *premier, maillon *dernier);
```

qui alloue et renvoie une liste dont les maillons sont donnés en arguments.

Question 15

Écrire une fonction de prototype

```
liste *liste_vide(void);
```

qui alloue et renvoie **une** liste vide.

Ici, on dit **une** car chaque appel va renvoyer une nouvelle liste vide.

Corrigé (questions 14 et 15)

```
liste *liste_creer(maillon *premier, maillon *dernier)
{
    liste *l = malloc(sizeof(liste));
    l->premier = premier;
    l->dernier = dernier;
    return l;
}

liste *liste_vide(void)
{
    return liste_creer(NULL, NULL);
}
```

Question 16

Écrire une fonction de prototype

```
void liste_affiche(liste *l);
```

qui affiche le contenu de la liste sous la forme `[1,2,3]`.

Note : on pourra utiliser `putchar(c);` pour afficher le caractère `c`.

Corrigé

```c
void liste_affiche(liste *l)
{
    putchar('[');
    maillon *m = l->premier;
    while(m != NULL)
    {
        printf("%d", m->valeur);
        if(m->suivant != NULL)
            putchar(',');
        m = m->suivant;
    }
    putchar(']');
    putchar('\n');
}
```

Question 17

Écrire une fonction de prototype

```c
maillon *maillon_creer(int valeur, maillon *suivant);
```

qui alloue et initialise un nouveau maillon.

Corrigé

```c
maillon *maillon_creer(int valeur, maillon *suivant)
{
    maillon *m = malloc(sizeof(maillon));
    m->valeur = valeur;
    m->suivant = suivant;
    return m;
}
```

Question 18

Écrire une fonction de prototype

```c
void maillon_detruire(maillon *m);
```

qui détruit, c'est-à-dire libère, la mémoire associée à un maillon.

Écrire une fonction de prototype

```c
void chaine_detruire(maillon *m);
```

qui détruit la chaîne de maillons accessible depuis le maillon `m`.

En déduire une fonction de prototype

```c
void liste_detruire(liste *l);
```

qui détruit la chaîne pointée par une liste, ainsi que la liste elle-même.

Corrigé

```c
void maillon_detruire(maillon *m)
{
    free(m);
}
```

```c
// en impératif
void chaine_detruire(maillon *m)
{
    while(m != NULL)
    {
        maillon *suivant = m->suivant; // à lire AVANT de libérer m
        maillon_detruire(m);
        m = suivant;
    }
}

// en récursif
void chaine_detruire_rec(maillon *m)
{
    if (m != NULL) {
        chaine_detruire_rec(m->suivant);
        maillon_detruire(m);
    }
}

void liste_detruire(liste *l)
{
    chaine_detruire(l->premier);
    free(l);
}
```

La version récursive utilise une profondeur de pile égale à la longueur de la chaîne : sur une très longue liste, elle peut provoquer un dépassement de pile, contrairement à la version impérative.

Question 19

Écrire une fonction de prototype

```c
int liste_longueur(liste *l);
```

qui renvoie la longueur de la liste `l`.

Corrigé

```c
int liste_longueur(liste *l)
{
    int longueur = 0;
    maillon *m = l->premier;
    while(m != NULL)
    {
        longueur += 1;
        m = m->suivant;
    }
    return longueur;
}
```

Question 20

Écrire des fonctions de prototype

```c
int liste_tete(liste *l);
liste *liste_queue(liste *l);
```

qui renvoient respectivement la tête et la queue d'une liste non vide. **Attention** : ici, contrairement au type des listes d'`OCaml`, le maillon suivant n'est pas une liste.

Corrigé

```c
int liste_tete(liste *l)
{
    assert(l->premier != NULL);
    return l->premier->valeur;
}

liste *liste_queue(liste *l)
{
    assert(l->premier != NULL);
    if (l->premier->suivant == NULL) // la queue est vide
        return liste_vide();
    return liste_creer(l->premier->suivant, l->dernier);
}
```

La queue **partage** ses maillons avec `l` : on la libère avec un simple `free`, et pas avec `liste_detruire`, sinon les maillons seraient libérés deux fois. De plus, toute modification de `l` peut rendre la queue incohérente. Il faut traiter à part la liste à un élément pour respecter l'invariant de la liste vide.

Question 21

Écrire une fonction de prototype

```c
void liste_ajout_en_tete(liste *l, int valeur);
```

qui ajoute un maillon en tête de la liste pointée par `l` en $O(1)$. On fera attention au cas où `l` pointe la liste vide.

Corrigé

```c
void liste_ajout_en_tete(liste *l, int valeur)
{
    l->premier = maillon_creer(valeur, l->premier);
    if(l->dernier == NULL) // la liste était vide
        l->dernier = l->premier;
}
```

Question 22

Écrire une fonction de prototype

```c
void liste_suppr_en_tete(liste *l);
```

qui supprime un maillon en tête de la liste `l` supposée non vide en $O(1)$.

On fera attention au cas où on supprime l'unique maillon de la liste.

Corrigé

```c
void liste_suppr_en_tete(liste *l)
{
    assert(l->premier != NULL);
    maillon *m = l->premier;
    l->premier = m->suivant;
    maillon_detruire(m); // on libère la mémoire
    if(l->premier == NULL) // on a vidé la liste
        l->dernier = NULL;
}
```

Question 23

Écrire une fonction de prototype

```
void liste_ajout_en_fin(liste *l, int valeur);
```

qui ajoute un maillon en fin de la liste `l` en $O(1)$.

Corrigé

```
void liste_ajout_en_fin(liste *l, int valeur)
{
    maillon *m = maillon_creer(valeur, NULL);
    if(l->dernier == NULL) // la liste était vide
        l->premier = m;
    else
        l->dernier->suivant = m;
    l->dernier = m;
}
```

Dans le cas de la liste vide, il n'y a pas d'ancien dernier maillon à chaîner : écrire `l->dernier->suivant` serait un déréférencement de `NULL`.

Question 24

Écrire une fonction de prototype

```
maillon *liste_cherche_maillon(liste *l, int valeur);
```

qui renvoie un pointeur sur le premier maillon de valeur `valeur` ou renvoie `NULL` s'il n'y en a pas.

Corrigé

```
maillon *liste_cherche_maillon(liste *l, int valeur)
{
    maillon *m = l->premier;
    while(m != NULL && m->valeur != valeur)
    {
        m = m->suivant;
    }
    return m;
}
```

L'évaluation paresseuse de `&&` garantit qu'on ne lit pas `m->valeur` quand `m` vaut `NULL`.

Question 25

Écrire une fonction de prototype

```
void liste_ajout_maillon_apres(liste *l, maillon *m, int valeur);
```

qui insère en $O(1)$ un maillon après le maillon `m`. On fera en sorte que l'insertion soit valide même si `m` est le dernier maillon de la liste.

Corrigé

```
void liste_ajout_maillon_apres(liste *l, maillon *m, int valeur)
{
    m->suivant = maillon_creer(valeur, m->suivant);
    if (m == l->dernier)
```

```
        l->dernier = m->suivant;
}
```

Remarque 13.9 Dans les fonctions suivantes, on suppose qu'on va aller chercher naïvement le maillon précédent pour faire les chaînages.

Cependant, on peut se contenter de faire des échanges de valeurs pour ne pas avoir ce problème.

Question 26

Écrire une fonction de prototype

```
void liste_ajout_maillon_avant(liste *l, maillon *m, int valeur);
```

qui ajoute un maillon avant `m`. Dans le cas où `m` est le premier maillon, l'insertion sera en $O(1)$.

Note : on peut rendre cela en $O(1)$ dans tous les cas en tenant compte de la remarque précédente.

Corrigé

```
// version naïve où on recherche le maillon précédent
void liste_ajout_maillon_avant(liste *l, maillon *m, int valeur)
{
    if (m == l->premier)
        liste_ajout_en_tete(l, valeur);
    else
    {
        maillon *prec = l->premier;
        while(prec->suivant != m)
        {
            prec = prec->suivant;
        }
        prec->suivant = maillon_creer(valeur, m);
    }
}

// version efficace : on insère une copie de m après m,
// puis on place la nouvelle valeur dans m
void liste_ajout_maillon_avant_O1(liste *l, maillon *m, int valeur)
{
    maillon *m2 = maillon_creer(m->valeur, m->suivant);
    m->valeur = valeur;
    m->suivant = m2;
    if (m == l->dernier)
        l->dernier = m2;
}
```

Attention, dans la version en $O(1)$, le pointeur `m` désigne ensuite le maillon contenant `valeur`, et non plus l'ancienne valeur de `m` : d'éventuels pointeurs conservés ailleurs sur ce maillon changent de sens.

Question 27

Écrire une fonction de prototype

```
void liste_suppr_maillon(liste *l, maillon *m);
```

qui supprime un maillon.

Dans le cas où `m` est le premier maillon, la suppression sera en $O(1)$.

Note : on peut rendre cela en $O(1)$ **sauf dans un cas**, en tenant compte de la remarque précédente.

Corrigé

```c
// version naïve avec une recherche
void liste_suppr_maillon(liste *l, maillon *m)
{
    if (m == l->premier)
        liste_suppr_en_tete(l);
    else
    {
        maillon *prec = l->premier;
        while(prec->suivant != m)
        {
            prec = prec->suivant;
        }
        prec->suivant = m->suivant;
        if (m == l->dernier)
            l->dernier = prec;
        maillon_detruire(m);
    }
}

// version efficace : on recopie le maillon suivant dans m
// et on supprime ce suivant
void liste_suppr_maillon_O1(liste *l, maillon *m)
{
    if (m == l->dernier)
    {
        // seul cas en O(n) : il faut trouver le précédent
        liste_suppr_maillon(l, m);
    } else {
        maillon *m2 = m->suivant;
        m->valeur = m2->valeur;
        m->suivant = m2->suivant;
        if (m2 == l->dernier)
            l->dernier = m;
        maillon_detruire(m2);
    }
}
```

Le cas exceptionnel est celui du dernier maillon, qui n'a pas de suivant à recopier.

13.2.2 Listes doublement chaînées

Pour les listes doublement chaînées, on va adapter le type précédent en rajoutant juste un pointeur `precedent` dans les maillons :

```c
struct maillon {
    int valeur;
    struct maillon *suivant;
    struct maillon *precedent;
};
typedef struct maillon maillon;

struct liste {
    maillon *premier;
    maillon *dernier;
};
typedef struct liste liste;
```

Remarque 13.10 Invariants à maintenir

- Dans un maillon, `suivant` est soit `NULL` si c'est le dernier maillon de la chaîne, soit un pointeur vers le maillon qui le suit.
- Dans un maillon, `precedent` est soit `NULL` si c'est le premier maillon de la chaîne, soit un pointeur vers le maillon qui le précède.

- Si la liste est vide, alors `premier` et `dernier` valent `NULL`. Sinon, ils pointent respectivement sur le premier et le dernier maillon de la chaîne.

Question 28

Reprendre les questions précédentes avec ce nouveau type.

Corrigé

Peu de modifications à faire. On fait en sorte que `maillon_creer` effectue le double chaînage avec ses voisins.

```
liste *liste_creer(maillon *premier, maillon *dernier)
{
    liste *l = malloc(sizeof(liste));
    l->premier = premier;
    l->dernier = dernier;
    return l;
}

liste *liste_vide(void)
{
    return liste_creer(NULL, NULL);
}

void liste_affiche(liste *l)
{
    putchar('[');
    maillon *m = l->premier;
    while(m != NULL)
    {
        printf("%d", m->valeur);
        if(m->suivant != NULL)
            putchar(',');
        m = m->suivant;
    }
    putchar(']');
    putchar(' ');
    // pour tester, on remonte dans l'autre sens
    putchar('[');
    m = l->dernier;
    while(m != NULL)
    {
        printf("%d", m->valeur);
        if(m->precedent != NULL)
            putchar(',');
        m = m->precedent;
    }
    putchar(']');
    putchar('\n');
}

// crée un maillon et le chaîne à precedent et à suivant (s'ils existent)
maillon *maillon_creer(int valeur,
        maillon *precedent, maillon *suivant)
{
    maillon *m = malloc(sizeof(maillon));
    m->valeur = valeur;
    m->suivant = suivant;
    m->precedent = precedent;
    if (precedent != NULL)
        precedent->suivant = m;
    if (suivant != NULL)
        suivant->precedent = m;
    return m;
}
```

```c
void maillon_detruire(maillon *m)
{
    free(m);
}

void chaine_detruire(maillon *m)
{
    while(m != NULL)
    {
        maillon *suivant = m->suivant;
        maillon_detruire(m);
        m = suivant;
    }
}

void liste_detruire(liste *l)
{
    chaine_detruire(l->premier);
    free(l);
}

int liste_longueur(liste *l)
{
    int longueur = 0;
    maillon *m = l->premier;
    while(m != NULL)
    {
        longueur += 1;
        m = m->suivant;
    }
    return longueur;
}

int liste_tete(liste *l)
{
    assert(l->premier != NULL);
    return l->premier->valeur;
}

void liste_ajout_en_tete(liste *l, int valeur)
{
    l->premier = maillon_creer(valeur, NULL, l->premier);
    if(l->dernier == NULL) // la liste était vide
        l->dernier = l->premier;
}

void liste_suppr_en_tete(liste *l)
{
    assert(l->premier != NULL);
    maillon *m = l->premier;
    l->premier = m->suivant;
    maillon_detruire(m); // on libère la mémoire
    if(l->premier == NULL) // on a vidé la liste
        l->dernier = NULL;
    else
        l->premier->precedent = NULL;
}

void liste_ajout_en_fin(liste *l, int valeur)
{
    // maillon_creer chaîne déjà l'ancien dernier au nouveau
    l->dernier = maillon_creer(valeur, l->dernier, NULL);
    if(l->premier == NULL) // la liste était vide
        l->premier = l->dernier;
}
```

```c
maillon *liste_cherche_maillon(liste *l, int valeur)
{
    maillon *m = l->premier;
    while(m != NULL && m->valeur != valeur)
    {
        m = m->suivant;
    }
    return m;
}

void liste_ajout_maillon_apres(liste *l, maillon *m, int valeur)
{
    maillon *mn = maillon_creer(valeur, m, m->suivant);
    if (m == l->dernier)
        l->dernier = mn;
}
```

La fonction `liste_queue` n'a pas été reprise : une queue qui partage ses maillons avec la liste aurait un premier maillon dont le champ `precedent` n'est pas `NULL`, ce qui viole l'invariant.

Question 29

Écrire une fonction de prototype

```c
void liste_suppr_en_fin(liste *l);
```

qui supprime en $O(1)$ le dernier maillon d'une liste non vide.

Corrigé

```c
void liste_suppr_en_fin(liste *l)
{
    assert(l->dernier != NULL);
    maillon *fin = l->dernier;
    l->dernier = fin->precedent;

    if (l->dernier == NULL)
        l->premier = NULL;
    else
        l->dernier->suivant = NULL;

    maillon_detruire(fin);
}
```

Question 30

Écrire une fonction de prototype

```c
void liste_ajout_maillon_avant(liste *l, maillon *m, int valeur);
```

qui ajoute un maillon avant `m` en $O(1)$.

Corrigé

```c
void liste_ajout_maillon_avant(liste *l, maillon *m, int valeur)
{
    maillon *mn = maillon_creer(valeur, m->precedent, m);
    if(m == l->premier)
        l->premier = mn;
}
```

Question 31

Écrire une fonction de prototype

```
void liste_suppr_maillon(liste *l, maillon *m);
```

qui supprime en $O(1)$ le maillon `m`.

Corrigé

```
void liste_suppr_maillon(liste *l, maillon *m)
{
    if (m == l->premier)
        liste_suppr_en_tete(l);
    else if (m == l->dernier)
        liste_suppr_en_fin(l);
    else
    {
        m->precedent->suivant = m->suivant;
        m->suivant->precedent = m->precedent;
        maillon_detruire(m);
    }
}
```

14

TRAVAUX PRATIQUES 14

DS3 - Examen pratique de C

DANS CE CHAPITRE

14.1 Une somme
14.2 Une suite
14.3 Tri par comptage
14.4 Chaine
14.5 Fusion de deux tableaux

L'examen peut être fait sur `Repl` ou sur votre ordinateur avec un compilateur `C`. Le rendu se fera par mail à l'adresse `exam@de-falco.fr` sous la forme d'un fichier `C` **nommé `nom.c`**, avec votre nom (et pas le prénom) sans accents, quitte à abréger mais avec le bon préfixe, contenant vos fonctions dans l'ordre des questions.

On vous donne des exemples de tests pour vérifier vos fonctions : ils ne sont pas exhaustifs, vos fonctions seront vérifiées sur un jeu de tests plus grand.

Vous pouvez laisser vos tests dans le rendu, mais vos fonctions doivent **nécessairement** avoir le prototype donné dans le sujet et pas un autre !

14.1 Une somme

Écrire une fonction de prototype

```c
int somme(int n);
```

qui renvoie pour $n \geq 0$, la valeur de $S_n = \sum_{k=1}^{n} k(k+2)$.

Réponse

```c
int somme(int n) {
    int s = 0;
    for(int k = 1; k <= n; k++) {
        s = s + k * (k+2);
    }
    return s;
}
```

14.2 Une suite

Écrire une fonction de prototype

```c
int suite(int n);
```

qui renvoie u_n pour $n \geq 0$, où $(u_n)_{n\in\mathbb{N}}$ est la suite définie par :

$$u_0 = 1, \quad \forall n \geq 0, u_{n+1} = \begin{cases} 2u_n + 1 & \text{si } u_n \text{ pair} \\ 3 - u_n & \text{sinon} \end{cases}$$

On a ainsi $u_{10} = 45$.

Réponse

```c
int suite(int n)
{
    int u = 1;
    for(int i = 1; i <= n; i++)
    {
        if (u % 2 == 0)
            u = 2*u+1;
        else
            u = 3-u;
    }
    return u;
}
```

Écrire une fonction de prototype

```c
int compteneg(int n);
```

qui renvoie le nombre de u_k pour $0 \leq k \leq n$ tels que $u_k < 0$.

Pour $n = 10$, cette fonction renvoie 4.

Réponse

```c
int compteneg(int n)
{
    int c = 0;
    for(int i = 0; i <= n; i++)
        if (suite(i) < 0)
            c++;
    return c;
}
```

14.3 Tri par comptage

On considère ici une méthode de tri dite **par comptage**. On va supposer que les tableaux qu'on cherche à trier ne contiennent que des entiers entre 0 et 99.

L'idée est alors, dans un tableau de taille 100, de compter le nombre d'occurrences de chacun des entiers.

Ensuite, on peut se servir de ce tableau d'occurrences pour en déduire le tableau trié.

On demande à ce que la complexité soit linéaire en le nombre d'éléments, c'est-à-dire que le nombre d'opérations élémentaires soit proportionnel au nombre d'éléments.

14.3.1 Occurrences

Écrire une fonction de prototype

```c
void occurrences(int t[], int n, int occ[]);
```

qui prend en entrée `t` un tableau de `n` entiers appartenant à $[\![0, 99]\!]$, `occ` un tableau de 100 entiers dont les valeurs sont non initialisées et qu'on utilisera uniquement pour écrire dedans.

Cette fonction doit alors placer, pour tout $i \in [\![0, 99]\!]$, dans `occ[i]` le nombre d'occurrences de l'entier `i` dans `t`.

```c
bool test_occ()
{
```

```
    int t[] = { 0, 1, 2, 3, 0, 1, 2, 1 };
    int occ[100];
    occurrences(t, 8, occ);
    int res[] = { 2, 3, 2, 1 };
    bool valide = true;
    for(int i = 0; i < 100; i++)
    {
        if(i < 4)
            valide = valide && occ[i] == res[i];
        else
            valide = valide && occ[i] == 0;
    }
    return valide;
}
```

Réponse

```
void occurrences(int t[], int n, int occ[])
{
    for(int i = 0; i < 100; i++)
        occ[i] = 0;
    for(int i = 0; i < n; i++)
        occ[t[i]]++;
}
```

14.3.2 Le tri

L'idée pour trier un tableau `1, 3, 0, 1, 3, 1` c'est de se rendre compte qu'il contient :

- 1 fois la valeur 0 et donc que le tableau trié doit commencer par `0`
- 3 fois la valeur 1 et donc qu'ensuite le tableau trié doit se poursuivre par `0, 1, 1, 1`
- 0 fois la valeur 2 donc on ne la met pas dans le tableau trié
- 2 fois la valeur 3 et donc on les rajoute `0, 1, 1, 1, 3, 3`
- 0 fois de toute autre valeur, donc c'est fini le tableau trié est `0, 1, 1, 1, 3, 3`

Écrire une fonction de prototype

```
void comptage(int t[], int n);
```

qui trie **en place** le tableau `t` ainsi. Note, il faudra surement définir des tableaux intermédiaires, mais cela ne pose pas de problèmes si on connait leur taille à la compilation.

Par contre, il est impossible d'écrire `int temp[n];` car `n` est une variable. Si vous voulez faire ça, c'est que vous faites fausse route.

Note cette fonction va naturellement appeler la fonction `occurrences`.

```
bool test_comptage()
{
    int t[] = { 0, 1, 2, 3, 0, 1, 2, 1 };
    comptage(t, 8);
    int res[] = { 0, 0, 1, 1, 1, 2, 2, 3 };
    bool valide = true;
    for(int i = 0; i < 8; i++)
    {
        valide = valide && t[i] == res[i];
    }
    return valide;
}
```

Réponse

```c
void comptage(int t[], int n)
{
    int occ[100];
    occurrences(t, n, occ);
    int a = 0;
    for(int i = 0; i < 100; i++)
    {
        for(int k = 0; k < occ[i]; k++)
        {
            t[a] = i;
            a++;
        }
    }
}
```

14.4 Chaine

Rappel on peut utiliser `strlen(s)` pour calculer la longueur de `s`. Cette fonction est déclarée dans `string.h`.

14.4.1 Plus longue plage minuscule

Écrire une fonction de prototype

```c
int plagemin(char *s);
```

qui renvoie le plus grand entier `k` tel qu'il existe `k` caractères consécutifs en minuscule dans `s`.

Exemple :

- `s = "totoTtotou"` on renvoie 5 qui correspond à `"totou"`
- `s = "AtA"` on renvoie 1
- `s = "AAB"` on renvoie 0

```c
bool test_plagemin()
{
    bool valide = true;
    valide = valide && plagemin("totoTtotou") == 5;
    valide = valide && plagemin("AtA") == 1;
    valide = valide && plagemin("AAB") == 0;
    return valide;
}
```

Réponse

```c
int plagemin(char *s)
{
    int k = 0;
    bool plage = false;
    int cur = 0;
    for(int i = 0; i < strlen(s); i++)
    {
        char c = s[i];
        bool minus = 'a' <= c && c <= 'z';
        if (minus)
        {
            if(!plage)
            {
                cur = 1;
                plage = true;
            } else
                cur++;
        } else {
            if(plage) {
                if (cur > k)
                    k = cur;
```

```
                plage = false;
            }
        }
    }
    if(plage && cur > k)
        k = cur;
    return k;
}
```

14.4.2 Recherche linéaire

Écrire une fonction de prototype

```
int recherche(char *s, char* motif);
```

qui prend en entrée deux chaines, une chaine `motif` correspondant à une chaine que l'on va chercher dans une autre chaine `s`, et qui renvoie l'indice de la première occurrence de `motif` dans `s` quand il est présent et `-1` sinon.

Exemples :

- `s = "tototatota", motif = "tota"` le motif est présent deux fois à l'indice 2 et à l'indice 6, on renvoie donc la valeur 2.
- `s = "tototatota", motif = "tata"` le motif n'est pas présent on renvoie −1
- `s = "toto", motif = "tototo"` pas présent non plus donc −1
- `s = "test", motif = ""` motif présent tout de suite, on renvoie 0

```
bool test_recherche()
{
    bool valide = true;
    valide = valide && recherche("tototatota", "tota") == 2;
    valide = valide && recherche("tototatota", "tata") == -1;
    valide = valide && recherche("toto", "tototo") == -1;
    valide = valide && recherche("test", "") == 0;
    return valide;
}
```

Réponse

```
int recherche(char *s, char *motif)
{
    int n = strlen(s);
    int p = strlen(motif);
    for(int i = 0; i <= n-p; i++)
    {
        bool flag = true;
        for(int j = 0; j < p; j++)
            flag = flag && s[i+j] ==motif[j];
        if (flag)
            return i;
    }
    return -1;
}
```

14.5 Fusion de deux tableaux

On souhaite ici écrire une fonction `fusion` de prototype

```
void fusion(int t1[], int n1, int t2[], int n2, int tf[]);
```

qui permet, étant donnés deux tableaux triés `t1` et `t2` de construire un tableau `tf` contenant les éléments de `t1` et de `t2` mais globalement triés.

Ainsi, si `t1` contient `0, 1, 4, 6` et `t2` contient `1, 2, 3, 4, 5`, le tableau `tf` contiendra `0, 1, 1, 2, 3, 4, 4, 5, 6`.

On suppose que `t1` contient `n1` éléments, `t2` contient `n2` éléments et `tf` contient `n1+n2` éléments dont la valeur est indéterminée (on considère qu'on a juste le droit d'écrire dans `tf`).

```
bool test_fusion()
{
    int t1[] = { 0, 1, 4, 6 };
    int t2[] = { 1, 2, 3, 4, 5 };
    int t[9];
    fusion(t1, 4, t2, 5, t);
    int tres[] = {0, 1, 1, 2, 3, 4, 4, 5, 6};
    bool valide = true;
    for(int i = 0; i < 9; i++) {
        if (tres[i] != t[i]) {
            valide = false;
        }
    }
    return valide;
}
```

Réponse

```
void fusion(int t1[], int n1, int t2[], int n2, int tf[])
{
    int a = 0;
    int b = 0;
    for(int i = 0; i < n1 + n2; i++)
    {
        if (a < n1 && (b == n2 || t1[a] < t2[b]))
        {
            tf[i] = t1[a];
            a++;
        }
        else
        {
            tf[i] = t2[b];
            b++;
        }
    }
}
```

PARTIE III

Structures de données

CHAPITRE 15

Structures de données abstraites et implémentations

DANS CE CHAPITRE

15.1 Introduction
15.2 Structure de données abstraite
15.3 Implémentation

15.1 Introduction

On va étudier dans ce chapitre la notion de structures de données. Une structure de données est une manière de stocker des données et d'interagir avec elles. On peut regrouper de grandes classes de structures de données en faisant abstraction de la manière dont les données sont stockées en se concentrant uniquement sur les interactions possibles. Cela permet de définir la notion de structure de données abstraite.

Dans un second temps, on s'intéresse à la notion d'implémentation d'une structure de données abstraite qui correspond à un choix concret de stockage et donc à une réalisation de l'interface attendue.

15.2 Structure de données abstraite

Définition 15.1

On appelle *structure de données abstraite* la donnée d'un type `t` et d'une interface, c'est-à-dire :

- d'une liste d'opérations portant sur `t`, avec leur signature (types des arguments et du résultat) ;
- d'une *spécification* décrivant le comportement attendu de chacune de ces opérations, sans faire référence à la manière dont les données sont stockées.

Le type `t` est alors dit *abstrait* : on ne le manipule qu'au travers des opérations de l'interface.

Exemple 15.2

La structure de pile, étudiée dans Piles et files, a pour interface les opérations `creer_pile`, `est_vide`, `empiler` et `depiler`. Sa spécification dit que :

- `est_vide(creer_pile())` est vrai ;
- après `empiler(p, x)`, la pile `p` n'est pas vide et un appel à `depiler(p)` renvoie `x` et redonne à `p` son état d'avant l'ajout ;
- `depiler` sur une pile vide est une erreur.

C'est exactement ce qui permet d'écrire la fonction `longueur_pile` de ce chapitre sans savoir comment la pile est représentée.

On peut écrire une interface directement dans un langage de programmation. En OCaml, c'est le rôle d'un fichier d'interface `.mli`, qui donne les types des opérations sans dévoiler la définition du type. En C, c'est le rôle d'un fichier d'en-tête `.h`, qui déclare la structure sans ses champs et les prototypes des fonctions.

OCaml

```
type 'a pile

val cree_pile : unit -> 'a pile
val est_vide : 'a pile -> bool
val empile : 'a pile -> 'a -> unit
val depile : 'a pile -> 'a
```

C

```
#include <stdbool.h>

struct pile;
typedef struct pile pile;

pile *pile_creer();
void pile_detruire(pile *p);
bool est_vide(pile *p);
void empiler(pile *p, int x);
int depiler(pile *p);
```

Remarque 15.3 La signature ne suffit pas à définir la structure. Une file a exactement la même signature qu'une pile, seule la spécification les distingue : l'ordre de retrait est LIFO pour une pile et FIFO pour une file.

D'autres structures abstraites seront étudiées dans les chapitres suivants :

- les piles, les files et les files de priorité dans Piles et files ;
- les ensembles finis et les dictionnaires dans Dictionnaires et ensembles finis ;
- les collections d'ensembles disjoints dans Union-Find ;
- les graphes dans Graphes, qu'on manipule au travers d'opérations comme l'ajout d'une arête, le test d'existence d'une arête ou le parcours des voisins d'un sommet.

15.3 Implémentation

15.3.1 Abstraction et implémentation

Définition 15.4

Une *implémentation* d'une structure de données abstraite est la donnée d'une représentation concrète du type `t` et d'un code pour chacune des opérations de l'interface, qui respecte la spécification.

Une même structure abstraite admet en général plusieurs implémentations. Par exemple, une pile peut être réalisée par un tableau borné et un indice de première case libre, ou par une liste chaînée. Une file peut être réalisée par un tableau circulaire ou par deux listes. Un graphe peut être représenté par une matrice d'adjacence ou par des listes d'adjacence.

L'intérêt de cette séparation est double :

- le code qui **utilise** la structure ne dépend que de l'interface. On peut donc changer d'implémentation sans le modifier ;
- le code qui **implémente** la structure peut maintenir des propriétés internes, appelées *invariants*, en étant sûr que personne d'autre ne modifie la représentation. Par exemple, pour la pile dans un tableau, l'invariant est que les éléments de la pile sont exactement ceux des cases d'indice 0 à `fond - 1`.

Remarque 15.5 En C, rien n'empêche réellement un utilisateur qui connaît la définition de `struct pile` d'aller modifier le champ `fond`. Le fichier d'en-tête ne déclarant pas les champs, un programme qui n'inclut

que lui ne peut pas le faire. En OCaml, un type déclaré abstrait dans un `.mli` est réellement inaccessible en dehors du module.

15.3.2 Structures mutables et immuables

Il y a deux grandes manières de concevoir l'interface d'une structure.

- Dans une structure **mutable**, les opérations modifient la structure qu'on leur passe en argument, par effet de bord. C'est le cas des piles ci-dessus : `empile p x` ne renvoie rien et modifie `p`.
- Dans une structure **immuable**, ou *persistante*, les opérations ne modifient jamais leur argument mais renvoient une nouvelle structure. L'ancienne version reste utilisable.

Cela se voit dès la signature : la version immuable d'une pile renvoie la nouvelle pile.

OCaml

```
type 'a pile = 'a list

let pile_vide = []

let est_vide p = p = []

let empile p x = x :: p

let depile p =
  match p with
  | [] -> failwith "Erreur : dépiler sur une pile vide"
  | x :: q -> (x, q)
```

Ici, `empile p x` est de type `'a pile` et non `unit`, et `depile` renvoie le sommet **et** la pile restante. Après `let p' = empile p 3`, on dispose à la fois de `p` et de `p'`, sans aucune copie : la liste `p` est partagée.

Remarque 15.6 Les structures immuables sont naturelles en OCaml et se prêtent bien aux raisonnements par récurrence. Les structures mutables sont naturelles en C et permettent souvent des implémentations plus efficaces, par exemple à base de tableaux. Les deux points de vue cohabitent : on rencontrera dans Dictionnaires et ensembles finis une implémentation immuable (les arbres binaires de recherche) et une implémentation mutable (les tables de hachage) de la même structure abstraite.

15.3.3 Complexité des opérations

La spécification d'une structure abstraite ne dit rien du coût des opérations : c'est le choix de l'implémentation qui le détermine. En pratique, ce coût est pourtant ce qui guide le choix d'une implémentation, et on le précise toujours avec elle.

Structure abstraite	Implémentation	Complexité des opérations
Pile	tableau borné, liste chaînée	$O(1)$
File	tableau circulaire	$O(1)$
File	deux listes	$O(1)$ amorti
Dictionnaire	arbre binaire de recherche de hauteur h	$O(h)$
Dictionnaire	table de hachage	$O(1)$ en moyenne
Union-find	forêt avec union par rang et compression	$O(\alpha(n))$ amorti

Remarque 15.7 Il faut toujours préciser de quelle complexité on parle. Pour une file réalisée avec deux listes, un retrait isolé peut coûter $O(n)$, mais une suite de m opérations coûte $O(m)$: on parle de complexité *amortie*. Pour une table de hachage, un accès peut coûter $O(n)$ dans le pire cas, et c'est seulement *en moyenne*, sous une hypothèse sur la fonction de hachage, qu'il est en $O(1)$.

CHAPITRE 16

Séquences et leurs implémentations, tableaux, listes chaînées

DANS CE CHAPITRE

16.1 Structure abstraite séquence ou liste
16.2 Implémentations concrètes des listes chaînées
16.3 Travaux pratiques

16.1 Structure abstraite séquence ou liste

La structure abstraite *liste*, ou également *séquence* dans des contextes, comme en `OCaml`, où le terme `liste` fait référence aux listes chaînées, est la structure la plus simple pour stocker des données.

Une *séquence* d'éléments de type `t` est un type `S(t)` dont les éléments représentent des valeurs du type `t` rangées séquentiellement dans un ordre, de la première à la dernière valeur. La vision logique la plus proche de cela est d'imaginer des cases ayant un indice, en commençant en général à l'indice 0, et contenant des valeurs.

✍ On renvoie au chapitre Structures de données abstraites et implémentations pour les notions générales d'interface, d'implémentation et de structure mutable ou immuable.

16.2 Implémentations concrètes des listes chaînées

16.2.1 En C

Représentation et type

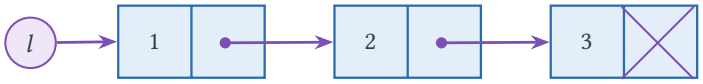
Il existe de nombreuses possibilités d'implémentation des listes chaînées en `C`. On présente ici les opérations autour d'une implémentation et on discutera ensuite des alternatives.

Dans les limites du programme, on ne présente que des listes permettant de contenir un même type, ici des entiers.

Une liste est ainsi un pointeur sur un maillon et un maillon est un couple `(valeur, suivant)` représenté dans une `struct` où `suivant` pointe vers le prochain maillon de la chaîne. Le pointeur nul, de valeur `NULL`, permet ainsi de représenter la liste vide.

```c
struct maillon {
    int valeur;
    struct maillon *suivant;
};
typedef struct maillon maillon;
typedef maillon *liste;
```

On représentera graphiquement le pointeur nul par une croix et les maillons par des blocs contenant une valeur et un pointeur. Ainsi, le dernier maillon de la liste contient une croix. La liste `l` correspondant à la valeur qu'on pourrait noter `[1,2,3]` sera représentée ainsi :



Constructeur

On parle de constructeur pour des fonctions qui permettent d'allouer et d'initialiser une valeur d'une structure de données. Ici, comme les listes sont des pointeurs sur des maillons, il s'agit uniquement de créer un maillon. Pour cela, on va utiliser `malloc` pour allouer dynamiquement un nouveau maillon.

```
maillon *maillon_creer(int valeur, maillon *suivant)
{
    maillon *m = malloc(sizeof(maillon));
    m->valeur = valeur;
    m->suivant = suivant;
    return m;
}
```

En fait, ce constructeur pourrait être découpé en deux parties : l'allocation qui va se charger de récupérer un emplacement mémoire pour le maillon et l'initialisation qui va attribuer des valeurs.

```
maillon *maillon_allouer(void)
{
    return malloc(sizeof(maillon));
}

void maillon_initialiser(maillon *m, int valeur, maillon *suivant)
{
    m->valeur = valeur;
    m->suivant = suivant;
}

maillon *maillon_creer(int valeur, maillon *suivant)
{
    maillon *m = maillon_allouer();
    maillon_initialiser(m, valeur, suivant);
    return m;
}
```

On peut alors commencer à créer des listes en enchaînant les maillons :

```
maillon *m1 = maillon_creer(1, NULL); // pas de suivant, c'est le dernier maillon
maillon *m2 = maillon_creer(2, m1);
maillon *m3 = maillon_creer(3, m2);
liste l = m3; // la liste pointe sur le premier maillon
```

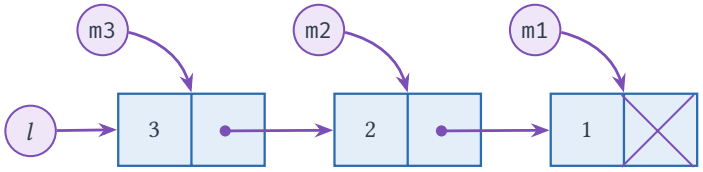


Figure 16.1 — Les trois maillons créés et la liste *l* qui pointe sur le premier.

Comme le pointeur sur le premier maillon suffit ici, on aurait pu directement écrire :

```
liste l = maillon_creer(3,
            maillon_creer(2,
                maillon_creer(1, NULL)));
```

Destructeur

Pour détruire un maillon, il suffit de libérer l'espace qu'on lui a attribué.

```
void maillon_detruire(maillon *m)
{
    free(m);
}
```

Pour détruire une liste, on va par contre avoir besoin de parcourir l'ensemble des maillons qui la constituent. Comme pour les autres parcours, on a alors deux choix :

- **parcours récursif** : on a un cas de base quand la liste est vide et dans le cas général, un éventuel appel récursif sur le pointeur `suivant`.

```
void liste_detruire(liste l)
{
    if (l != NULL)
    {
        liste_detruire(l->suivant);
        maillon_detruire(l); // Attention à l'ordre pour l->suivant
    }
}
```

- **parcours impératif** : on boucle tant que la liste est non nulle. On fait ici attention à ne pas accéder à `->suivant` après avoir libéré le maillon.

```
void liste_detruire(liste l)
{
    while (l != NULL)
    {
        liste suivante = l->suivant;
        maillon_detruire(l);
        l = suivante;
    }
}
```

Ajout et suppression en tête

Pour ajouter ou supprimer un maillon en tête de la liste, on va avoir besoin de modifier le pointeur vers le premier maillon. Pour cela, on a deux approches possibles :

- passer un pointeur vers la liste elle-même, c'est-à-dire un pointeur sur un pointeur sur un maillon, ce qui en `C` aura le type `maillon **`, qui s'écrit aussi `liste *`. On aura alors le prototype :

```
void liste_ajout_en_tete(liste *l, int valeur);
```

- renvoyer un pointeur vers le nouveau premier maillon. On aura alors le prototype :

```
liste liste_ajout_en_tete(liste l, int valeur);
```

On présente ici les versions renvoyant une nouvelle liste :

```
liste liste_ajout_en_tete(liste l, int valeur)
{
    maillon *m = maillon_creer(valeur, l);
    return m;
}
```

Si `l` est la liste précédente contenant `3,2,1` et qu'on ajoute `4` en tête, on va donc directement créer un nouveau maillon et renvoyer un pointeur vers celui-ci.

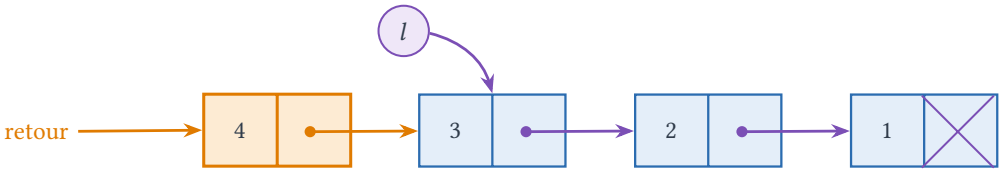


Figure 16.2 — Ajout de 4 en tête : le nouveau maillon (en orange) pointe sur l'ancienne tête.

```
liste liste_suppr_en_tete(liste l)
{
    assert(l != NULL);
    liste queue = l->suivant;
    maillon_detruire(l); // Attention, on détruit juste le maillon, pas la liste
    return queue;
}
```

Si `l` est la liste précédente contenant `3,2,1` et qu'on supprime le maillon de tête, on va renvoyer un pointeur sur le second maillon. **Attention**, ici le pointeur initial `l` est devenu invalide.

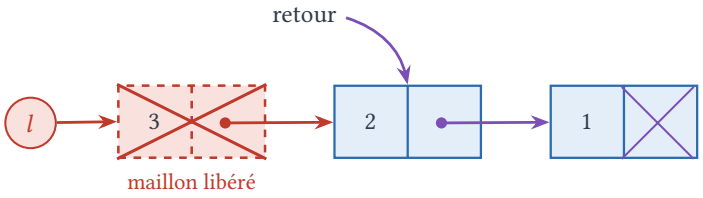


Figure 16.3 — Suppression en tête : le maillon est libéré et le pointeur *l* devient invalide.

Longueur de la liste

On présente ici le calcul de la longueur comme exemple de parcours de la liste. C'est encore très proche du parcours effectué dans le destructeur.

```
int liste_longueur(liste l)
{
    int longueur = 0;
    while(l != NULL)
    {
        longueur = longueur + 1;
        l = l->suivant;
    }
    return longueur;
}
```

Accès au n-ième maillon de la liste

On effectue un parcours similaire pour accéder au n-ième maillon (les indices commençant à 0).

```
maillon *liste_nieme(liste l, int n)
{
    while(n > 0)
    {
        assert(l != NULL);
        l = l->suivant;
        n = n-1;
    }
    assert(l != NULL);
    return l;
}
```

Toujours avec un programme similaire, on peut chercher un maillon avec sa valeur :

```
maillon *liste_recherche(liste l, int x)
{
```

```
    while(l != NULL && l->valeur != x)
    {
        l = l->suivant;
    }
    return l;
}
```

Ici, pas besoin d'`assert` : en cas d'échec de la recherche, on renvoie un pointeur nul.

Ajout/suppression ailleurs qu'en tête

Pour ajouter ou supprimer ailleurs qu'en tête, il est nécessaire de pouvoir repérer précisément un maillon. Pour cela, on peut le faire :

- par son indice, celui utilisé dans `liste_nieme` ;
- par sa valeur, avec une recherche ;
- ou encore directement par un pointeur sur le maillon.

Une fois le maillon repéré, on peut procéder comme pour un ajout/suppression en tête, cependant il va falloir reconnecter le pointeur `suivant` du maillon précédent. Pour cela deux choix :

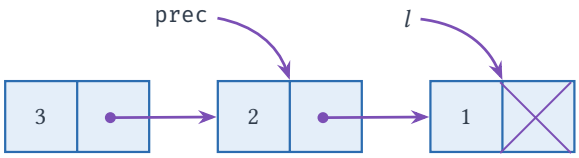
- soit on considère qu'on ajoute après, auquel cas on dispose du précédent ;
- soit on considère qu'on ajoute avant ou qu'on supprime, et pour cela on effectue une boucle pour déterminer le maillon précédent.

```
void liste_ajout_apres(liste l, maillon *m, int x)
{
    m->suivant = liste_ajout_en_tete(m->suivant, x);
}

void liste_ajout_avant(liste l, maillon *m, int x)
{
    liste prec = NULL;
    while(l != m)
    {
        assert(l != NULL);
        prec = l;
        l = l->suivant;
    }
    assert(prec != NULL); // m n'est pas le premier maillon
    prec->suivant = liste_ajout_en_tete(m, x);
}
```


> **Remarque 16.1** La fonction `liste_ajout_avant` ne peut pas ajouter avant le **premier** maillon : il faudrait pour cela modifier la liste `l` elle-même, ce qui n'est pas possible avec ce prototype. Dans ce cas, on utilise `liste_ajout_en_tete`. Le paramètre `l` de `liste_ajout_apres` n'est pas utilisé : il n'est là que pour garder la même interface.

Si on reprend la liste `l` contenant `3,2,1` et qu'on souhaite insérer la valeur `4` avant la valeur `1`, on va passer un pointeur `m` vers le maillon contenant `1` et effectuer un parcours jusqu'à avoir `prec` et `l` dans la configuration :



On dispose alors des pointeurs permettant de réaliser l'ajout :

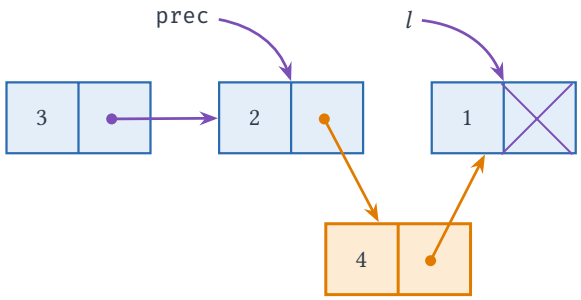


Figure 16.4 — Le nouveau maillon 4 (en orange) est inséré entre prec et *l*.

De même pour la suppression d'un maillon qui n'est pas en tête :

```
void liste_suppr_non_en_tete(liste l, maillon *m)
{
    liste prec = NULL;
    while(l != m)
    {
        assert(l != NULL);
        prec = l;
        l = l->suivant;
    }
    assert(prec != NULL); // m n'est pas le premier maillon
    prec->suivant = liste_suppr_en_tete(m);
}
```

En fait, lorsqu'on regarde le parcours précédent, on remarque deux points :

- comme on a donné le maillon concerné, le parcours a uniquement pour but de repérer le maillon qui le précède ;
- on pourrait se contenter d'utiliser uniquement un pointeur sur le maillon précédent lors du parcours car le maillon qui le suit est accessible avec `->suivant`.

Autres implémentations

Une autre implémentation standard consiste à *cacher* le pointeur sur le premier maillon, ce qui permet de donner également un pointeur sur le dernier maillon. En effet, le pointeur sur le dernier maillon permet de réaliser un ajout en fin de liste en $O(1)$ car il n'y a pas besoin de parcourir la liste pour faire cet ajout.

On obtient alors un type comme :

```
struct maillon {
    int valeur;
    struct maillon *suivant;
};
typedef struct maillon maillon;

struct liste {
    maillon *premier;
    maillon *dernier;
};
typedef struct liste liste;
```

Cette implémentation est l'occasion de se poser la question sur la répartition entre pile et tas pour les données. On peut légitimement penser que les listes ici ne sont que des couples de pointeurs, et qu'ainsi les passer par copie est léger comparativement à la complexité induite par une allocation sur le tas.

16.2.2 En OCaml

Cas des `'a list`

Tout d'abord, il faut se rendre compte que le type par défaut

```
type 'a list = [] | (::) of 'a * 'a list
```

fait intervenir des maillons et des pointeurs. La différence principale avec C est qu'on ne peut pas changer la valeur des pointeurs.

Type des listes chaînées

Pour retrouver la richesse du type précédent, on peut définir un type comme :

```
type 'a maillon = {
    mutable valeur : 'a;
    mutable suivant : 'a liste
} and 'a liste = Vide | Lien of 'a maillon
```

Ici, le type est une traduction directe du type précédent. On remarque que les types sont mutuellement récursifs car un maillon contient une liste. Le type somme `'a liste` ressemble fortement à un pointeur qui peut être nul ou pointer sur un maillon. On remarque qu'on aurait pu aussi se contenter d'écrire le type suivant :

```
type 'a maillon = {
    mutable valeur : 'a;
    mutable suivant : 'a maillon option
}

type 'a liste = 'a maillon option
```

Mais on va préférer le premier type qui a l'avantage de permettre de bien faire apparaître la structure.

Ajout et suppression en tête

Pour rajouter un maillon en tête, on peut écrire :

```
let cons x l = Lien { valeur = x; suivant = l}
```

On remarque que la fonction est beaucoup plus simple que celle en C car l'allocation est automatique et l'initialisation se fait naturellement dans la syntaxe. Cette fonction renvoie une nouvelle liste, on aurait pu aussi rajouter une référence pour les listes afin de permettre de les rendre modifiables, c'est la même discussion que dans la partie précédente.

La suppression en tête revient à renvoyer la queue de la liste, comme le fait `List.tl` :

```
let tl l  = match l with
    | Vide -> failwith "Liste vide"
    | Lien { valeur = _; suivant = q} -> q
```

On remarque le filtrage imbriqué `| Lien { valeur = t; suivant = q }` qui correspond au filtrage `| t :: q` du type `'a list`.

Exemple de parcours sans modification : calcul de la longueur

Vu les remarques précédentes, il n'est pas étonnant que le parcours d'une liste de manière récursive soit très proche de ce qu'on a déjà pu voir avec les listes de base.

```
let rec longueur l = match l with
    | Vide -> 0
    | Lien { valeur = _; suivant = q } -> 1 + longueur q
```

On en déduit de même une fonction renvoyant un maillon par son indice :

```
let rec nieme l n = match l with
    | Vide -> failwith "Liste vide"
    | Lien m -> if n = 0 then m else nieme m.suivant (n-1)
```

Remarque 16.2 Il est possible de réécrire la fonction précédente en permettant à la fois de donner un nom, ici `m`, au maillon et de faire un filtrage sur ce qu'il contient. Pour cela, on utilise le mot clé `as` en `OCaml` :

```
let rec nieme l n = match l with
    | Vide -> failwith "Liste vide"
    | Lien ({ valeur=_; suivant = q} as m) ->
            if n = 0 then m else nieme q (n-1)
```

Exemple de parcours avec modification : ajout d'un maillon en fin de liste

On va montrer un exemple de modification de liste en rajoutant un maillon en fin d'une liste non vide. Ici, on effectue un parcours jusqu'à tomber sur le dernier maillon auquel on rajoute le nouveau à la suite.

```
let rec ajout_fin l x =
    match l with
    | Vide -> failwith "Liste vide"
    | Lien m ->
        if m.suivant = Vide
        then m.suivant <- Lien { valeur = x; suivant = Vide }
        else ajout_fin m.suivant x
```

Raffinement pour permettre l'ajout à la fin en temps constant

Le programme précédent est à comparer au programme suivant :

```
let rec ajout_fin l x =
    match l with
    | [] -> [x]
    | t::q -> t :: ajout_fin q x
```

On n'a pas l'impression d'avoir vraiment gagné en expressivité ou en efficacité.

Cependant, on peut se dire qu'en changeant le type `'a liste` on peut tirer parti des maillons pour obtenir un ajout en fin de liste en temps constant. Le programme suivant présente une interface permettant de le faire.

```
type 'a maillon = {
    mutable valeur : 'a;
    mutable suivant : 'a maillon_ptr
  }
and 'a maillon_ptr = 'a maillon option
and 'a liste = {
      mutable premier : 'a maillon_ptr;
      mutable dernier : 'a maillon_ptr
  }

let liste_vide () = { premier = None; dernier = None }

let ajout_debut l x =
    l.premier <- Some { valeur = x; suivant = l.premier };
    if l.dernier = None
    then l.dernier <- l.premier

let suppr_debut l =
    match l.premier with
    | None -> failwith "Liste vide"
    | Some { valeur = _; suivant = l' } ->
            l.premier <- l';
            if l' = None
            then l.dernier <- None

let ajout_fin l x =
    let m = { valeur = x; suivant = None } in
    (match l.dernier with
    | Some m' -> m'.suivant <- Some m
    | None -> () );
    l.dernier <- Some m;
```

```
    if l.premier = None
    then l.premier <- l.dernier
```

Quelques remarques sur ce programme :

- comme on change directement `premier` et `dernier`, il est nécessaire de générer une nouvelle liste vide, ce qui est assuré ici par le paramètre `()` ;
- les pointeurs sont représentés par des options dans des champs mutables ;
- afin de préserver l'intégrité des deux pointeurs, on est obligé de gérer les cas où il n'y a qu'un seul maillon : quand on ajoute dans une liste vide, ou quand on supprime l'unique maillon, `premier` et `dernier` doivent être mis à jour ensemble ;
- attention à la priorité du cas de filtrage sur `;` qui oblige à mettre des parenthèses dans `ajout_fin`.

✍ Cette structure réalise une file : on ajoute avec `ajout_fin` et on retire avec `suppr_debut`, chaque opération étant en temps constant (voir le chapitre Piles et files).

16.2.3 Structure de la mémoire

En mémoire, les maillons d'une liste chaînée, comme celle vue en `C`, sont sur le tas et de manière désorganisée. Cela signifie qu'il n'y a aucune raison que deux maillons proches dans une liste soient proches en mémoire.

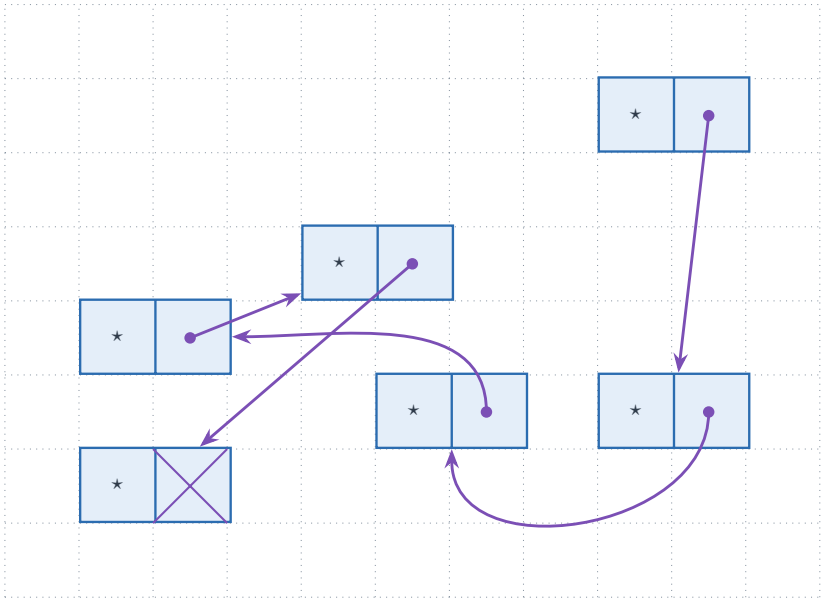


Figure 16.5 — Les maillons d'une liste chaînée sont dispersés dans la mémoire.

Or, les processeurs optimisent la gestion de la mémoire à l'aide d'un cache qui, au lieu de n'accéder qu'à une seule valeur située à une adresse, va charger une zone mémoire autour de cette adresse. Ceci encourage une cohérence spatiale dans l'organisation de la mémoire afin de profiter au maximum de cette mise en cache.

Une stratégie pour réaliser des listes chaînées efficacement peut consister à allouer un tableau de maillons et à gérer ensuite les allocations parmi cette réserve.

16.3 Travaux pratiques

On présente ici un énoncé de travaux pratiques en `C` en lien avec ce chapitre.

16.3.1 Tableaux non statiques et tableaux dynamiques

On va définir ici une structure de données pour gérer des tableaux dont la taille ne sera connue qu'à l'exécution. Pour simplifier, ce seront des tableaux d'entiers `int` mais la méthode s'adapte naturellement pour des tableaux de n'importe quoi.

Type `array`

On définit le type suivant :

```
struct array {
    int *elements;
    unsigned int size;
};
typedef struct array array;
```

Un `array` est donc une structure qui contient :

- un pointeur vers un tableau `elements` d'entiers ;
- un entier indiquant la taille de ce tableau.

Remarque importante : quand on copie un pointeur, on copie uniquement un entier qui est l'adresse pointée. Ainsi, quand on copie un `array`, ce n'est finalement qu'un couple d'entiers qu'on copie. C'est pourquoi, dans la suite, on passe tous les `array` par valeur. C'est-à-dire que pour définir une fonction de recherche d'un élément dans un `array` on va écrire :

```
int search(array t, int x)
{
    for(int i = 0; i < t.size; i++)
    {
        if(t.elements[i] == x)
        {
            return i;
        }
    }

    return -1;
}
```

Quand on va appeler la fonction, on va avoir une copie de l'argument, c'est-à-dire du `array`, mais le tableau `elements` lui sera toujours à la même place. L'alternative serait de passer les valeurs `array` par pointeurs comme dans la variante suivante :

```
int search(array *t, int x)
{
    for(int i = 0; i < t->size; i++)
    {
        if(t->elements[i] == x)
        {
            return i;
        }
    }

    return -1;
}
```

On n'a rien à gagner au surplus de complexité induit par cela. L'unique avantage serait de permettre de modifier les paramètres de la structure passée en argument, mais on réservera ça aux fonctions qui le nécessitent.

Allocation, initialisation, libération

Pour les fonctions suivantes, on fera usage de `malloc` et `free`.

Question 1

Écrire une fonction de prototype `array array_alloc(unsigned int size)` qui alloue un tableau de `size` entiers avec `malloc` et renvoie le `array` pointant dessus.

Conformément à ce qui est écrit au-dessus, on peut créer un `array` en variable locale et le renvoyer avec `return`, ce qui compte c'est la mémoire pointée.

Corrigé

```
array array_alloc(unsigned int size)
{
    array a;
    a.elements = (int *)malloc(sizeof(int) * size);
    a.size = size;
    return a;
}
```

Question 2

Écrire une fonction de prototype `void array_free(array t)` qui libère le tableau pointé par `t` avec `free`. À partir de ce moment, on ne peut plus utiliser cette adresse sans faire d'erreurs.

Corrigé

```
void array_free(array a)
{
    free(a.elements);
}
```

Question 3

Le tableau alloué n'est pas initialisé, écrire une fonction de prototype `array array_make(unsigned int size, int def)` qui alloue un tableau et initialise toutes les valeurs de celui-ci à `def`.

Corrigé

```
array array_make(unsigned int size, int def)
{
    array a = array_alloc(size);
    for(int i = 0; i < size; i++)
    {
        a.elements[i] = def;
    }
    return a;
}
```

Question 4

Il est possible de mesurer le temps pris par un programme à l'aide de la commande `time` :

```
$> time ./main
./main  0.38s user 0.00s system 99% cpu 0.379 total
```

Comparer le temps d'exécution d'un programme qui

- alloue puis libère 100 fois un tableau d'un million d'entiers ;
- alloue *en initialisant à 0* avec `array_make` puis libère 100 fois un tableau d'un million d'entiers.

Que peut-on en conclure ?

Corrigé

On se rend compte que l'allocation est presque instantanée alors que l'initialisation prend un temps linéaire en la taille des données. Ceci est cohérent avec le fait que la mémoire est allouée de manière paresseuse. On s'en rend d'autant plus compte en examinant l'empreinte mémoire des programmes.

Remarque 16.3 Il existe d'autres fonctions que `malloc` comme

```
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
```

qui permettent :

- pour `calloc` d'allouer un tableau de `nmemb` membres qui sont chacun de taille `size` et **d'initialiser les octets à 0** ;
- pour `realloc` de déplacer en mémoire un tableau en changeant sa taille (en plus ou en moins). Si jamais il y a de la place on ne bouge pas le tableau. C'est donc en général plus efficace que de le faire à la main.

Ces deux fonctions sont **hors programme !**

Affichage, copie

Question 5

Écrire une fonction `void array_print(array a)` qui affiche `[a1,a2,...,an]` si `a` contient les éléments de `a1` à `an`.

Remarque : cette fonction vous sera utile pour tester ce que vous avez fait dans la suite.

Corrigé

```
void array_print(array a)
{
    printf("[");
    for(int i = 0; i < a.size; i++)
    {
        printf("%d", a.elements[i]);
        if(i < a.size - 1)
            printf(",");
    }
    printf("]\n");
}
```

Question 6

Écrire une fonction de prototype

```
void array_blit(array src, int src_start,
    array dst, int dst_start, int len)
```

qui copie `len` éléments depuis le tableau `src` à partir de l'indice `src_start` dans le tableau `dst` à partir de l'indice `dst_start`.

Remarque : on a déjà écrit cette fonction en `OCaml`, vous pouvez vous en inspirer.

Corrigé

```
void array_blit(array src, int src_start,
            array dst, int dst_start, int len)
{
    for(int i = 0; i < len; i++)
        dst.elements[i + dst_start] =
            src.elements[i + src_start];
}
```

Implémentation d'une pile non bornée

Question 7

Écrire une fonction de prototype `void array_push(array *a, int x)` qui rajoute `x` à la fin du array `a`, c'est-à-dire :

- on va allouer un nouveau `array b` dont le tableau contient `a->size+1` éléments ;
- on va copier les anciennes valeurs de `*a` dans `b` ;
- écrire `x` ;
- libérer `*a` ;
- remplacer avec `*a = b`.

Remarque : on peut tout à fait écrire `*a = b` pour que `a` ait les mêmes valeurs que `b` en dehors de la fonction.

Rappel : pour une `struct` passée par pointeur, on utilise `->` pour accéder aux champs. Donc `a->elements` et `a->size` ici.

Remarque : faites des tests, par exemple en créant un tableau de taille 0, en empilant les entiers de 1 à 10 et en affichant le résultat. Faites les tests avec `-fsanitize=address`.

Corrigé

```
void array_push(array *a, int x)
{
    array b = array_alloc(a->size+1);
    array_blit(*a, 0, b, 0, a->size);
    b.elements[a->size] = x;
    array_free(*a);
    *a = b;
}

// Note qu'on aurait pu proposer une interface persistante pour push/pop ainsi
// pas de pointeurs mais on renvoie le nouveau array
array array_push_2(array a, int x)
{
    array b = array_alloc(a.size+1);
    array_blit(a, 0, b, 0, a.size);
    b.elements[a.size] = x;
    // pas de raison de faire de free ici de a, c'est
    // a l'appelant de gérer.
    return b;
}
```

Question 8

Écrire une fonction `int array_pop(array *a)` qui retire le dernier élément du tableau `a`, supposé non vide, et renvoie sa valeur. Il faudra donc :

- allouer un nouveau `array b` dont le tableau contient `a->size-1` éléments ;
- copier les anciennes valeurs sauf une ;
- récupérer dans une variable `x` la valeur de la dernière case de `a` (*parce qu'il n'est pas possible d'y accéder après l'étape suivante*) ;
- libérer l'ancien tableau ;
- remplacer avec `*a = b`.

Corrigé

```
int array_pop(array *a)
{
    assert(a->size > 0); // nécessite #include <assert.h>
```

```
    array b = array_alloc(a->size-1);
    array_blit(*a, 0, b, 0, a->size-1);
    int x = a->elements[a->size-1];
    array_free(*a);
    *a = b;
    return x;
}
```


Remarque 16.4 Avec cette implémentation, chaque `array_push` et chaque `array_pop` recopie tout le tableau : il coûte $O(n)$ où n est la taille courante. Empiler n éléments à partir d'un tableau vide coûte donc $O(1 + 2 + \ldots + n) = O(n^2)$. C'est ce qui motive les tableaux dynamiques de la partie suivante.

Tableaux dynamiques

Reprendre toutes les questions précédentes en changeant la structure de données pour permettre de faire des tableaux dynamiques. Il faudra ainsi avoir deux entiers : `p_size` qui contiendra la taille *physique* en mémoire et `l_size` qui contiendra la taille *logique*, c'est-à-dire le nombre d'éléments signifiants stockés.

Quand on a besoin de réallouer la mémoire, on choisira de doubler la taille physique (en passant à une taille 1 si elle était nulle). Le `pop` ne libérera jamais de taille physique.

Corrigé

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

struct array {
    int *elements;
    unsigned int p_size;
    unsigned int l_size;
};

typedef struct array array;

array array_alloc(unsigned int size)
{
    array a;
    a.elements = (int *)malloc(sizeof(int) * size);
    a.p_size = size;
    a.l_size = 0;
    return a;
}

void array_free(array a)
{
    free(a.elements);
}

array array_make(unsigned int size, int def)
{
    array a = array_alloc(size);
    for(int i = 0; i < size; i++)
    {
        a.elements[i] = def;
    }
    a.l_size = size;
    return a;
}

void array_print(array a)
{
    printf("[");
    for(int i = 0; i < a.l_size; i++)
```

```
    {
        printf("%d", a.elements[i]);
        if(i < a.l_size - 1)
            printf(",");
    }
    printf("]\n");
}

void array_blit(array src, int src_start,
            array dst, int dst_start, int len)
{
    for(int i = 0; i < len; i++)
        dst.elements[i + dst_start] =
            src.elements[i + src_start];
}

void array_push(array *a, int x)
{
    if (a->l_size < a->p_size)
    {
        a->elements[a->l_size] = x;
        a->l_size += 1;
    }
    else
    {
        // si le tableau est vide, on passe à une taille physique de 1
        array b = array_alloc(a->p_size == 0 ? 1 : 2*a->p_size);
        array_blit(*a, 0, b, 0, a->l_size);
        b.elements[a->l_size] = x;
        b.l_size = a->l_size + 1;
        array_free(*a);
        *a = b;
    }
}

int array_pop(array *a)
{
    assert(a->l_size > 0);
    a->l_size -= 1;
    return a->elements[a->l_size];
}

int main(void)
{
    array a = array_alloc(0);
    for(int i = 0; i < 10; i++)
        array_push(&a, i);
    for(int i = 0; i < 3; i++)
        printf("Pop %d\n", array_pop(&a));
    array_print(a);
    array_free(a);
}
```

✍ **Remarque 16.5** Un `array_push` qui provoque un redoublement coûte toujours $O(n)$, mais ces redoublements sont rares : pour n ajouts successifs à partir d'un tableau vide, les recopies portent sur $1 + 2 + 4 + \ldots + 2^k < 2n$ éléments (où $2^k < n$ est la dernière taille physique pleine). Le coût total est donc en $O(n)$, soit $O(1)$ *amorti* par ajout. C'est le même argument que pour le redimensionnement des tables de hachage ; la notion de complexité amortie est développée dans le chapitre Complexité amortie.

17

TRAVAUX PRATIQUES 17

Tableaux dynamiques

17.1 Tableaux dynamiques

On va se servir d'un champ mutable pour définir un type `'a dynarray` de tableaux dynamiques, c'est-à-dire de tableaux dont la taille pourra changer en ajoutant des éléments dedans à la manière des `list` de `Python`.

Pour cela, on va d'abord commencer par définir des fonctions sur les `array` (qui existent pour beaucoup dans la bibliothèque standard mais c'est surtout l'occasion de s'entraîner).

17.1.1 Préliminaires : fonctions de manipulation de tableaux

Question 1 Écrire une fonction `array_blit` telle que `array_blit tab1 debut1 tab2 debut2 nb` copie les `nb` éléments de `tab1` à partir de l'indice `debut1` dans le tableau `tab2` à partir de l'indice `debut2`.

On appellera toujours cette fonction en faisant en sorte qu'il n'y ait pas d'accès erronés en mémoire.

Tests

```
let t1 = Array.init 5 (fun i -> i) in
let t2 = Array.init 6 (fun i -> i+1) in
array_blit t1 3 t2 1 2;
assert(t2 = [|1;3;4;4;5;6|]);
let t1 = [| 'a';'b';'c' |] in
let t2 = [| 'd'; 'e' |] in
array_blit t1 1 t2 0 2;
assert(t2 = [|'b';'c'|]);
"Valide";;
```

Corrigé

```
let array_blit (tab1:'a array) (debut1:int)
               (tab2:'a array) (debut2:int) (nb:int) : unit =
    for i = 0 to nb-1 do
        tab2.(debut2 + i) <- tab1.(debut1 + i)
    done
```

Question 2 Écrire une fonction `array_extend` telle que l'appel `array_extend tab debut nb` renvoie un **nouveau** tableau contenant le même contenu que `tab` sauf que l'on a rajouté `nb` éléments à partir de l'indice `debut`. La valeur des éléments rajoutés n'est pas signifiante.

On suppose que `tab` est non vide.

Si `debut` vaut la longueur de `tab`, les nouveaux éléments sont rajoutés après la recopie de `tab`.

On utilisera `array_blit`.

Tests

```
let t = Array.init 5 (fun i -> i) in
let t2 = array_extend t 3 5 in
assert(Array.length t2 = 10
    && Array.sub t2 0 3 = [|0;1;2|]
    && Array.sub t2 8 2 = [|3;4|]);
let t = Array.init 5 (fun i -> char_of_int (i+97) ) in
let t2 = array_extend t 0 3 in
assert(Array.length t2 = 8 && Array.sub t2 3 5 = t);
let t2 = array_extend t 5 3 in
assert(Array.length t2 = 8 && Array.sub t2 0 5 = t);
"Valide";;
```

Corrigé

```
let array_extend (tab:'a array) (debut:int) (nb:int) : 'a array =
    let n = Array.length tab in
    let tabn = Array.make (n + nb) tab.(0) in
    array_blit tab 0 tabn 0 debut;
    array_blit tab debut tabn (debut + nb) (n - debut);
    tabn
```

En exécutant le code suivant, vous devriez obtenir `[|0; 1; 2; 0; 0; 0; 0; 0; 3; 4|]` si vous avez rempli les nouvelles cases avec `tab.(0)` comme dans le corrigé (seules les trois premières et les deux dernières valeurs sont imposées).

```
let t = Array.init 5 (fun i -> i)
in array_extend t 3 5
```

Question 3 Écrire une fonction `array_concat` telle que l'appel `array_concat tab1 tab2` renvoie le **nouveau** tableau obtenu en plaçant les éléments de `tab2` à la suite de ceux de `tab1`.

Tests

```
assert(array_concat [|1;2|] [|3|] = [|1;2;3|]);
assert(array_concat [||] [||] = [||]);
assert(array_concat [||] [|1|] = [|1|]);
assert(array_concat [|1|] [||] = [|1|]);
"Valide";;
```

Corrigé

```
let array_concat (tab1:'a array) (tab2:'a array) : 'a array =
    let n1 = Array.length tab1 in
    let n2 = Array.length tab2 in
    if n1 = 0 then tab2
    else if n2 = 0 then tab1
    else
        let tab = Array.make (n1+n2) tab1.(0) in
        array_blit tab1 0 tab 0 n1;
        array_blit tab2 0 tab n1 n2;
        tab
```

Attention, quand l'un des deux tableaux est vide, ce corrigé renvoie l'autre tableau lui-même et non une copie.

Question 4 Écrire une fonction `array_delete` telle que l'appel `array_delete tab debut nb` renvoie un **nouveau** tableau obtenu en supprimant `nb` éléments de `tab` à partir de l'indice `debut`.

On suppose que `nb > 0` et donc que `tab` est non vide.

Tests

```
assert(array_delete [|1;2;3;4;5;6|] 2 2 = [|1;2;5;6|]);
assert(array_delete [|1;2;3;4;5;6|] 0 3 = [|4;5;6|]);
"Valide";;
```

Corrigé

```
let array_delete (tab : 'a array) (debut : int)
                 (nb : int) : 'a array =
    let n = Array.length tab in
    let t = Array.make (n - nb) tab.(0) in
    array_blit tab 0 t 0 debut;
    array_blit tab (debut + nb) t debut (n - nb - debut);
    t
```

17.1.2 Premier type de tableaux dynamiques

On définit le type suivant pour contenir un tableau dynamique. On va devoir réallouer les éléments, donc on aura besoin d'accéder en écriture au tableau qui les contient, le plus simple dans un premier temps est de faire une référence sur un tableau.

Note le type est un type synonyme, par défaut, `OCaml` va afficher `'a array ref` dans ses résultats mais on peut forcer l'affichage de `'a dynarray` en spécifiant les types dans les fonctions comme on va le faire en dessous.

```
type 'a dynarray = 'a array ref
```

Comme on va définir des tableaux en partant du tableau vide, comme en Python avec `[]`, on peut se permettre de ne pas donner le type des éléments. En fait, contrairement à ce qu'il semble, la valeur `v` ci-dessous n'a pas un type magique : le `'a` sera instancié le plus vite possible, ce qu'on peut voir avec le `'_weak1` dans son type.

Attention : vraiment comprendre ces notions de paramètres génériques faibles est trop complexe et dépasse le cadre du cours de CPGE.

```
let dynarray_init () = ref [||]
```

```
let v = dynarray_init ()
```

```
v := [|1; 2|]
```

```
(* Maintenant v est un int dynarray *)
v
```

On va maintenant définir des fonctions d'ajout et de suppression d'éléments dans le tableau. Pour cela, on va réallouer le tableau grâce aux fonctions de la partie précédente. **Attention** : on rappelle que dans `array_extend` on a supposé le tableau non vide, il faudra donc sûrement traiter ce cas à part.

Question 5 Écrire une fonction `dynarray_append` telle que l'appel `dynarray_append tab x` ajoute `x` à la fin du tableau `tab`.

Tests

```
let t = dynarray_init () in
dynarray_append t 1;
dynarray_append t 2;
dynarray_append t 3;
dynarray_append t 4;
assert(!t = [|1;2;3;4|]);
"Valide";;
```

Corrigé

```
let dynarray_append (tab : 'a dynarray) (x : 'a) : unit =
    let n = Array.length !tab in
    if n = 0 then tab := [|x|]
    else begin
        let t = array_extend !tab n 1 in
        t.(n) <- x;
        tab := t
    end
```

Question 6 Écrire une fonction `dynarray_pop` telle que l'appel à `dynarray_pop tab` supprime le dernier élément de `tab` et le renvoie.

Tests

```
let t = ref [|1;2;3;4|] in
let x = dynarray_pop t in
assert(x = 4 && !t = [|1;2;3|]);
"Valide";;
```

Corrigé

```
let dynarray_pop (tab : 'a dynarray) : 'a =
    let n = Array.length !tab in
    let x = !tab.(n-1) in
    let t = array_delete !tab (n-1) 1 in
    tab := t;
    x
```

Chaque ajout et chaque suppression recopie tout le tableau : une suite de n ajouts à partir du tableau vide coûte $\sum_{k=1}^{n} O(k) = O(n^2)$.

17.1.3 Type efficace pour les tableaux dynamiques

Afin de résoudre les problèmes de complexité observés précédemment, on va définir un nouveau type pour les tableaux en permettant d'avoir un tableau contenant plus de cases que ce qui est vraiment utilisé. Ainsi, si on a un tableau de taille 10 mais qui ne contient que 9 éléments réels, on pourra en temps constant rajouter un élément. Pour cela, on va avoir besoin non seulement de gérer les éléments du tableau, mais aussi un paramètre `length` donnant la longueur logique, c'est-à-dire le nombre d'éléments signifiants du tableau.

```
type 'a dynarray = {
    mutable elements : 'a array;
    mutable length : int
}
```

```
let dynarray_init () = { elements = [||]; length = 0 }
```

Question 7 Écrire une fonction `dynarray_of_array` telle que l'appel `dynarray_of_array tab` où `tab` est un `array` renvoie le dynarray le contenant : on initialisera sa longueur à la longueur de `tab`.

Tests

```
assert(dynarray_of_array [|1;2|] = { elements = [|1;2|]; length = 2 });
"Valide";;
```

Corrigé

```
let dynarray_of_array (tab : 'a array) : 'a dynarray =
    { elements = tab; length = Array.length tab }
```

Question 8 Écrire des fonctions `get` et `set` telles que l'appel `get tab i` renvoie le i-ème élément du tableau contenu par `tab` et l'appel `set tab i x` place la valeur `x` dans le tableau `tab` à l'indice `i`. On vérifiera que `i` est un indice valide, c'est-à-dire strictement inférieur à la longueur logique.

Tests

```
let d = dynarray_of_array [|1;2|] in
assert(get d 1 = 2);
set d 1 3;
assert(d.elements = [|1;3|]);
"Valide";;
```

Corrigé

```
let get (tab : 'a dynarray) (i : int) : 'a =
    if i < 0 || i >= tab.length then invalid_arg "get";
    tab.elements.(i)

let set (tab : 'a dynarray) (i : int) (x : 'a) : unit =
    if i < 0 || i >= tab.length then invalid_arg "set";
    tab.elements.(i) <- x
```

Pour ajouter des éléments au tableau, on va regarder s'il y a de la place (`t.length < Array.length t.elements`) et sinon on va **doubler** sa taille. Si le tableau contenait 3 éléments, on va en allouer 6 donc 3 de plus, ce qui va permettre de faire 3 ajouts sans réallocation (dont celui en cours).

Attention : quand le tableau est vide, on ne pourra pas doubler sa longueur, on se contente de faire un tableau de taille 1 dans ce cas.

Question 9 Écrire une fonction `dynarray_append` telle que l'appel `dynarray_append tab x` ajoute `x` à la fin du tableau `tab` avec la stratégie exposée.

Tests

```
let t = dynarray_init () in
dynarray_append t 1;
dynarray_append t 2;
dynarray_append t 3;
dynarray_append t 4;
dynarray_append t 5;
assert(Array.sub t.elements 0 5 = [|1; 2; 3; 4; 5|]
    && t.length = 5);
"Valide";;
```

Corrigé

```
let dynarray_append (tab : 'a dynarray) (x : 'a) : unit =
    let n = Array.length tab.elements in
    if n = 0 then begin
        tab.elements <- [|x|];
        tab.length <- 1
    end
    else begin
        if tab.length = Array.length tab.elements
        then tab.elements <-
            array_extend tab.elements tab.length tab.length;
```

```
        tab.elements.(tab.length) <- x;
        tab.length <- 1 + tab.length
    end
```

Question 10 Écrire une fonction `dynarray_pop` telle que l'appel `dynarray_pop tab` supprime le dernier élément de `tab` et le renvoie.

On ne supprime rien physiquement, on se contente de changer la valeur `length`.

Tests

```
let t = { elements = [|1;2;3;4;5|]; length = 4 } in
let x = dynarray_pop t in
assert(x = 4 && t.elements = [|1;2;3;4;5|] && t.length = 3);
"Valide";;
```

Corrigé

```
let dynarray_pop (tab : 'a dynarray) : 'a =
    if tab.length = 0 then failwith "tableau vide";
    let x = tab.elements.(tab.length-1) in
    tab.length <- tab.length - 1;
    x
```

Question 11 Prouver que si on effectue une suite de n `append` sur un tableau vide on fait $O(n)$ opérations élémentaires.

Même question avec des `append` et des `pop`.

Corrigé

Un `append` sans réallocation coûte $O(1)$. Une réallocation a lieu quand la capacité c est atteinte, et coûte $O(c)$ (recopie de c éléments). Lors de n ajouts à partir du tableau vide, les capacités successives sont $1, 2, 4, \ldots, 2^k$ avec $2^k < n$ pour celles qui sont recopiées : le coût total des recopies est $O(1 + 2 + \ldots + 2^k) = O(2^{k+1}) = O(n)$, et le coût total est donc $O(n)$.

Avec des `append` et des `pop`, un `pop` coûte $O(1)$ et ne diminue jamais la capacité. Si on effectue n opérations, la longueur reste inférieure à n, donc la capacité aussi (elle vaut au plus le double d'une longueur atteinte). Les réallocations se font pour des capacités deux à deux distinctes, puissances de 2 et inférieures à n : leur coût total est encore $O(n)$.

Si on effectue 1 milliard d'`append` suivis d'1 milliard de `pop`, la taille physique sera toujours d'1 milliard d'éléments. Cela semble peu efficace et on aimerait que la taille physique soit toujours proche du nombre réel d'éléments.

Question 12 Comment peut-on modifier `dynarray_pop` pour s'assurer que le nombre d'éléments réels est toujours proportionnel à la taille physique tout en conservant un temps constant en moyenne ?

Faites la modification et faites une preuve assurant que cela fonctionne.

Corrigé

Diviser la capacité par 2 dès que le tableau est à moitié vide ne convient pas : en alternant `append` et `pop` autour de cette limite, on réalloue à chaque opération. On divise donc la capacité par 2 quand le tableau n'est plus rempli qu'au quart :

```
let dynarray_pop (tab : 'a dynarray) : 'a =
    if tab.length = 0 then failwith "tableau vide";
```

```
    let x = tab.elements.(tab.length-1) in
    tab.length <- tab.length - 1;
    let cap = Array.length tab.elements in
    (* on divise la capacité par 2 quand le tableau est rempli au quart *)
    if 4 * tab.length <= cap && cap > 1
    then tab.elements <- Array.sub tab.elements 0 (cap / 2);
    x
```

En notant ℓ la longueur et c la capacité, on vérifie que $c \leq 4\ell$ reste vrai après chaque opération dès que $\ell \geq 1$ (après une division, $c = 2\ell$) : la taille physique reste proportionnelle au nombre d'éléments.

Pour la complexité, on utilise la méthode du potentiel (voir le chapitre sur la complexité amortie) avec $\Phi = |\, 2\ell - c \,|$, qui est positif et nul pour le tableau vide initial ($\ell = c = 0$). Une opération sans réallocation modifie ℓ de 1 donc Φ d'au plus 2 : coût amorti $O(1)$. Juste avant un doublement, $\ell = c$ donc $\Phi = c$, et juste après $\Phi = 2(c+1) - 2c = 2$: la baisse de Φ paie la recopie de c éléments. Juste avant une division par 2, $\ell = \frac{c}{4}$ donc $\Phi = \frac{c}{2}$, et juste après $\Phi = |\, \frac{c}{2} - \frac{c}{2} \,| = 0$: la baisse de Φ paie la recopie des $\frac{c}{2}$ cases. Le coût amorti de chaque opération est donc $O(1)$, et une suite de n opérations coûte $O(n)$.

CHAPITRE 18

Piles et files, structures abstraites et implémentations

DANS CE CHAPITRE

18.1 Structures abstraites
18.2 Implémentations concrètes

Dans ce chapitre, nous allons étudier les structures de données permettant de stocker des éléments et de les récupérer ultérieurement. Ces structures sont omniprésentes en algorithmique pour gérer des collections d'éléments « en attente » de traitement.

Nous adopterons une démarche en deux temps : d'abord la définition formelle de ces structures de manière **abstraite** (quelles opérations offrent-elles ? comment les manipuler sans connaître leur mécanique interne ?), puis l'étude de leurs **implémentations** concrètes en machine, avec un focus sur les complexités temporelles et spatiales.

18.1 Structures abstraites

18.1.1 Le concept de Sac (Bag)

De manière très générale, on peut définir un concept abstrait de structure d'attente appelé le **Sac** (*Bag* en anglais).

Définition 18.1 — : Structure de Sac

Un **Sac** est une structure de données abstraite qui maintient une collection d'éléments. L'ordre interne n'est pas spécifié à ce niveau d'abstraction. Un sac abstrait expose minimalement les opérations suivantes :

- `creer_sac()` : initialise un sac vide.
- `est_vide(s)` : renvoie vrai si le sac `s` ne contient aucun élément.
- `ajouter(s, x)` : insère l'élément `x` dans le sac `s`.
- `retirer(s)` : retire et renvoie un élément du sac `s`. Si le sac est vide, le comportement est indéfini (erreur).

En pratique, on utilise très rarement un « Sac » générique. On le spécialise en imposant une règle stricte sur **l'ordre de retrait** des éléments. C'est cette règle de retrait qui distingue fondamentalement une pile d'une file.

18.1.2 Piles (Stacks)

Principe et vocabulaire

Une pile est une spécialisation du sac fonctionnant sur le principe **LIFO** (**L**ast **I**n **F**irst **O**ut). Le dernier élément ajouté est le premier à être retiré.

La bonne manière de se représenter une pile est d'imaginer une pile d'assiettes ou de dossiers sur un bureau : on ajoute toujours un dossier sur le dessus de la pile, et lorsqu'on traite un dossier, on prend toujours celui qui est au sommet.

Dictionnaire Anglais / Français Dans la littérature algorithmique et dans la plupart des bibliothèques standards (API), le vocabulaire anglophone est la norme incontournable. Il est impératif de le maîtriser :

- **Pile** : *Stack*
- **Empiler** (ajouter un élément au sommet) : *Push*
- **Dépiler** (retirer et renvoyer l'élément au sommet) : *Pop*
- **Sommet** (lire l'élément au sommet sans le retirer) : *Peek* ou *Top*
- **Est vide** : *Is_empty* ou *Empty*

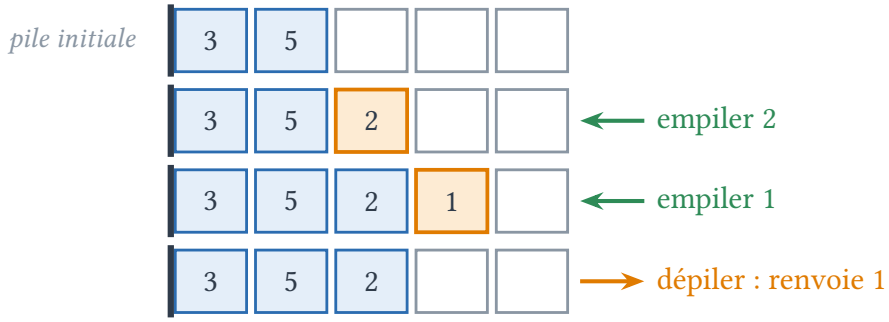


Figure 18.1 — Évolution d'une pile représentée par un tableau : deux empilements puis un dépilement

Le principe du parcours destructeur

Dans le paradigme des structures abstraites pures, il n'y a pas d'opération pour accéder au i-ème élément d'une pile. Le seul élément accessible est le sommet. Pour parcourir ou chercher un élément au fond d'une pile, on est obligé de **dépiler** les éléments un à un. Ce processus **détruit** la pile originale.

Si l'on a besoin de conserver les données de la pile initiale, il faut impérativement sauvegarder les éléments dépilés dans une structure temporaire (généralement une seconde pile).

Longueur d'une pile avec deux piles

Imaginons le problème suivant : on souhaite écrire une fonction `longueur` qui calcule le nombre d'éléments d'une pile `p`, en n'utilisant *que* les opérations abstraites de pile, et qui garantit qu'à la fin de la fonction, la pile `p` a retrouvé son état exact d'origine.

L'algorithme consiste à :

1. Créer une pile temporaire `temp`.
2. Tant que `p` n'est pas vide, `pop(p)`, incrémenter un compteur, et `push(temp, element)`.
3. Une fois `p` vide, vider `temp` pour remettre les éléments dans `p` : tant que `temp` n'est pas vide, `push(p, pop(temp))`.

Python

```python
def longueur_pile(p):
    '''Calcule la longueur de la pile p sans la modifier de manière permanente.'''
    temp = creer_pile()
    compteur = 0

    # Parcours destructeur et sauvegarde
    while not est_vide(p):
        x = depiler(p)
        empiler(temp, x)
        compteur += 1

    # Restauration de la pile initiale
    while not est_vide(temp):
        x = depiler(temp)
        empiler(p, x)

    return compteur
```

OCaml

```
let longueur_pile p =
  let temp = cree_pile () in
  let rec depiler_tout acc =
    if est_vide p then acc
    else begin
      let x = depile p in
      empile temp x;
      depiler_tout (acc + 1)
    end
  in
  let res = depiler_tout 0 in

  let rec restaurer () =
    if not (est_vide temp) then begin
      empile p (depile temp);
      restaurer ()
    end
  in
  restaurer ();
  res
```

18.1.3 Files (Queues)

Principe et vocabulaire

Une file est une structure de données abstraite permettant d'ajouter et de retirer des éléments selon le principe **FIFO** (**F**irst **I**n **F**irst **O**ut). Le premier élément inséré sera le premier à en sortir, exactement comme une file d'attente à la caisse d'un magasin.

> **Dictionnaire Anglais / Français**
> - **File** : *Queue*
> - **Enfiler** (ajouter un élément à l'arrière) : *Enqueue* (parfois *Push*)
> - **Défiler** (retirer et renvoyer l'élément à l'avant) : *Dequeue* (parfois *Pop*)
> - **Avant / Tête** : *Front* ou *Head*
> - **Arrière / Queue** : *Rear* ou *Tail*

Rotation d'une file

Contrairement à la pile qui nécessite une seconde pile pour préserver ses éléments lors d'un parcours, la file possède une propriété remarquable due à sa géométrie « en tube » : la **rotation**. L'opération de rotation consiste à défiler l'élément en tête et à le ré-enfiler immédiatement à la fin de la file.

```
def rotation(f):
    x = defiler(f)
    enfiler(f, x)
```

En effectuant exactement N rotations consécutives (où N est la taille de la file), la file retrouve son état initial !

Calcul de la longueur à l'aide d'une sentinelle

Comment calculer la longueur d'une file si l'on ne connaît pas N à l'avance, tout en utilisant la rotation pour ne pas la détruire ? L'astuce classique consiste à enfiler une **sentinelle** : une valeur spéciale reconnaissable (par exemple `None` en Python, ou un pointeur `NULL` en C), dont on est sûr qu'elle n'apparaît pas parmi les éléments de la file.

L'algorithme :

1. Enfiler la sentinelle.
2. Initialiser un compteur à 0.
3. Boucle : défiler un élément.
   - Si c'est la sentinelle, on arrête la boucle.
   - Sinon, on incrémente le compteur, et on ré-enfile l'élément (rotation).

À la fin de cet algorithme, tous les éléments originaux ont fait exactement un tour complet de la file, et la sentinelle a été détruite. La file est restaurée, et on a la longueur.

Python

```python
def longueur_file(f):
    enfiler(f, None)  # La sentinelle
    compteur = 0

    while True:
        x = defiler(f)
        if x is None:
            break
        compteur += 1
        enfiler(f, x) # Rotation

    return compteur
```

18.1.4 Files de priorité (Priority Queues)

Une **file de priorité** est une structure où chaque élément ajouté possède une « priorité » (qui peut être sa propre valeur, ou une clé associée). La règle de retrait n'est ni LIFO ni FIFO : l'opération `retirer()` extrait systématiquement **l'élément de plus haute priorité** (souvent défini comme la plus petite valeur, par exemple dans un algorithme de graphe).

> **Dictionnaire Anglais / Français**
> - **File de priorité** : *Priority Queue*
> - En général, l'implémentation sous-jacente utilise un **Tas** (*Heap*).

18.1.5 Choisir la bonne structure et applications

Le choix de la structure « sac » dicte l'ordre d'exploration d'un espace de recherche, ce qui modifie drastiquement le comportement des algorithmes.

Illustration classique : le Flood-Fill

Prenons l'algorithme de remplissage par diffusion (*flood-fill*), typique de l'outil « pot de peinture » des logiciels de dessin. L'objectif est de colorier une zone de pixels de même couleur. On maintient un Sac de pixels à visiter. À chaque étape, on retire un pixel du Sac ; s'il n'est pas déjà colorié, on le colorie et on ajoute au Sac ses voisins ayant encore la couleur d'origine (un même pixel peut donc être présent plusieurs fois dans le Sac).

- **Si le Sac est une Pile (LIFO)** : Le comportement sera un **Parcours en Profondeur** (*Depth-First Search* ou *DFS*). L'algorithme va suivre un chemin le plus loin possible avant de revenir sur ses pas. Visuellement, le remplissage avance sous la forme de longs « serpents » qui rebondissent sur les bords.
- **Si le Sac est une File (FIFO)** : Le comportement sera un **Parcours en Largeur** (*Breadth-First Search* ou *BFS*). L'algorithme traitera tous les pixels à distance 1, puis à distance 2. Visuellement, le remplissage se fait par « fronts concentriques » réguliers autour du point de départ (des losanges si l'on considère les 4 voisins d'un pixel, puisque la distance est alors celle de Manhattan).

18.2 Implémentations concrètes

D'un point de vue algorithmique, les opérations d'ajout et de retrait dans une pile ou une file doivent s'exécuter en temps constant, $\mathcal{O}(1)$, éventuellement seulement en complexité amortie. Nous allons voir comment garantir cette complexité en C et en OCaml.

18.2.1 Implémentation des piles

Dans un tableau borné (C)

C'est l'implémentation la plus courante et la plus rapide si on connaît une borne supérieure au nombre d'éléments. On utilise un tableau statique (ou alloué dynamiquement) `t` de taille N, et un entier `fond` qui représente l'indice de la première case libre.

- **Initialisation** : `fond = 0`
- **Empiler(x)** : On écrit `x` à l'indice `fond`, puis on incrémente `fond`.
- **Dépiler()** : On décrémente `fond`, et on renvoie l'élément à ce nouvel indice. (L'élément reste physiquement en mémoire, mais il est logiquement « hors » de la pile et sera écrasé au prochain `empiler`).

C

```c
#include <stdlib.h>
#include <stdbool.h>
#include <assert.h>

#define N 1000 // Capacité maximale de la pile

struct pile {
    int *t;
    int fond;
};
typedef struct pile pile;

pile *pile_creer() {
    pile *p = malloc(sizeof(pile));
    p->t = malloc(sizeof(int) * N);
    p->fond = 0;
    return p;
}

void pile_detruire(pile *p) {
    free(p->t);
    free(p);
}

void empiler(pile *p, int x) {
    assert(p->fond < N); // Vérification de débordement (Stack Overflow)
    p->t[p->fond] = x;
    p->fond++;
}

int depiler(pile *p) {
    assert(p->fond > 0); // Vérification de pile vide (Stack Underflow)
    p->fond--;
    return p->t[p->fond];
}

bool est_vide(pile *p) {
    return p->fond == 0;
}
```

Avec des listes chaînées (OCaml)

Dans les langages fonctionnels, les listes chaînées (simples) sont la structure de base. Ajouter un élément en tête d'une liste chaînée (`x :: liste`) ou récupérer la tête (`match liste with x :: q -> ...`) sont des opérations en $\mathcal{O}(1)$. Une liste est donc naturellement une pile.

Pour offrir une interface mutable (opérations avec effets de bord), on encapsule la liste dans une référence (pointeur) :

OCaml

```ocaml
type 'a pile = 'a list ref

let cree_pile () = ref []

let est_vide p = !p = []

let empile p x =
  p := x :: !p

let depile p =
```

```
match !p with
| [] -> failwith "Erreur : dépiler sur une pile vide"
| x :: q ->
  p := q;
  x
```

18.2.2 Implémentation des files

L'implémentation des files est plus délicate car les opérations se font aux deux extrémités de la structure (ajout à l'arrière, retrait à l'avant).

Dans un tableau borné (Circulaire en C)

Si l'on implémentait une file naïvement dans un tableau en ajoutant à l'indice i et en retirant à l'indice 0 (en décalant tout le monde), le retrait prendrait un temps $\mathcal{O}(N)$, ce qui est inacceptable.

On utilise donc un **tableau circulaire** avec deux indices : `debut` et `fin`.

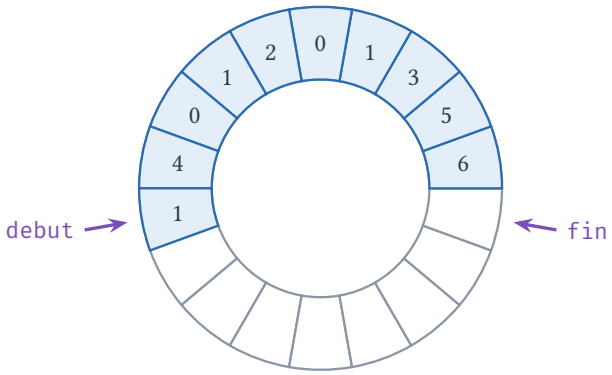


Figure 18.2 — File dans un tableau circulaire : debut repère le premier élément, fin la première case libre.

Dans cette représentation, `debut` est l'indice du premier élément de la file et `fin` celui de la première case libre, située juste après le dernier élément (les indices croissent dans le sens des aiguilles d'une montre).

Pour enfiler, on écrit l'élément dans la case `fin` puis on avance `fin` d'une case :

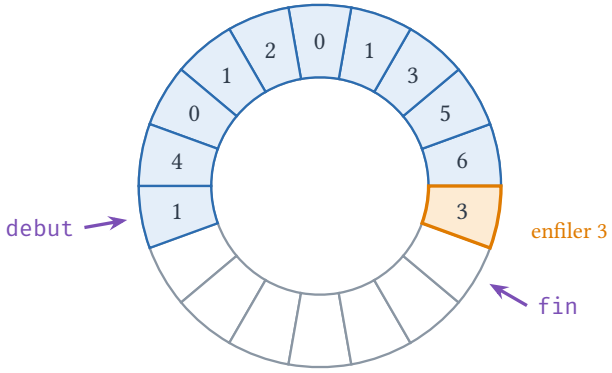


Figure 18.3 — Enfiler 3 : on écrit dans la case fin, puis fin avance d'une case.

Pour défiler, on lit l'élément de la case `debut` puis on avance `debut` d'une case :

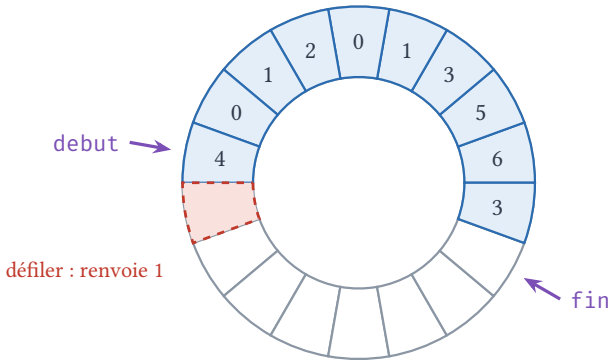


Figure 18.4 — Défiler : on lit la case debut (on renvoie 1), puis debut avance d'une case.

Lorsque `fin` (ou `debut`) arrive au bout du tableau en mémoire, il doit repartir à l'indice 0. L'astuce est d'utiliser l'opérateur modulo (`% N`).

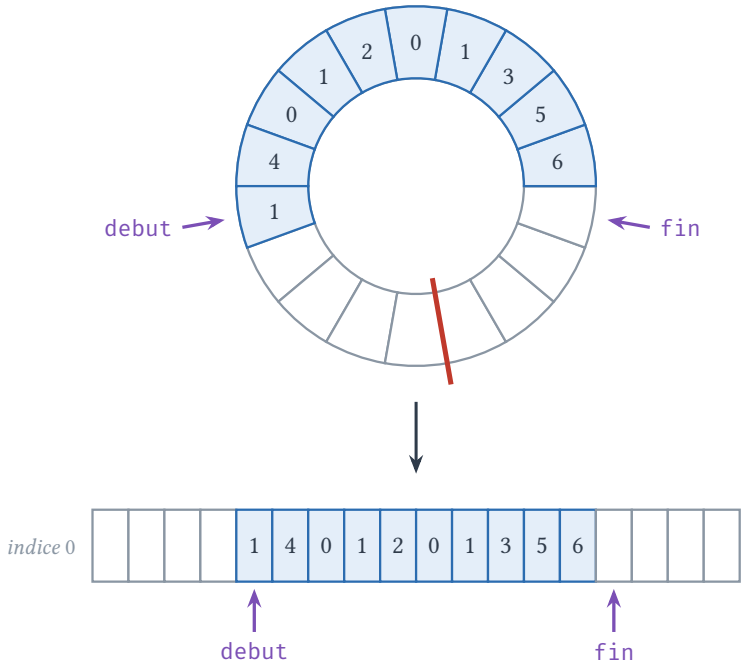


Figure 18.5 — Le même tableau circulaire, coupé et mis à plat comme il est rangé en mémoire.

C

```c
struct file {
    int *t;
    int debut;
    int fin;
    int taille; // On maintient la taille courante pour distinguer file vide de file pleine
};
typedef struct file file;

file *file_creer() {
    file *f = malloc(sizeof(file));
    f->t = malloc(sizeof(int) * N);
    f->debut = 0;
    f->fin = 0;
    f->taille = 0;
    return f;
}

void file_detruire(file *f) {
    free(f->t);
```

```c
    free(f);
}

bool file_est_vide(file *f) {
    return f->taille == 0;
}

void enfiler(file *f, int x) {
    assert(f->taille < N);
    f->t[f->fin] = x;
    f->fin = (f->fin + 1) % N; // Circularité
    f->taille++;
}

int defiler(file *f) {
    assert(f->taille > 0);
    int res = f->t[f->debut];
    f->debut = (f->debut + 1) % N; // Circularité
    f->taille--;
    return res;
}
```

Remarque 18.2 Le C ne permet pas de définir deux fonctions de même nom, même si elles portent sur des types différents. Si l'on souhaite utiliser une pile et une file dans un même programme, on ne peut pas appeler `est_vide` les deux tests : on préfixe donc le nom des fonctions par celui de la structure, comme pour `file_creer` et `file_detruire`. Le test de la pile gagnerait ainsi à s'appeler `pile_est_vide`.

Implémentation avec deux listes (OCaml)

Comment créer une file FIFO efficace dans un langage fonctionnel sans utiliser de tableaux ? L'astuce brillante consiste à utiliser **deux piles** (donc deux listes) :

- Une pile `entree` dédiée aux insertions.
- Une pile `sortie` dédiée aux retraits.

Pour enfiler, on pousse simplement dans `entree` ($\mathcal{O}(1)$). Pour défiler, on extrait de `sortie` ($\mathcal{O}(1)$). Mais que faire si `sortie` est vide ? On prend alors l'intégralité de la pile `entree`, on la renverse (`List.rev`), et on la place dans `sortie`.

Renverser une liste de longueur k coûte $\mathcal{O}(k)$. Mais ce coût élevé n'est payé que ponctuellement. Sur une longue séquence d'opérations, chaque élément est empilé une fois dans `entree`, transféré une seule fois dans `sortie` lors d'un renversement, et dépilé une fois. Le coût **amorti** reste donc de $\mathcal{O}(1)$ par opération.

Preuve Comptons une unité de coût par cellule de liste créée ou parcourue. Considérons une suite de m opérations partant d'une file vide, dont e appels à `enfile`. Chaque élément enfilé passe au plus une fois de `entree` vers `sortie` (une fois dans `sortie`, il n'en sort que pour être défilé). Le coût total des renversements est donc au plus proportionnel à $e \leq m$, et en dehors des renversements chaque opération est en $\mathcal{O}(1)$. Le coût total est ainsi en $\mathcal{O}(m)$, soit $\mathcal{O}(1)$ amorti par opération.

De manière équivalente, avec le potentiel $\Phi = |\,\texttt{entree}\,|$: `enfile` a un coût réel 1 et augmente Φ de 1 ; un `defile` qui renverse une liste de longueur k a un coût réel $k+1$ et fait diminuer Φ de k ; les autres `defile` ont un coût réel 1 et ne modifient pas Φ. Le coût amorti (coût réel $+\Delta\Phi$) de chaque opération est donc au plus 2, et comme $\Phi \geq 0$ avec $\Phi = 0$ initialement, le coût réel total de m opérations est au plus $2m$.

Cette analyse suppose que la file est utilisée de manière *éphémère* (chaque version n'est utilisée qu'une fois), ce qui est le cas ici puisque les champs sont mutables. Avec une version persistante (sans `mutable`), on pourrait appeler plusieurs fois `defile` sur une même version ayant `sortie` vide et payer plusieurs fois le même renversement.

□

OCaml

```
type 'a file = {
  mutable entree : 'a list;
  mutable sortie : 'a list;
}

let cree_file () = { entree = []; sortie = [] }

let est_vide f = f.entree = [] && f.sortie = []

let enfile f x =
  f.entree <- x :: f.entree

let defile f =
  match f.sortie with
  | x :: q ->
      f.sortie <- q;
      x
  | [] ->
      match List.rev f.entree with
      | [] -> failwith "Erreur : défiler sur une file vide"
      | x :: q ->
          f.sortie <- q;
          f.entree <- [];
          x
```

TRAVAUX DIRIGÉS 19

19 Piles et files

19.1 TD — Piles et files

MP2I, année scolaire 2021/2022.

La plupart des exercices donnés ici peuvent être implémentés en `OCaml` ou en `C`, mais l'objectif premier est de réfléchir et d'en faire plus que si on le faisait en TP.

Rappel

Chacune des structures piles et files est caractérisée par deux opérations essentielles, ajouter et retirer un élément selon la stratégie suivante :

- Une pile est **LIFO** : Last In First Out. Le dernier élément ajouté est le premier qu'on retire.
- Une file est **FIFO** : First In First Out. Le premier élément ajouté est le premier qu'on retire.

Les opérations de création et de test si la structure est vide sont dans la suite supposées peu intéressantes. On se concentre sur ajouter/retirer.

19.1.1 Réalisation d'une file

Exercice 1. On souhaite réaliser une file avec une liste chaînée. Indiquer comment réaliser les opérations et leur complexité :

- avec des listes comme en `OCaml` ;
- en utilisant explicitement des maillons mais sans liste doublement chaînée.

Exercice 2. On souhaite maintenant réaliser une file avec un tableau de taille fixe. La stratégie va être de maintenir deux indices : le début et la fin de la file et de considérer que les bords du tableau se rejoignent.

Indiquer comment réaliser les opérations et leur complexité.

Exercice 3. Montrer qu'on peut réaliser efficacement une file avec deux piles et indiquer la complexité des opérations.

On en déduit une implémentation des files en `OCaml` avec deux références sur des listes.

Exercice 4. Montrer qu'on peut réaliser efficacement une pile avec deux files et indiquer la complexité des opérations.

Exercice 5. Montrer qu'on peut réaliser inefficacement une pile avec une file et indiquer la complexité des opérations.

19.1.2 Algorithme de Dijkstra pour l'évaluation d'expressions

On a vu comment évaluer une expression postfixe avec une pile, Dijkstra a trouvé un algorithme utilisant deux piles pour évaluer une expression infixe, c'est-à-dire une expression standard avec des parenthèses $(1 + (2 \times 3) - (3 \times 4)) - 2$.

On considère deux piles, une pile de nombres et une pile d'opérateurs.

Pour chaque symbole :

- Si c'est une parenthèse ouvrante, on l'ignore ;
- Si c'est un nombre on le pousse sur la pile de nombres ;
- Si c'est un opérateur on le pousse sur la pile d'opérateurs ;
- Si c'est une parenthèse fermante, on dépile un opérateur et autant de nombres que son arité (son nombre d'opérandes), puis on empile le résultat.

1. Évaluer ainsi $(1 + ((2 + 3) \times (4 \times 5)))$.
2. Justifier que la méthode fonctionne.
3. En déduire une conversion en notation postfixe.

19.1.3 Sous-chaîne parenthésée

On considère ici le problème des expressions bien parenthésées dans le contexte simple où il n'y a que deux caractères `'('` et `')'`. Une chaîne est donc de la forme `(()((` ou `(()(()))` et on considère qu'elle est sous la forme d'un tableau de caractères.

On a vu qu'on pouvait vérifier si la chaîne était bien parenthésée avec un entier.

On cherche une sous-chaîne bien parenthésée de longueur maximale, c'est-à-dire un sous-tableau entre les indices i et j bien parenthésé et tel que $j - i + 1$ soit maximal. Votre algorithme doit être de complexité $O(n)$ où n est la longueur initiale de la chaîne.

On vous demande également de prouver que votre algorithme est correct.

20

TRAVAUX PRATIQUES 20

Piles et files

DANS CE CHAPITRE

20.1 Piles
20.2 File avec deux piles

20.1 Piles

On redonne l'interface des piles en `OCaml` (voir le chapitre sur les piles et files). On utilisera directement ces fonctions dans la suite.

```
let creer_pile () = ref []
let empile p x = p := x :: !p
let depile p = match !p with
    | [] -> failwith "pile vide"
    | t::q -> p := q; t
let est_vide p = !p = []
```

Comme on va être amenés à dépiler en ignorant la valeur renvoyée, `OCaml` va faire un avertissement car on jette la valeur alors qu'elle n'est pas de type `unit`. Pour éviter cet avertissement, on utilise la fonction `ignore` : `ignore (depile p)` pourra ainsi être appelé pour retirer une valeur de la pile en l'ignorant.

20.1.1 Expressions bien parenthésées

Avec un seul type de parenthèses

On va considérer des chaînes de caractères contenant des parenthèses et des lettres comme `"(a(bc)a)"` et on va chercher à savoir si elles sont bien parenthésées : c'est-à-dire que toutes les parenthèses ouvertes sont bien fermées. Pour cela, il suffit de compter le nombre de parenthèses ouvertes non encore fermées, de s'assurer qu'il ne devient jamais négatif et qu'à la fin il vaut 0.

Rappel, on accède au i-ème caractère de la chaîne `s` avec `s.[i]` et on obtient sa taille avec `String.length`.

Question 1 Écrire une fonction `bien_parenthesee : string -> bool`.

Tests

```
assert(bien_parenthesee "(a(bc)a)");
assert(not (bien_parenthesee "(a(bc)a"));
assert(not (bien_parenthesee "(a)bc)a("));
assert(bien_parenthesee "(()(()))(()())(())");
"OK";;
```

Corrigé

```
let bien_parenthesee s : bool =
    let ouvertes = ref 0 in
    let bp = ref true in
    for i = 0 to String.length s - 1 do
       let c = s.[i] in
       if c = '('
       then incr ouvertes;
       if c = ')'
       then begin
           if !ouvertes = 0 then bp := false;
           decr ouvertes
       end
    done;
    !ouvertes = 0 && !bp
```

Avec plusieurs types de parenthèses

Maintenant, on va considérer une liste `l` de couples `(ouvrant,fermant)` comme `('(', ')')` ou `('{','}')` indiquant quels types de parenthèses sont autorisés et on va poser la même question du parenthésage. Ainsi avec la liste `let l = [('(', ')'); ('{', '}'); ('[', ']') ]` la chaîne `(a{bc})[a(b)]` est bien parenthésée mais `(a}` ne l'est pas (pas le même symbole qui ferme) et `([(b)]` non plus (pas de `)` fermante).

On ne va donc plus pouvoir utiliser un simple entier. On ne peut pas non plus utiliser un entier par symbole car cela ne permettrait pas de détecter des problèmes comme `([)]` (*exercice : vous en convaincre*). On va donc utiliser une pile de symboles de fermeture attendus. Au départ, la pile est vide, ensuite, à chaque fois qu'on voit un symbole d'ouverture on empile le symbole de fermeture correspondant, et si on voit un symbole de fermeture on s'assure que ce soit le sommet de la pile, que l'on dépile.

Question 2 Écrire une fonction `bien_parenthesee_l : (char * char) list -> string -> bool`.

Tests

```
assert(bien_parenthesee_l [('(',')')] "(a(bc)a)");
assert(not (bien_parenthesee_l [('(',')')] "(a(bc)a"));
assert(not (bien_parenthesee_l [('(',')')] "(a)bc)a("));
assert(bien_parenthesee_l [('(',')')] "(()(()))(()())(())");
assert(bien_parenthesee_l [('(', ')'); ('{', '}'); ('[', ']')] "([]{})[[()]()}");
assert(not (bien_parenthesee_l [('(', ')'); ('{', '}'); ('[', ']')] "([)]"));
assert(not (bien_parenthesee_l [('(', ')'); ('{', '}'); ('[', ']')] "([)}"));
"OK";;
```

Corrigé

```
let bien_parenthesee_l (l:(char * char) list) (s:string) : bool =
    let ouvertes = creer_pile () in
    let bp = ref true in
    for i = 0 to String.length s - 1 do
       let c = s.[i] in
       let rec aux l = match l with
           | (o,f)::q ->
               if c = o
               then empile ouvertes f;
               if c = f
               then bp := !bp && (not (est_vide ouvertes) && depile ouvertes = f);
               aux q
           | [] -> ()
       in aux l
    done;
    est_vide ouvertes && !bp
```

20.1.2 Pile pour évaluer des expressions

Forme postfixe

On va considérer ici des expressions écrites sous forme postfixe.

La présentation traditionnelle des expressions est infixe : le symbole d'opération est au milieu des opérandes, comme dans `2 + 3`. Cette notation est naturelle mais a le désavantage de nécessiter des parenthèses comme dans `(2 + 3) * 5`. Une autre possibilité est d'avoir une notation postfixe `2 3 +` où le symbole vient après ses opérandes. L'avantage est de ne pas avoir besoin de parenthèses, en effet, on peut écrire `2 3 + 5 *` dans l'exemple précédent.

En `OCaml`, on représentera une telle expression sous la forme d'une liste contenant :

- soit des entiers pour les constantes ;
- soit une opération parmi `+`, `-`, `*`, `/`.

Pour cela on définit le type suivant :

```
type op = Plus | Minus | Mult | Div
type token = C of int | O of op
```

L'expression précédente correspond alors à la liste `[ C 2; C 3; O Plus; C 5; O Mult ]`.

Pour évaluer une telle expression, on utilise une pile, initialement vide, et on parcourt de gauche à droite l'expression postfixe. Si la valeur lue est

- un entier : on le place sur la pile ;
- un symbole d'opération : on dépile deux valeurs sur la pile et on effectue l'opération, puis on empile le résultat.

Quand on a fini le parcours, il ne doit y avoir qu'une seule valeur sur la pile qui est le résultat (sinon, c'est un problème similaire à un mauvais parenthésage). On dit qu'une expression qui permet d'aboutir à une unique valeur à la fin du parcours, sans jamais dépiler une pile vide, est **bien formée**.

Pour les opérations non commutatives comme `-` et `/`, on adopte la convention que `a b -` calcule `a - b`. Il faudra donc faire attention à respecter cela dans l'évaluation. **On effectue aussi des divisions entières ici et non flottantes.**

Question 3 Écrire une fonction d'évaluation des expressions `evalue : token list -> int`.

Tests

```
assert(evalue [C 1; C 2; O Plus] = 3);
assert(evalue [C 1; C 2; O Minus] = -1);
assert(evalue [C 3; C 1; O Minus; C 5; O Mult] = 10);
assert(evalue [C 1; C 2; O Plus;  C 2; O Div] = 1);
assert(evalue [C 1; C 2; C 3; C 4; C 5; C 6; O Plus; O Mult; O Div; O Minus; O Plus] = 3);
"OK";;
```

Corrigé

```
let evalue (e : token list) : int =
    let p = creer_pile () in
    let rec aux e = match e with
        | C k :: e' -> empile p k; aux e'
        | O o :: e' ->
            let op2 = depile p in
            let op1 = depile p in
            let res = match o with
                | Plus -> op1 + op2
                | Minus -> op1 - op2
```

```
                | Mult -> op1 * op2
                | Div -> op1 / op2
            in empile p res; aux e'
        | [] -> ()
    in
    aux e;
    let res = depile p in
    if not (est_vide p) then failwith "expression mal formée";
    res
```

Forme infixe : algorithme de Dijkstra

On a vu comment évaluer une expression postfixe avec une pile. Dijkstra a trouvé un algorithme utilisant deux piles pour évaluer une expression infixe, c'est-à-dire une expression standard avec des parenthèses comme $((1 + (2 \times 3)) - (3 \times (4 \times 2)))$.

Attention Il y a même des parenthèses autour de l'expression entière et on place également les parenthèses pourtant inutiles des opérations associatives : chaque opération est entourée de parenthèses. On verra plus tard qu'il s'agit ici d'un arbre binaire.

On considère deux piles, une pile de nombres et une pile d'opérateurs.

Pour chaque symbole :

- si c'est une parenthèse ouvrante, on l'ignore ;
- si c'est un nombre on le pousse sur la pile de nombres ;
- si c'est un opérateur on le pousse sur la pile d'opérateurs ;
- si c'est une parenthèse fermante, on dépile un opérateur et autant de nombres que son arité (son nombre d'opérandes), puis on empile le résultat.

Pour simplifier, on ne considère ici que des opérations binaires. Il faudra donc retirer deux nombres de la pile à chaque fois.

Question 4 Évaluer ainsi à la main $(1 + ((2 + 3) \times (4 \times 5)))$.

Corrigé

En notant les piles avec le sommet à droite :

Symbole lu	Pile de nombres	Pile d'opérateurs
`(1+((2+3`	1 2 3	1. +
`)`	1 5	+
`*(4*5`	1 5 4 5	1. * *
`)`	1 5 20	1. *
`)`	1 100	+
`)`	101	

Le résultat est 101.

Pour simplifier, on ne va considérer que des chiffres, ainsi, il suffira de s'assurer qu'un caractère est entre `'0'` et `'9'` pour le convertir avec `int_of_char c - int_of_char '0'` en un `int`.

Remarque 20.1 En `OCaml`, on peut filtrer une plage de caractères comme ici en utilisant le filtrage `| '0'..'9' ->`

Question 5 Écrire une fonction `evalue_infixe : string -> int` qui met en œuvre cet algorithme.

Tests

```
assert (evalue_infixe "(1+2)" = 3);
assert (evalue_infixe "(2-1)" = 1);
assert (evalue_infixe "(3*(1+2))" = 9);
assert(evalue_infixe "(1+((2+3)*(4*5)))" = 101);
"OK";;
```

Corrigé

```
let evalue_infixe s =
    let nb = creer_pile () in
    let op = creer_pile () in
    for i = 0 to String.length s - 1 do
        match s.[i] with
        | '(' -> ()
        | '0'..'9' ->
            empile nb (int_of_char s.[i] - int_of_char '0')
        | '*' | '-' | '/' | '+' -> empile op s.[i]
        | ')' -> begin
            let b = depile nb in
            let a = depile nb in
            match depile op with
            | '*' -> empile nb (a*b)
            | '-' -> empile nb (a-b)
            | '/' -> empile nb (a/b)
            | '+' -> empile nb (a+b)
            | _ -> failwith "unreachable"
        end
        | _ -> failwith "Invalid character"
    done;
    depile nb
```

20.2 File avec deux piles

On va implémenter une file avec deux piles.

On a vu qu'il suffisait de faire une `'a list ref` pour avoir une pile en `OCaml`, mais pour réaliser une file, comme on va changer les deux listes, on considère plutôt le type suivant :

```
type 'a file = {
    mutable entree : 'a list;
    mutable sortie : 'a list
}
```

Ainsi, chaque champ `mutable` sera une sorte de référence. On enfile en tête de la liste `entree` et on défile en tête de la liste `sortie` ; quand `sortie` est vide, on y transvase `entree` en la renversant.

✍ **Question 6** Écrire les fonctions suivantes :

- `cree_file : unit -> 'a file` qui crée une file vide ;
- `est_vide : 'a file -> bool` ;
- `enfile : 'a file -> 'a -> unit` ;
- `transvase : 'a file -> unit` qui, lorsque `sortie` est vide, y place les éléments de `entree` dans le bon ordre ;
- `defile : 'a file -> 'a` qui retire et renvoie l'élément le plus ancien.

Tests

```
let f = cree_file () in
Printf.printf "Test enfile: ";
assert(est_vide f);
enfile f 1;
enfile f 2;
```

```
enfile f 3;
Printf.printf "OK\n";
Printf.printf "Test transvase: ";
assert(f.entree = [3;2;1] && f.sortie = []);
transvase f;
Printf.printf "OK\n";
Printf.printf "Test defile: ";
assert(f.entree = [] && f.sortie = [1;2;3]);
assert(defile f = 1);
assert(defile f = 2);
enfile f 4;
assert(defile f = 3);
assert(defile f = 4);
assert(est_vide f);
Printf.printf "OK\n";;
```

Corrigé

```
let cree_file () : 'a file =
    {
        entree = [];
        sortie = []
    }

let est_vide (f:'a file) : bool =
    f.entree = [] && f.sortie = []

let enfile (f:'a file) (x:'a) : unit =
    f.entree <- x::f.entree

let transvase (f:'a file) : unit =
    assert(f.sortie = []);
    f.sortie <- List.rev f.entree;
    f.entree <- []

let defile (f : 'a file) : 'a =
    if f.sortie = []
    then transvase f;
    match f.sortie with
    | [] -> failwith "File vide"
    | t::q -> f.sortie <- q; t
```

Chaque élément est déplacé au plus une fois de `entree` vers `sortie` : une suite de n opérations coûte $O(n)$, soit un coût amorti constant par opération (voir le chapitre sur la complexité amortie).

CHAPITRE 21

Arbres

DANS CE CHAPITRE

21.1 Arbres binaires
21.2 Arbres
21.3 Parcours
21.4 Arbres binaires de recherche
21.5 Tas
21.6 Exercices

21.1 Arbres binaires

21.1.1 Définition inductive

Définition 21.1

Un arbre binaire étiqueté par $\mathscr{E}$ est :

- soit vide, et on le note alors nil ou $\perp$;
- soit un triplet (g, x, d) où $x \in \mathscr{E}$ et g et d sont des arbres binaires.

Remarque 21.2 Cette définition inductive ressemble à la définition des listes.

On peut la formaliser en introduisant $T_b(\mathscr{E})$ l'ensemble des arbres étiquetés par $\mathscr{E}$. C'est le **plus petit ensemble** tel que :

- $\text{nil} \in T_b(\mathscr{E})$;
- $\forall x \in \mathscr{E}, \forall g, d \in T_b(\mathscr{E}), (g, x, d) \in T_b(\mathscr{E})$.

Les conséquences de la précision **plus petit ensemble** sont importantes :

- les arbres sont nécessairement des expressions finies, c'est-à-dire qu'ils ne comportent qu'un nombre fini de constructions. C'est automatique car l'ensemble des arbres finis vérifie les conditions précédentes ;
- pour tout triplet (g, x, d) il ne peut exister qu'un arbre, sinon, en enlevant un arbre en double on vérifierait encore les deux conditions précédentes.

On en déduit directement la notion de preuve par **induction structurelle** sur les arbres :

$$\forall a \in T_b(\mathscr{E}), P(a) \Leftrightarrow \{P(\text{nil})$$
$$\forall x \in \mathscr{E}, \forall g, d \in T_b(\mathscr{E}), P(g) \wedge P(d) \Rightarrow P(g, x, d)$$

Cette induction correspond à un pseudo principe de récurrence qui nous permettra de faire les preuves.

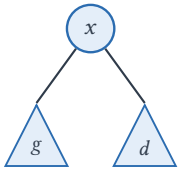
Pour démontrer cette équivalence, il suffit de vérifier que $T_P = \{t \in T_b(\mathscr{E}), P(t)\}$ vérifie les relations de la définition inductive ; comme $T_P \subset T_b(\mathscr{E})$, par minimalité, on a bien $T_P = T_b(\mathscr{E})$.

Exemple 21.3

Si $\mathscr{E} = \mathbb{N}$, les éléments suivants sont des arbres :

- nil
- $(\text{nil}, 2, \text{nil})$
- $((\text{nil}, 1, \text{nil}), 2, \text{nil})$

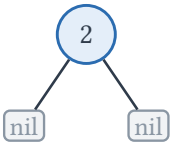
On adoptera une représentation graphique très naturelle pour les arbres binaires où un nœud (g, x, d) sera représenté par :

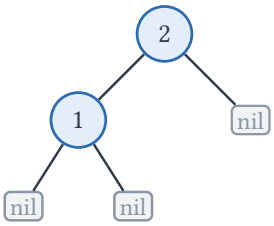


Exemple 21.4

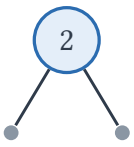
Les trois arbres précédents sont donc représentés par :

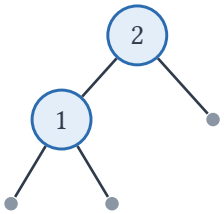
nil






```
Afin d'alléger la notation, on omettra $\text{nil}$, sauf pour l'arbre vide.
On fera cependant attention à conserver les arêtes donnant sur $\text{nil}$
pour ne pas confondre $(\text{nil}, 1, (\text{nil}, 2, \text{nil}))$ et
$( (\text{nil}, 2, \text{nil}), 1, \text{nil} )$. Ainsi on représentera les
arbres précédents par :
```





21.1.2 Vocabulaire

Définition 21.5

Soit a un arbre binaire. Les arbres non vides présents dans a sont appelés les **nœuds** de a. Parmi ceux-ci on distingue ceux qui sont de la forme (nil, x, nil), appelés des **feuilles**. Les nœuds qui ne sont pas des feuilles sont appelés des **nœuds internes**. Le nœud a lui-même est appelé la **racine** de l'arbre. On note $N(a)$ l'ensemble des nœuds de a.

Si $n \in N(a)$ n'est pas la racine, il est le fils gauche ou le fils droit d'un unique nœud appelé le **père** de n.

Si $n \in N(a)$, on appelle **sous-arbre** de a l'arbre dont n est la racine.

Remarque 21.6 En partant de la définition inductive, il y a une identification entre un sous-arbre et sa racine. Mais afin de raisonner, on distinguera un nœud en tant qu'emplacement au sein d'un arbre et le sous-arbre lui-même.

Définition 21.7

Soit a un arbre binaire.

- On appelle **taille** de a, et on note $|\, a \,|$, le nombre de nœuds de a.
- On appelle **hauteur** de a l'entier

$$h(a) = \begin{cases} -1 & \text{si } a = \text{nil} \\ 1 + \max(h(g), h(d)) & \text{si } a = (g, x, d) \end{cases}$$

Définition 21.8

Soit a un arbre binaire et $n \in N(a)$. On appelle **profondeur** de n l'unique entier $p(n)$ tel qu'il existe une suite finie $(n_0, n_1, \ldots, n_{p(n)})$ de nœuds de a vérifiant :

- n_0 est la racine de a ;
- $n_{p(n)} = n$;
- pour tout $i < p(n)$, n_i est le père de n_{i+1}.

Cette suite finie est le **chemin** de la racine à n. Il est nécessairement unique car chaque nœud autre que la racine a un unique père.

Ainsi, la racine est de profondeur 0, ses fils de profondeur 1, etc. L'ensemble des nœuds de profondeur p est appelé le **niveau** p de l'arbre.

Théorème 21.9

Soit a un arbre ; si a est non vide, alors $h(a) = \max_{n \in N(a)} p(n)$.

Preuve Par induction structurelle sur a.

- **Initialisation** Si $a = \text{nil}$ la prémisse est fausse, donc l'implication est trivialement vérifiée.
- **Hérédité** Supposons la propriété vérifiée pour deux arbres g et d, soit $x \in \mathscr{C}$, on va montrer qu'elle est vérifiée pour $a = (g, x, d)$. On a quatre cas pour le couple (g, d) :

- Soit $g \neq \text{nil}$ et $d \neq \text{nil}$. Dans ce cas, par hypothèse $h(g) = \max_{n \in N(g)} p_g(n)$ où $p_g(n)$ est la profondeur de n en tant que nœud de l'arbre g. Or, mis à part la racine de a, le chemin menant dans a au nœud n est dans g. On a donc directement $p_g(n) = p(n) - 1$. Ainsi $h(g) = \max_{n \in N(g)} p(n) - 1$. De même, $h(d) = \max_{n \in N(d)} p(n) - 1$. On a

$$h(a) = 1 + \max(h(g), h(d)) = \max\left(\max_{n \in N(g)} p(n), \max_{n \in N(d)} p(n)\right)$$

  Or, le seul nœud de a qui n'est ni dans g ni dans d est sa racine, qui est de profondeur nulle donc $h(a) = \max_{n \in N(a)} p(n)$.
- Soit $g = \text{nil}$ et $d \neq \text{nil}$. Ainsi $h(g) = -1$ et donc $h(a) = 1 + h(d) = \max_{n \in N(d)} p(n)$ par l'analyse précédente. On conclut donc avec la propriété voulue.
- Soit $g \neq \text{nil}$ et $d = \text{nil}$. Cas symétrique du précédent.
- Soit $g = d = \text{nil}$. Auquel cas, $h(a) = 0$ qui est bien la profondeur de son unique nœud.

On a bien montré la propriété voulue par induction structurelle.

□

La taille et la hauteur d'un arbre binaire sont liées par l'encadrement suivant, qu'on utilisera constamment pour estimer la complexité des opérations sur les arbres.

Théorème 21.10

Pour tout arbre binaire a, on a $h(a) + 1 \leq |\, a \,| \leq 2^{h(a)+1} - 1$.

De manière équivalente, si a est non vide : $\log_2(|\, a \,| + 1) - 1 \leq h(a) \leq |\, a \,| - 1$.

Preuve Par induction structurelle sur a.

- Si $a = \text{nil}$: $h(a) + 1 = 0 = |\, a \,| = 2^0 - 1$.
- Si $a = (g, x, d)$ avec g et d vérifiant l'encadrement. Quitte à échanger les rôles, on suppose $h(g) \geq h(d)$ de sorte que $h(a) = 1 + h(g)$. On a alors $|\, a \,| = 1 + |\, g \,| + |\, d \,| \geq 1 + |\, g \,| \geq 1 + h(g) + 1 = h(a) + 1$, et $|\, a \,| = 1 + |\, g \,| + |\, d \,| \leq 1 + (2^{h(g)+1} - 1) + (2^{h(d)+1} - 1) \leq 2 \times 2^{h(g)+1} - 1 = 2^{h(a)+1} - 1$.

L'équivalence s'obtient en passant au $\log_2$ dans $|\, a \,| + 1 \leq 2^{h(a)+1}$.

□

Remarque 21.11 Les deux bornes sont atteintes : la borne $|\, a \,| = h(a) + 1$ pour les arbres dont chaque nœud a au plus un fils non vide (des *peignes*, qui sont essentiellement des listes), la borne $|\, a \,| = 2^{h(a)+1} - 1$ pour les arbres dont tous les niveaux sont pleins (les arbres *parfaits* définis plus bas). Un arbre de taille n a donc une hauteur comprise entre environ $\log_2 n$ et $n - 1$.

21.1.3 Implémentations

En `OCaml`

En `OCaml`, on traduit directement la définition inductive par un type récursif :

```
type 'a arbre = Noeud of 'a arbre * 'a * 'a arbre | Nil
```

On pourra alors définir des fonctions récursives sur les arbres par induction structurelle à l'aide d'un filtrage.

```
let rec taille a = match a with
    | Nil -> 0
    | Noeud(g,x,d) -> 1 + taille g + taille d

let rec hauteur a = match a with
    | Nil -> -1
    | Noeud(g, x, d) -> 1 + max (hauteur g) (hauteur d)
```

En C

En C, un nœud d'un arbre binaire a la même structure qu'un maillon d'une liste doublement chaînée : une valeur et deux pointeurs vers d'autres nœuds, nommés ici `gauche` et `droite`. L'arbre vide est représenté par le pointeur `NULL`.

```c
struct noeud {
    struct noeud *gauche;
    struct noeud *droite;
    int valeur;
};

typedef struct noeud *arbre;

int taille(arbre a)
{
    if (a == NULL)
        return 0;
    return 1 + taille(a->gauche) + taille(a->droite);
}
```

Les fonctions de manipulation de ce type sont étudiées dans le TP sur les arbres.

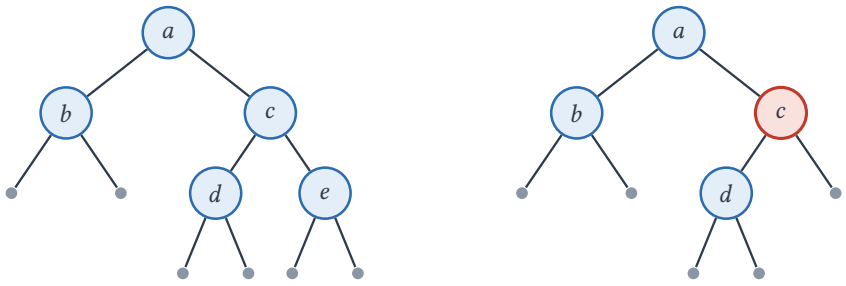
21.1.4 Arbres binaires stricts

Définition 21.12

Un arbre binaire est dit **strict** lorsqu'aucun de ses nœuds n'a exactement un fils vide.

Cela revient à dire que seules les feuilles ont des fils vides. On peut donc omettre l'arbre vide et ne garder que les feuilles à la place.

Remarque 21.13 L'arbre de gauche ici est un arbre binaire strict alors que ce n'est pas le cas de l'arbre de droite (le nœud c n'a qu'un fils non vide) :



binaire strict *non strict*

Théorème 21.14

Soit a un arbre binaire strict non vide. Si a a n nœuds internes, alors il a $n+1$ feuilles.

Preuve Par induction structurelle sur les arbres stricts non vides. On note $n_i(t)$ le nombre de nœuds internes et $n_f(t)$ le nombre de feuilles de l'arbre t.

- **Initialisation** Si a est une feuille, il a 0 nœud interne et 1 feuille : on a directement la relation.
- **Hérédité** Si $a = (g, x, d)$ n'est pas une feuille, comme a est strict, g et d sont tous deux non vides, et ce sont des arbres stricts. On suppose la propriété vraie pour g et d. On a $n_i(a) = 1 + n_i(g) + n_i(d)$ et $n_f(a) = n_f(g) + n_f(d) = (n_i(g) + 1) + (n_i(d) + 1) = 1 + n_i(a)$. La propriété est démontrée pour a.

□

Cela permet naturellement de considérer des arbres où les feuilles et les nœuds internes ont deux types différents d'étiquettes.

L'exemple classique d'un tel arbre est celui des expressions arithmétiques :

- les nœuds internes sont étiquetés par des opérateurs binaires ;
- les feuilles sont étiquetées par des nombres.

En `OCaml`, on peut ainsi représenter un tel arbre par le type récursif :

```
type ('a, 'b) arbre_bin = Feuille of 'a
    | Noeud of ('a, 'b) arbre_bin * 'b * ('a, 'b) arbre_bin
```

Remarque 21.15 On ne peut plus représenter l'arbre vide avec ce type.

Et pour les expressions, on pourra ainsi définir un type pour les opérateurs et écrire :

```
type operateur = Plus | Times | Minus | Divides

let e = Noeud(Feuille 2, Plus, Noeud(Feuille 3, Times, Feuille 4))
```

L'évaluation de telles expressions est étudiée dans le TP sur les arbres.

21.1.5 Arbres binaires complets

Définition 21.16

Un arbre binaire dont tous les niveaux sont pleins sauf éventuellement le dernier est dit **complet**. Si son dernier niveau est également plein, on dit qu'il est **parfait**.

Si, de plus, les nœuds du dernier niveau sont tous placés le plus à gauche possible, on dit que l'arbre est **complet à gauche**. C'est le cas de l'exemple ci-dessous, et c'est cette hypothèse qui permet la représentation dans un tableau.

On peut représenter un arbre complet à gauche dans un tableau niveau par niveau en partant de la racine.

Si on place les indices comme étiquettes on aura, par exemple :

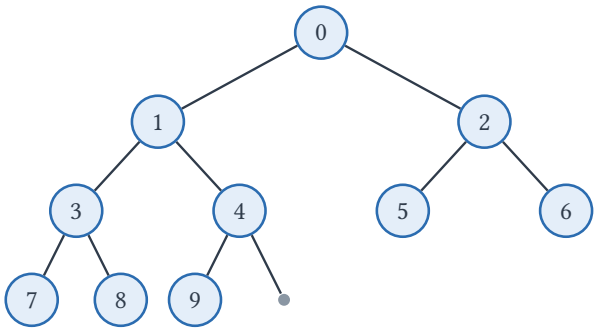


Figure 21.1 — Arbre complet à gauche dont chaque nœud est étiqueté par son indice.

Et on pourra donc le représenter par le tableau des étiquettes dans cet ordre d'énumération.

On remarque que :

- le nœud 0 est toujours la racine ;
- si on considère un nœud i, son fils gauche est le nœud $2i+1$ et son fils droit est le nœud $2i+2$;
- le fils gauche de $i+1$ est $2(i+1)+1 = 2i+3$: il vient bien directement après le fils droit de i ;
- le k-ième nœud du niveau l, en commençant à numéroter à 0, a pour indice $\sum_{j=0}^{l-1} 2^j + k = 2^l - 1 + k$;
- si $i > 0$, son père est le nœud $\left\lfloor \frac{i-1}{2} \right\rfloor$.

On peut donc, avec cette représentation, manipuler assez simplement un arbre binaire **à plat**.

Si le nœud de numéro i a l'étiquette e_i, le tableau a la structure suivante :

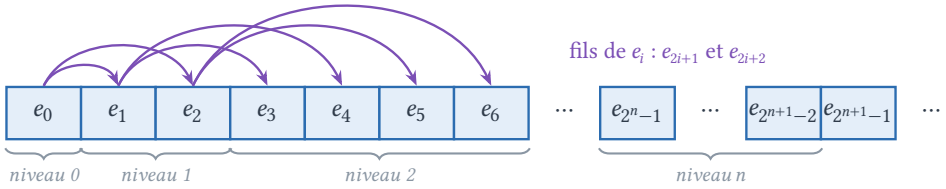


Figure 21.2 — Arbre complet stocké à plat : les fils de e_i sont e_{2i+1} et e_{2i+2}.

21.2 Arbres

21.2.1 Définition

On étend directement la définition inductive des arbres binaires au cas où les nœuds ont un nombre quelconque de fils.

Définition 21.17

Un arbre étiqueté par $\mathscr{E}$ est un couple (x, f) où $x \in \mathscr{E}$ et f est une suite **finie** d'arbres.

On étend naturellement le vocabulaire des arbres binaires :

- le nœud de tête d'un arbre, en fin de compte sa *valeur*, est appelé sa racine ;
- les éléments de f sont appelés les fils du nœud ;
- si f est vide, on dit que le nœud est une **feuille**, sinon, on dit que c'est un nœud interne ;
- $|\, f \,|$, le nombre d'éléments de f, est appelé l'**arité** du nœud. Les feuilles sont donc les nœuds d'arité nulle ;
- une suite finie d'arbres, ou plus généralement un ensemble d'arbres, est appelé une **forêt**.

Si on note $T(\mathscr{E})$ l'ensemble des arbres étiquetés par $\mathscr{E}$, on a

$$\forall x \in \mathscr{E}, \forall n \in \mathbb{N}, \forall a_1, ..., a_n \in T(\mathscr{E}), (x, (a_1, ..., a_n)) \in T(\mathscr{E})$$

Attention contrairement à ce que peut laisser entendre la formule précédente, il est tout à fait possible que $n = 0$ ce qui correspond à la suite finie vide et donc à une feuille. On perd ici la possibilité de représenter un arbre vide.

Remarque 21.18 La taille, la hauteur et la profondeur se définissent comme pour les arbres binaires : $|\,(x, (a_1, ..., a_n))\,| = 1 + \sum_{i=1}^{n} |\, a_i \,|$ et $h(x, (a_1, ..., a_n)) = 1 + \max(-1, h(a_1), ..., h(a_n))$. Une feuille est ainsi de hauteur 0, ce qui est cohérent avec la convention $h(\text{nil}) = -1$ des arbres binaires.

Remarque 21.19 Un arbre binaire non vide peut être vu comme un arbre dont les nœuds sont d'arité au plus deux. On remarque en disant cela qu'on perd une information : la position gauche ou droite de l'unique fils d'un nœud unaire.

Comme pour les arbres binaires stricts, on pourra être amenés à considérer des arbres dont les nœuds internes et les feuilles sont étiquetés par des ensembles différents.

21.2.2 Implémentation

L'implémentation d'un arbre repose sur l'implémentation d'une suite finie. Celle-ci peut être faite par un tableau ou par une liste chaînée.

On retrouve alors des implémentations différentes des arbres mais qui sont, au fond, très proches.

En `OCaml`, on pourra alors avoir les deux implémentations suivantes (à écrire dans deux fichiers différents, car elles utilisent les mêmes noms de champs) :

```
type 'a arbre_l = { etiquette : 'a; enfants : 'a arbre_l list }
type 'a arbre_a = { etiquette : 'a; enfants : 'a arbre_a array }
```

On va alors avoir des programmes à la présentation assez différente suivant le type choisi. Voici, par exemple, les deux implémentations du calcul de la taille d'un arbre.

Avec des listes et aucune fonction du module `List`, on écrit souvent une fonction récursive sur les arbres et une fonction récursive sur les forêts.

```
let rec taille_arbre a =
    1 + taille_foret a.enfants
    and taille_foret l = match l with
        | [] -> 0
        | t::q -> taille_arbre t + taille_foret q
```

On peut aussi utiliser directement `fold_left` et `map` :

```
let rec taille_arbre a =
    1 + List.fold_left (+) 0 (List.map taille_arbre a.enfants)
```

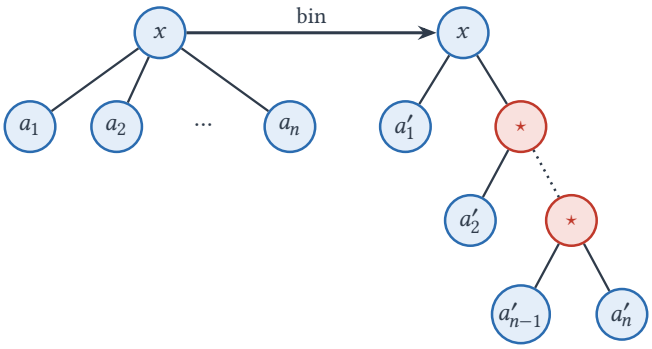
Avec l'autre représentation, on procédera avec un mélange de fonction récursive et de boucles. Cela n'est pas sans rappeler ce qui a été fait pour le *backtracking*.

```
let rec taille_arbre a =
    let t = ref 1 in
    for i = 0 to Array.length a.enfants - 1 do
        t := !t + taille_arbre a.enfants.(i)
    done;
    !t
```

21.2.3 Représentation par un arbre binaire

Pour des raisons d'efficacité ou de réutilisation de programmes existants, on peut vouloir représenter un arbre comme un arbre binaire.

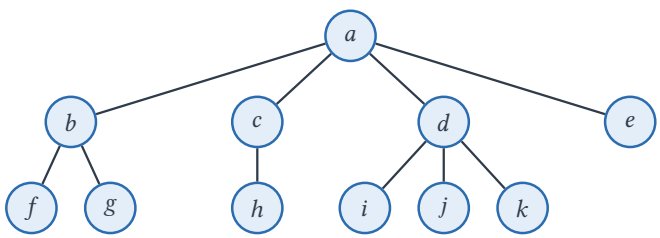
On va définir une application $\text{bin} : T(\mathscr{E}) \to T_b(\mathscr{E} \cup \{\star\})$ ainsi : $\text{bin}\,(x, ()) = (\text{nil}, x, \text{nil})$, $\text{bin}\,(x, (a)) = (\text{bin}\,(a), x, \text{nil})$ et pour un nœud d'arité strictement plus grande que 1 :



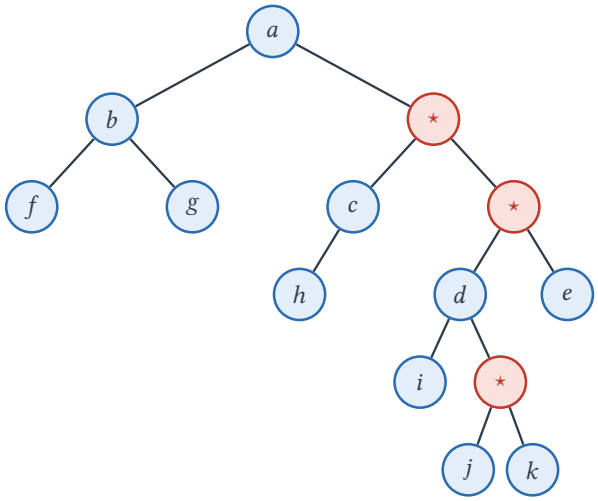
où $a'_i = \text{bin}\,(a_i)$. Les nœuds rouges sont des nœuds ayant une étiquette spéciale $\star$ telle que $\star \notin \mathscr{E}$.

Exemple 21.20

L'arbre suivant :



sera alors représenté par l'arbre binaire :



21.2.4 Applications

Arbres d'expressions

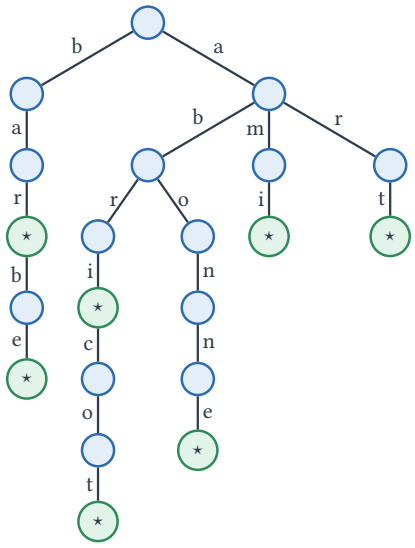
On a déjà vu l'utilisation des arbres binaires pour représenter des expressions arithmétiques avec des opérateurs binaires. On généralise naturellement cela à des expressions dont les opérateurs sont d'arité quelconque.

C'est ainsi qu'on représente un programme en mémoire, on parle d'arbre syntaxique.

Arbres préfixes ou tries

Un *arbre préfixe* ou *trie* est un arbre permettant de représenter un ensemble de mots. On étiquette les arêtes par des caractères et les nœuds par un booléen indiquant si la suite des étiquettes qui mène de la racine à ce nœud est un mot (ce qu'on représente par ⋆).

Par exemple, le trie :



permet de représenter l'ensemble de mots : bar, barbe, art, ami, abri, abricot, abonne.

Afin de représenter un tel arbre, il est nécessaire d'avoir une étiquette sur les arêtes, le plus simple pour cela est de remplacer la liste des enfants par une liste de couples (étiquette, enfant) ainsi :

```
type trie = {
    mot : bool;
```

```
    enfants : (char * trie) list
    }
```

Les fonctions de manipulation de ce type de donnée sont étudiées dans le TP sur les arbres.

21.3 Parcours

21.3.1 Parcours récursif d'un arbre binaire

On a déjà vu dans les paragraphes précédents ce qui constitue le cœur du parcours d'un arbre :

```
let rec parcours a =
    match a with
    | Nil -> ()
    | Noeud(g, x, d) ->
        parcours g;
        parcours d
```

Cette fonction ne fait rien, mais elle va parcourir chacun des nœuds de l'arbre. Si on veut effectuer un traitement sur le nœud, on peut le faire à trois moments :

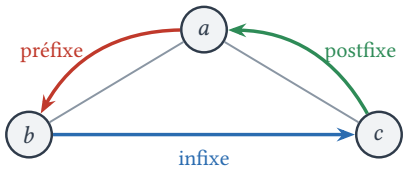


Figure 21.3 — Les trois moments où l'on peut traiter un nœud : préfixe, infixe et postfixe.

- avant l'appel à `parcours g`, on parle de *traitement préfixe* ;
- entre les deux appels, on parle de *traitement infixe* ;
- après l'appel à `parcours d`, on parle de *traitement postfixe*.

On représente ici l'ordre dans lequel on va effectuer chacun de ces trois traitements selon le code couleur : rouge pour préfixe, bleu pour infixe et vert pour postfixe. On a également indiqué, pour les premiers traitements, l'ordre dans lequel ils sont effectués par des numéros.

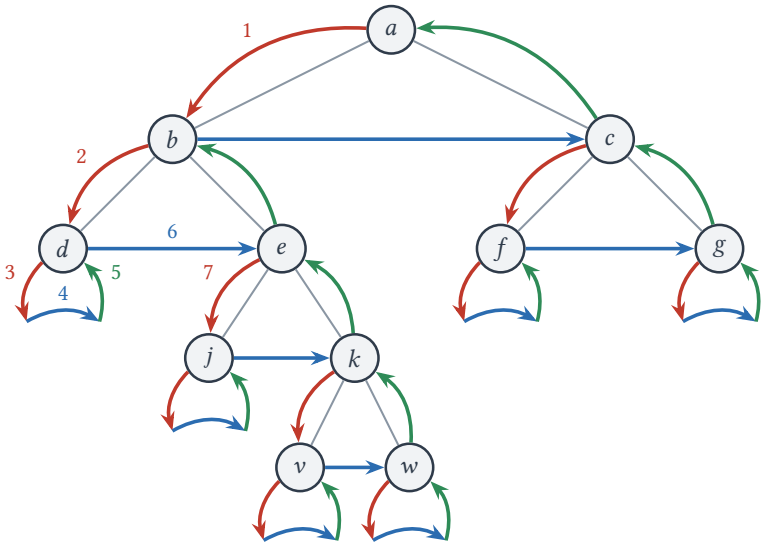


Figure 21.4 — Ordre des traitements préfixes (rouge), infixes (bleu) et postfixes (vert) au cours du parcours.

Pour illustrer ces traitements, on peut rajouter trois arguments au parcours :

```
let rec parcours prefixe infixe postfixe a =
    match a with
    | Nil -> ()
    | Noeud(g, x, d) ->
        prefixe a;
        parcours prefixe infixe postfixe g;
        infixe a;
        parcours prefixe infixe postfixe d;
        postfixe a
```

On peut, par exemple, définir les deux fonctions suivantes :

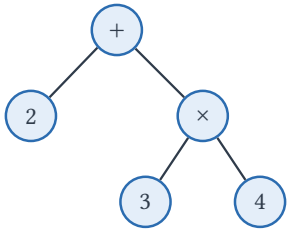
```
let idle a = () (* ne fait rien *)

let print a = match a with
    | Nil -> failwith "Vide"
    | Noeud(_,x,_) -> print_char x
```

On peut alors observer les nœuds affichés sur l'arbre précédent selon les traitements effectués :

- `parcours print idle idle a` : `abdejkvwcfg`, on suit uniquement les arêtes rouges ;
- `parcours idle print idle a` : `dbjevkwafcg`, on suit uniquement les arêtes bleues ;
- `parcours idle idle print a` : `djvwkebfgca`, on suit uniquement les arêtes vertes ;
- `parcours print print print a` : `abdddbejjjekvvvkwwwkebacfffcgggca`, on suit toutes les arêtes.

Remarque 21.21 Si on considère un arbre d'expression comme :




```
On va alors obtenir en affichant à chaque traitement la suite :
$+222+\times333\times444\times +$. Chaque nombre $n$ étant une feuille,
on va observer l'affichage de $nnn$ qu'on peut réduire à $n$. Pour les
opérateurs, on remplace le premier affichage par $($ et le dernier par
$)$. En procédant ainsi, on affiche $(2+(3\times 4))$ et on a ainsi
retrouvé l'écriture bien parenthésée de l'expression.
```

Définition 21.22

L'ordre dans lequel on effectue un traitement *préfixe* dans un parcours sur les nœuds d'un arbre est appelé l'ordre *préfixe* sur les nœuds.

On définit, de même, l'ordre *infixe* et l'ordre *postfixe*.

21.3.2 Parcours impératifs

Cadre général

On va considérer une structure de donnée abstraite qui généralise les piles et les files et permet de représenter un ensemble de tâches à traiter. Pour cela, on dispose d'un type paramétrique `'a t` et de l'interface :

- `cree : unit -> 'a t` crée un ensemble de tâches ;
- `ajoute : 'a t -> 'a -> unit` ajoute une nouvelle tâche à traiter ;
- `retire : 'a t -> 'a` retire une tâche de l'ensemble des tâches à traiter ;
- `est_vide : 'a t -> bool` renvoie un booléen indiquant si l'ensemble de tâches est vide.

On a ainsi vu, dans le chapitre sur les piles et les files, deux implémentations possibles, qui sont elles-mêmes des structures abstraites mais un peu moins abstraites que celle-ci :

- les piles pour lesquelles on retire le dernier élément ajouté ;
- les files pour lesquelles on retire l'élément le plus anciennement ajouté.

On peut écrire un parcours impératif générique :

```
let parcours traitement a =
    let avisiter = cree () in
    ajoute avisiter a;
    while not (est_vide avisiter) do
        let a = retire avisiter in
        match a with
        | Nil -> ()
        | Noeud(g, _, d) -> traitement a;
                            ajoute avisiter g;
                            ajoute avisiter d
    done
```

Cas du parcours en profondeur

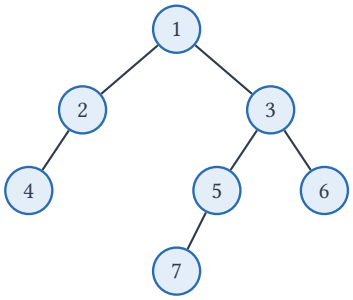
Dans le cas où on utilise une pile, on retrouve essentiellement le parcours récursif précédent avec quelques changements :

- on empile ici d'abord `g` puis `d`, donc on va à droite avant d'aller à gauche. Il suffit de permuter les deux ajouts pour retrouver l'ordre précédent ;
- on effectue un unique traitement préfixe ;
- si on considère la pile d'appels récursifs du parcours récursif, on constate que sa taille est majorée par $h(a) + 2$. Ici, la pile explicite reste également de taille $O(h(a))$: quand on dépile un arbre de profondeur p, on empile ses deux fils de profondeur $p + 1$, si bien que la pile contient, à tout moment, au plus un arbre de chaque profondeur, sauf pour la plus grande profondeur présente qui peut en contenir deux. Pour un arbre d'arité quelconque, on empile en revanche tous les fils d'un nœud et la pile peut contenir jusqu'à (arité maximale) × (hauteur) éléments.

On parle de **parcours en profondeur** ou *depth-first search* (DFS) en anglais.

Exemple 21.23

Si on considère l'arbre suivant :




```
En ignorant les arbres vides, par exemple en ne les ajoutant pas,
on va avoir le déroulement suivant du parcours (le sommet de la pile est
à droite) :

| avisiter                | étiquette traitée |
|-------------------------|-------------------|
| $1 \rightarrow$         |                   |
| $2, 3 \rightarrow$      | $1$               |
| $2, 5, 6 \rightarrow$   | $3$               |
| $2, 5 \rightarrow$      | $6$               |
| $2, 7 \rightarrow$      | $5$               |
```

```
| $2 \rightarrow$         | $7$              |
| $4 \rightarrow$         | $2$              |
|                         | $4$              |
```

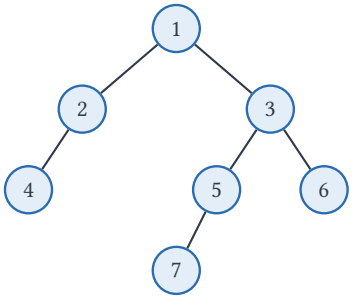
Cas du parcours en largeur

Dans le cas où on utilise une file, on obtient un parcours appelé le **parcours en largeur** ou *breadth-first search* (BFS) en anglais.

Dans ce parcours, on va visiter les nœuds *niveau par niveau* en partant de la racine jusqu'aux feuilles de plus grande profondeur.

Exemple 21.24

Si on considère l'arbre suivant :




```
En ignorant les arbres vides, par exemple en ne les ajoutant pas,
on va avoir le déroulement suivant du parcours (on ajoute à gauche et on
retire à droite) :

| avisiter                              | étiquette traitée |
|---------------------------------------|-------------------|
| $\rightarrow 1 \rightarrow$           |                   |
| $\rightarrow 3, 2 \rightarrow$        | $1$               |
| $\rightarrow 4, 3 \rightarrow$        | $2$               |
| $\rightarrow 6, 5, 4 \rightarrow$     | $3$               |
| $\rightarrow 6, 5 \rightarrow$        | $4$               |
| $\rightarrow 7, 6 \rightarrow$        | $5$               |
| $\rightarrow 7 \rightarrow$           | $6$               |
|                                       | $7$               |
```

Ce parcours est particulièrement intéressant quand on cherche une information avec la plus petite profondeur possible. Par exemple, dans le cas de la recherche d'une solution à un problème, on peut vouloir tester les petites solutions avant les plus grandes.

Pour des problèmes d'énumération, c'est également intéressant, car on va obtenir des éléments par ordre croissant de la longueur du chemin depuis la racine. Par exemple, pour les tries, on en déduit les mots par ordre croissant de longueur.

21.3.3 Parcours d'arbres

Pour des arbres, on va suivre les mêmes principes. La différence va se situer au niveau de l'implémentation car il faudra alors manipuler des listes ou des tableaux d'enfants.

Notons qu'on a déjà vu un tel problème quand on a résolu des problèmes par *backtracking*. En effet, avec le *backtracking*, on a un arbre implicite des positions partielles dont les enfants sont les mouvements possibles vers de nouvelles positions.

21.4 Arbres binaires de recherche

21.4.1 Objectif

On souhaite ici réaliser une structure efficace d'ensembles finis. Pour cela, on cherche à définir une structure immuable `'a ensemble` munie de cinq opérations :

- `ensemble_vide : 'a ensemble` l'ensemble vide ;
- `ajoute : 'a ensemble -> 'a -> 'a ensemble` rajoute un élément à un ensemble et renvoie le nouvel ensemble ;
- `supprime : 'a ensemble -> 'a -> 'a ensemble` retire un élément à un ensemble et renvoie le nouvel ensemble ;
- `contient : 'a ensemble -> 'a -> bool` teste si l'ensemble contient un élément ;
- `cardinal : 'a ensemble -> int` renvoie le nombre d'éléments de l'ensemble.

Notons qu'on pourra souvent relâcher la contrainte naturelle des ensembles en permettant d'ajouter plusieurs fois un même élément.

Une implémentation possible de cette structure serait d'utiliser des listes ou des tableaux triés, on aurait alors des opérations en $O(n)$ pour manipuler n éléments (seul le test d'appartenance passe en $O(\log_2 n)$ dans un tableau trié, grâce à une recherche dichotomique).

Ici, on présente une implémentation rendant possible une complexité en $O(\log_2 n)$ pour chaque opération sous certaines hypothèses dont on montrera qu'elles peuvent être satisfaites dans un second temps.

21.4.2 Définition

Définition 21.25

On dit qu'un arbre a étiqueté par X, muni d'une relation d'ordre total, est un **arbre binaire de recherche** (ABR) lorsque :

- soit $a = \text{nil}$;
- soit $a = (g, x, d)$ où g et d sont des arbres binaires de recherche et de plus

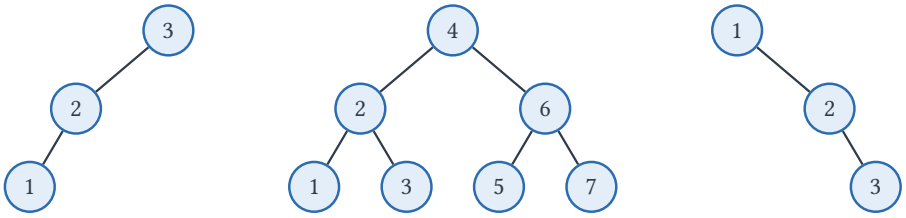
$$\max_{y \in N(g)} e(y) \leq x \leq \min_{y \in N(d)} e(y)$$

où $e(y)$ est l'étiquette du nœud y (la condition portant sur g, resp. d, est vide si g, resp. d, est vide).

Remarque 21.26

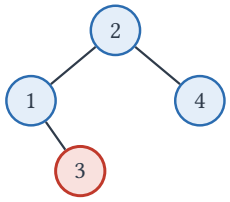
- Tout sous-arbre d'un arbre binaire de recherche est un arbre binaire de recherche.
- Pour un ensemble donné d'étiquettes il n'y a pas unicité de l'arbre binaire de recherche : voir ci-après.

Exemples d'ABR pour les ensembles $\{1, 2, 3\}$ et $\{1, \ldots, 7\}$:



$\{1, 2, 3\}$ $\{1, \ldots, 7\}$ $\{1, 2, 3\}$

Remarque 21.27 La propriété caractéristique des ABR n'est pas locale, c'est-à-dire que tester si chaque nœud a une étiquette supérieure à son fils gauche et inférieure à son fils droit ne suffit pas. Dans l'arbre suivant, 3 est dans le sous-arbre gauche de 2 :



21.4.3 Opérations

Minimum et maximum

On peut obtenir la valeur minimale et maximale d'un ABR en effectuant un parcours le long de la branche la plus à gauche ou de la branche la plus à droite.

En effet, si l'arbre est (g, x, d) : soit g est vide et $x \leq \min d$ est le minimum, soit g est non vide et on a alors $\min g \leq \max g \leq x$ donc le minimum de g est celui de l'arbre.

On en déduit le programme suivant :

```
let rec minimum a =
    match a with
    | Nil -> failwith "vide"
    | Noeud(Nil, x, _) -> x
    | Noeud(g, _, _) -> minimum g
```

On remarque que pour parcourir cette branche, on est en $O(h(a))$ où a est l'arbre considéré.

Test si un arbre est un ABR

Pour tester si un arbre est bien un arbre binaire de recherche on doit connaître la valeur maximum de l'arbre gauche et minimum de l'arbre droit et tester si l'étiquette de chaque nœud n vérifie

$$\max_{\eta \in A_g} e(\eta) \leq e(n) \leq \min_{\eta \in A_d} e(\eta)$$

On peut écrire une procédure qui indique si l'arbre est un arbre binaire de recherche et les valeurs minimum et maximum de l'arbre (puisqu'on doit les calculer...).

```
let rec abr_test a = match a with
| Nil -> failwith "Arbre vide"
| Noeud(Nil,n,Nil) -> (true, n, n)
| Noeud(Nil,n,d) -> let bd, mind, maxd = abr_test d in
    (bd && n <= mind, min n mind, max n maxd)
| Noeud(g,n,Nil) -> let bg, ming, maxg = abr_test g in
    (bg && maxg <= n, min n ming, max n maxg)
| Noeud(g,n,d) ->
    let bg, ming, maxg = abr_test g in
    let bd, mind, maxd = abr_test d in
    (bg && bd && maxg <= n && n <= mind,
     min n (min ming mind), max n (max maxg maxd))
```

Ici on a fait le choix de ne rien associer à l'arbre vide et de le traiter en amont au niveau des nœuds. Cela permet de ne pas imposer de contraintes sur le type des étiquettes comme cela aurait été le cas en mettant des valeurs *ad hoc* dans le cas de l'arbre vide (typiquement $\pm\infty$).

Remarque 21.28 On rajoute des fonctions permettant de manipuler facilement ces données et on obtient alors un algorithme de test plus concis :

```
let valeur a = match a with
    | None -> failwith "None n'a pas de valeur"
    | Some x -> x

let opt_bin f a b = match a, b with
    | None, _ -> b
```

```
    | _, None -> a
    | Some a, Some b -> Some (f a b)

let omax = opt_bin max
let omin = opt_bin min

let abr_test a =
    let rec aux a = match a with
        | Nil -> (true, None, None)
        | Noeud(g,n,d) ->
            let bg, ming, maxg = aux g in
            let bd, mind, maxd = aux d in
            (bg && bd && (maxg = None || valeur maxg <= n)
                && (mind = None || n <= valeur mind),
            omin (Some n) (omin ming mind),
            omax (Some n) (omax maxg maxd))
    in let est_abr, min_abr, max_abr = aux a in
    (est_abr, valeur min_abr, valeur max_abr)
```

Chaque nœud étant traité une fois en temps constant, ce test est en $O(|\,a\,|)$. On verra en exercice qu'une lecture infixe permet de conclure également.

Recherche d'un élément

Comme leur nom l'indique, la recherche est adaptée à ce type d'arbres. On peut construire la fonction de recherche :

```
let rec recherche x a = match a with
  |Nil -> false
  |Noeud(gauche,n,droite) when x=n -> true
  |Noeud(gauche,n,droite) when x>n -> recherche x droite
  |Noeud(gauche,n,droite)  -> recherche x gauche
```

Elle est correcte : si $x > n$, x ne peut pas être une étiquette de `gauche` dont toutes les étiquettes sont $\leq n$, il suffit donc de le chercher dans `droite` (et symétriquement si $x < n$). Elle suit une branche, d'où une complexité en $O(h(a))$.

Si l'ABR est construit par insertions successives (voir ci-dessous) des éléments pris dans un ordre aléatoire uniforme, on peut montrer (résultat admis) que sa hauteur moyenne est un $O(\log_2 n)$. Ce modèle aléatoire est étudié dans le TP (deuxième modèle).

Insertion d'un élément

Étant donné un arbre binaire de recherche a et une étiquette x, on veut construire un nouvel arbre binaire de recherche contenant les étiquettes des nœuds de a et x. Il n'y a bien sûr pas unicité de la solution… Dans un arbre binaire de recherche, l'opération se décompose en une étape de recherche, qui renvoie des informations sur la place où doit être inséré le nouvel élément, suivie de l'adjonction proprement dite.

```
let rec insere x a = match a with
  | Nil -> Noeud(Nil,x,Nil)
  | Noeud(gauche,n,droite) when x>n -> Noeud(gauche,n,insere x droite)
  | Noeud(gauche,n,droite) -> Noeud(insere x gauche,n,droite)
```

On compare x à la racine pour déterminer s'il faut l'ajouter dans le sous-arbre gauche ou le sous-arbre droit, et l'on rappelle la procédure récursivement. Le dernier appel récursif se fait sur un arbre vide et l'on crée alors à cette place le nœud d'étiquette x. Le nouvel arbre est reconstruit de proche en proche depuis cette branche vers la racine, sans modifier les branches non explorées. Les anciens nœuds ne sont pas modifiés et l'ancien arbre reste donc disponible : c'est une structure **immuable** (ou persistante), qui partage avec l'ancien arbre tous les sous-arbres non explorés et ne crée que $O(h(a))$ nouveaux nœuds.

Propriété 21.29

Si a est un ABR, `insere x a` est un ABR dont les étiquettes sont celles de a et x (en tant que multiensemble). Le calcul est en $O(h(a))$.

Preuve Par induction structurelle sur a. Si $a = \text{nil}$, on renvoie $(\text{nil}, x, \text{nil})$ qui convient. Si $a = (g, n, d)$ et $x > n$, par hypothèse d'induction `insere x d` est un ABR d'étiquettes celles de d et x ; elles sont toutes $\geq n$ (celles de d par hypothèse, et $x > n$), et g est inchangé : $(g, n,$ `insere x d`$)$ est bien un ABR. Le cas $x \leq n$ est symétrique. Chaque appel descend d'un niveau et fait un travail constant, d'où le $O(h(a))$.

□

Construction par insertion successive de l'ABR pour les éléments 5, 9, 1, 20, 4, 7, 17, 12 :

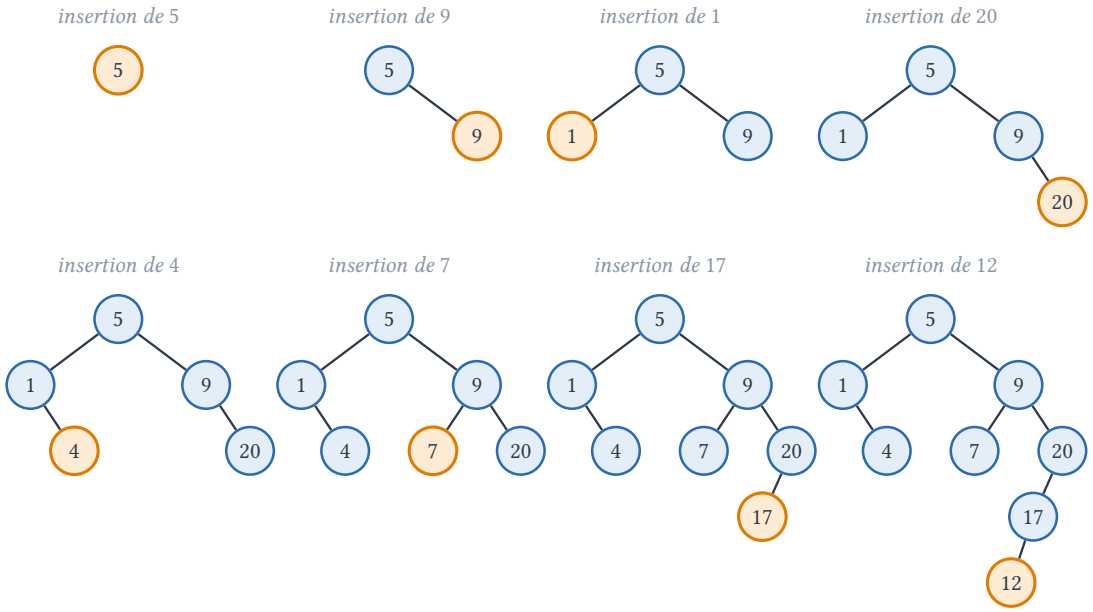


Figure 21.5 — Insertions successives de 5, 9, 1, 20, 4, 7, 17, 12 ; en orange, le nœud inséré à chaque étape.

Suppression d'un élément

Le problème de la suppression d'un élément (ou plutôt de sa première occurrence) d'un arbre binaire de recherche est plus compliqué. On peut le décomposer en :

- recherche de l'élément ;
- suppression ;
- recomposition de l'arbre.

Le nœud à supprimer est la racine d'un sous-arbre de l'arbre binaire de recherche. Commençons par considérer le problème de la suppression de la racine z d'un arbre binaire de recherche.

- Si les deux branches issues de z sont vides, on supprime et c'est fini (cas a) de la figure, $z = 13$).
- Si l'une des deux branches issues de z est vide, on se contente de garder l'autre branche. On conserve ainsi un arbre binaire de recherche (cas b) de la figure, $z = 16$).
- Si les deux branches sont non vides, soit y l'élément le plus à gauche du sous-arbre droit A_d de z, c'est-à-dire son minimum. On a $y \geq z$ par la propriété des ABR.

  Remarquons que y ne peut avoir de fils gauche car sinon il ne serait pas le plus à gauche de A_d.

  On peut donc le supprimer et le reporter en lieu et place de z : on obtient alors un nouvel arbre qui est un arbre binaire de recherche. En effet :

  - la branche droite est un arbre binaire de recherche (on a supprimé un nœud sans fils gauche) et l'étiquette de la racine y est, par construction, inférieure ou égale aux étiquettes des nœuds de cette branche ;

- la branche gauche est un arbre binaire de recherche (on n'y a pas touché) et la nouvelle étiquette de la racine y est supérieure ou égale à z qui majorait les étiquettes des nœuds de la branche gauche.

(cas c) de la figure, $z = 5$ et $y = 6$).

Les trois cas de suppression :

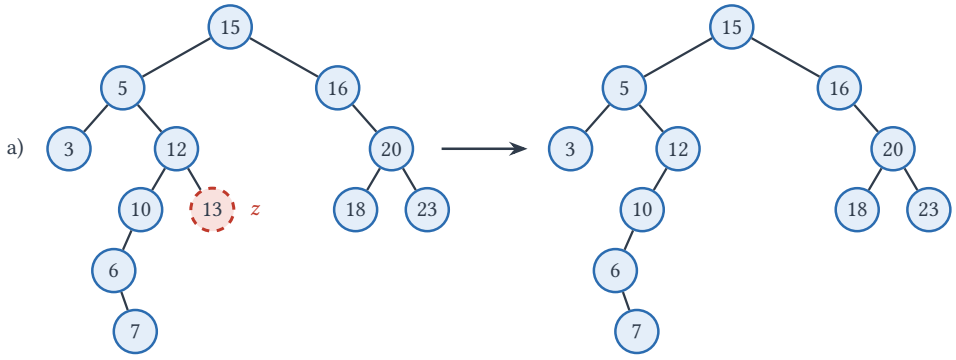


Figure 21.6 — Cas a) : z (en rouge) n'a pas de fils, on le supprime.

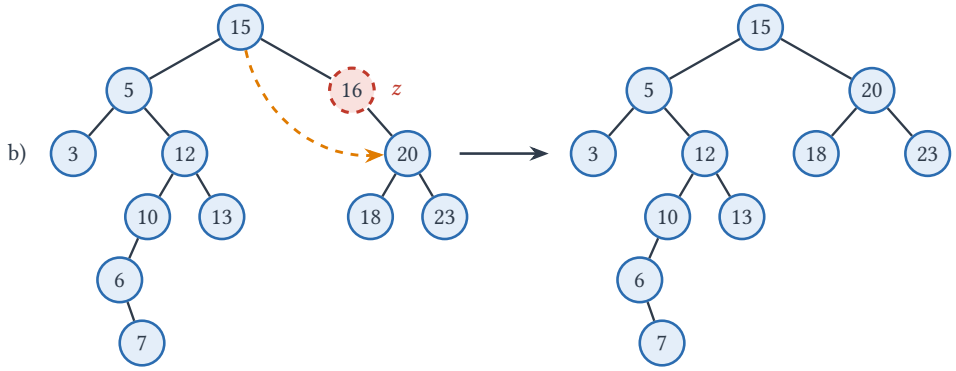


Figure 21.7 — Cas b) : z n'a qu'un fils, son sous-arbre prend sa place.

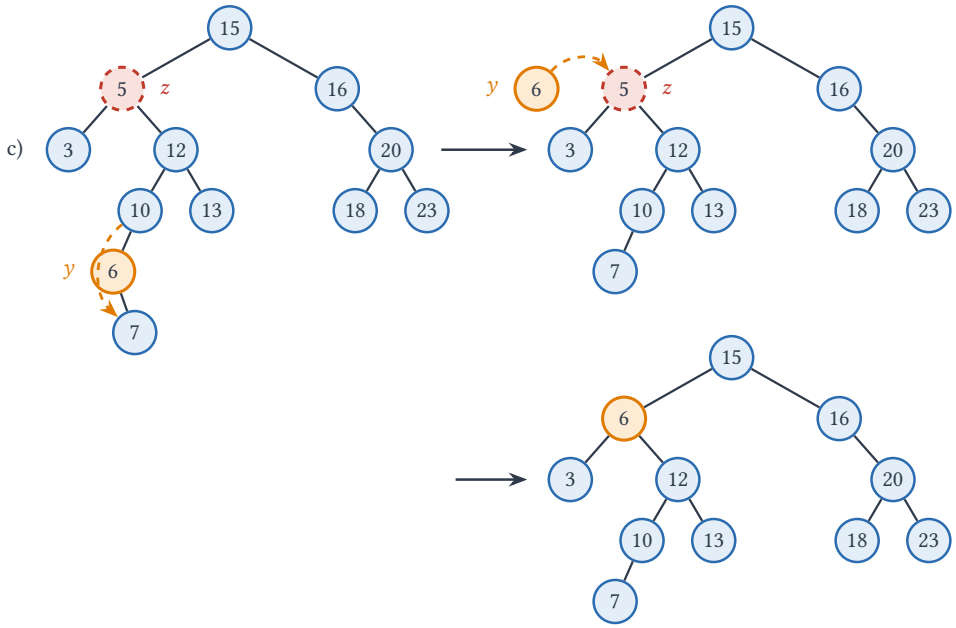


Figure 21.8 — Cas c) : z est remplacé par le minimum y de son sous-arbre droit.

On en déduit le programme suivant :

```
let rec supprime a x =
    match a with
    | Nil -> Nil
    | Noeud(g, y, d) when y < x ->
        Noeud(g, y, supprime d x)
    | Noeud(g, y, d) when y > x ->
        Noeud(supprime g x, y, d)
    (* à partir d'ici l'étiquette est forcément x *)
    | Noeud(Nil, _, Nil) -> Nil
    | Noeud(Nil, _, d) -> d
    | Noeud(g, _, Nil) -> g
    | Noeud(g, _, d) ->
        let y = minimum d in
        Noeud(g, y, supprime d y)
```

✍ La recherche de x, celle du minimum de `d` puis la suppression de y dans `d` suivent chacune une branche : la suppression est en $O(h(a))$. La correction se démontre par induction structurelle sur a, comme pour l'insertion, en utilisant l'analyse des trois cas ci-dessus.

21.4.4 Équilibrage

Principe et approche naïve

✎ Les opérations précédentes sont toutes en $O(h(a))$, or on a vu que $h(a)+1 \leq |\,a\,| \leq 2^{h(a)+1}-1$ donc $\log_2(|\,a\,|+1)-1 \leq h(a) \leq |\,a\,|-1$. On aimerait se rapprocher au plus de $\log_2|\,a\,|$ et pour cela, il faut que l'arbre soit le plus proche possible d'un arbre parfait. On parle alors d'équilibrage.

Par exemple, insérer successivement $1, 2, \ldots, n$ dans un ABR vide produit un peigne de hauteur $n-1$: toutes les opérations sont alors en $O(n)$.

Une approche naïve consiste à extraire les valeurs dans l'ordre croissant à l'aide d'un parcours infixe puis à reconstruire un arbre équilibré par dichotomie : on place la valeur médiane à la racine et on construit récursivement les sous-arbres gauche et droit. Cela coûte $O(n)$ à chaque rééquilibrage : on cherche donc plutôt des structures qui restent équilibrées au fil des opérations.

Arbres 2-3

Définition 21.30

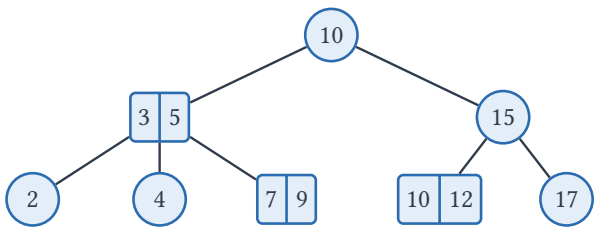
Un arbre 2-3 est un arbre étiqueté, éventuellement vide, contenant des nœuds de deux types :

- des nœuds binaires $N(g, x, d)$ où g, d sont des arbres 2-3 et x une étiquette tels que $\max g \leq x \leq \min d$ (c'est la même condition que les ABR) ;
- des nœuds ternaires $M(g, x, m, y, d)$ où g, m, d sont des arbres 2-3 et x, y sont des étiquettes telles que : $\max g \leq x \leq \min m \leq \max m \leq y \leq \min d$.

et parfaitement équilibré : tous les chemins de la racine à un arbre vide ont même longueur. En particulier les fils d'un nœud sont soit tous vides (le nœud est alors une feuille), soit tous non vides.

Un nœud ternaire contient donc un fils central dont les valeurs sont comprises entre ses deux étiquettes.

Voici un exemple d'arbre 2-3 :



Théorème 21.31

Un arbre 2-3 non vide à n nœuds est de hauteur au plus $\log_2(n+1) - 1 \leq \lfloor \log_2 n \rfloor$.

Preuve Montrons par récurrence sur $h \geq 0$ qu'un arbre 2-3 de hauteur h a au moins $2^{h+1} - 1$ nœuds. Pour $h = 0$, l'arbre est une feuille : il a 1 nœud. Si $h > 0$, la racine a au moins deux fils, tous de hauteur $h - 1$ par équilibre parfait, qui ont donc chacun au moins $2^h - 1$ nœuds par hypothèse de récurrence ; l'arbre a donc au moins $1 + 2(2^h - 1) = 2^{h+1} - 1$ nœuds.

Ainsi $n + 1 \geq 2^{h+1}$, soit $h \leq \log_2(n+1) - 1$. Comme h est entier et $\log_2(n+1) - 1 < \log_2 n$ pour $n \geq 1$, on a $h \leq \lfloor \log_2 n \rfloor$. Le maximum est atteint quand tous les nœuds sont binaires : les nœuds ternaires rendent l'arbre plus compact.

□

Recherche

Pour rechercher un élément x on adapte l'algorithme des ABR :

- pour un nœud binaire $N(g, y, d)$
  - si $x < y$ on cherche dans g ;
  - si $x = y$ on a trouvé le nœud voulu ;
  - si $x > y$ on cherche dans d.
- pour un nœud ternaire $M(g, y, m, z, d)$
  - si $x < y$ on cherche dans g ;
  - si $x = y$ ou $x = z$ on a trouvé le nœud voulu ;
  - si $y < x < z$ on cherche dans m ;
  - si $z < x$ on cherche dans d.

La complexité de cet algorithme est donc encore en $O(h)$ où h est la hauteur de l'arbre. D'après le théorème précédent, elle est en $O(\log n)$.

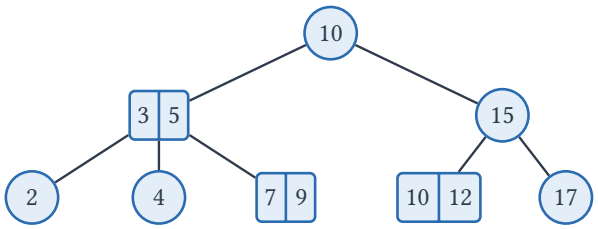
Insertion

Pour insérer un élément x dans un arbre 2-3, on va temporairement enrichir sa structure par des nœuds quaternaires satisfaisant les mêmes conditions sur les clés que les nœuds binaires et ternaires.

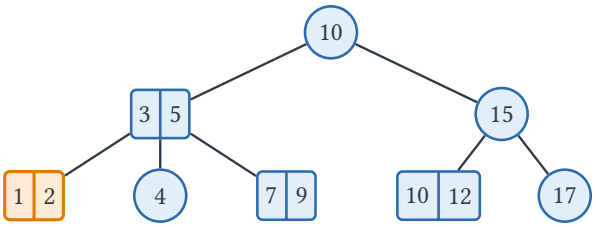
Pour insérer, on effectue alors une recherche qui nous amène à la feuille où x doit être placé. On transforme alors cette feuille en augmentant son arité et en plaçant x à la bonne position par rapport aux étiquettes qu'elle contient. L'équilibre parfait est préservé, car on n'a créé aucun nouveau niveau.

Exemples :

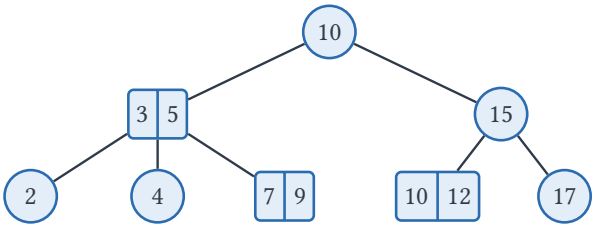
- insertion de 1 dans



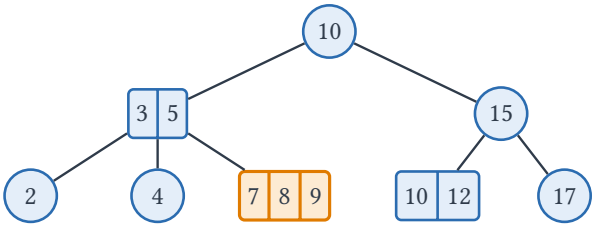
donne



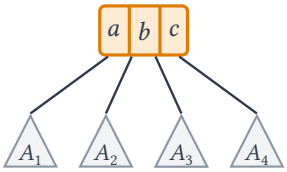
- insertion de 8 dans



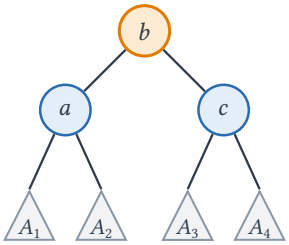
donne



Maintenant on va supprimer les nœuds quaternaires en les faisant remonter, au pire, jusqu'à la racine pour laquelle on peut faire l'opération suivante :

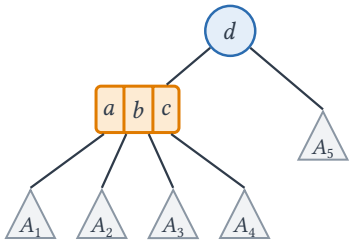


→

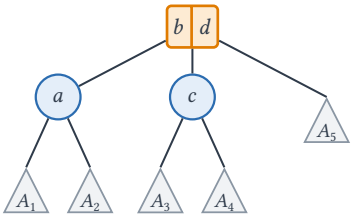


Voici tous les cas de transformations selon le type du père et la position du nœud quaternaire :

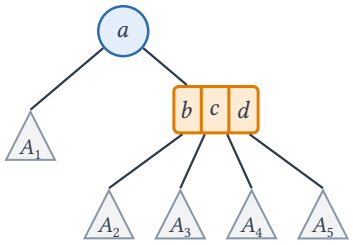
- fils gauche d'un binaire :



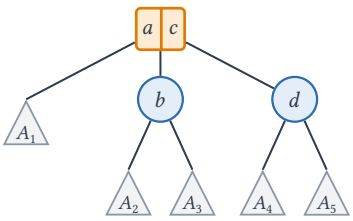

```
$\rightarrow$
```



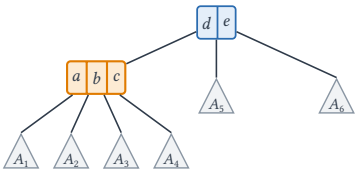
- fils droit d'un binaire :



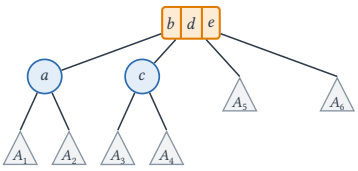

```
$\rightarrow$
```



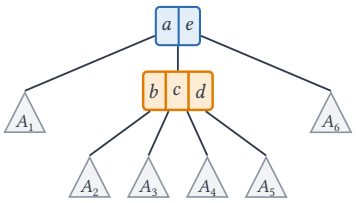
- fils gauche d'un ternaire :



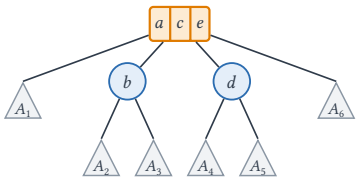

```
$\rightarrow$
```



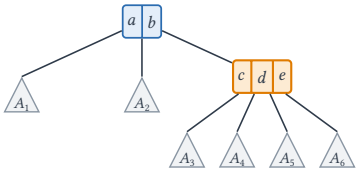
- fils central d'un ternaire :



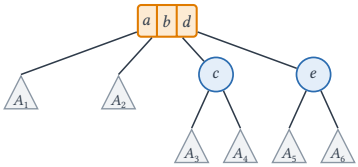

```
$\rightarrow$
```



- fils droit d'un ternaire :




```
$\rightarrow$
```



On observe rapidement que toutes ces transformations préservent la position des sous-arbres vis-à-vis des étiquettes des nœuds, ainsi que la profondeur des sous-arbres A_i (sauf l'éclatement de la racine, qui augmente de un la profondeur de tous les nœuds, donc préserve l'équilibre parfait). Chaque transformation fait remonter le nœud quaternaire d'un niveau, ou le fait disparaître. Ainsi, au final, on aura obtenu un arbre 2-3, en $O(h)$ étapes.

Remarque sur l'implémentation

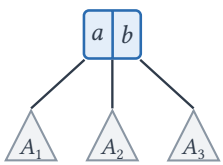
Les arbres 2-3 sont simples à comprendre mais complexes à programmer :

- beaucoup de cas à traiter, notamment en fonction des parents ;
- des nœuds intermédiaires 2-3-4 qui demandent d'avoir un type de travail et un type final.

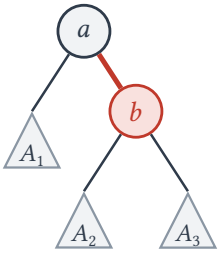
On va voir dans un dernier temps des arbres quasiment équivalents à ceux-ci mais bien plus simples à programmer.

Arbres rouges et noirs

On va transformer un arbre 2-3 en ABR en transformant les nœuds ternaires ainsi :



→



Afin de se souvenir que le nœud b provient de cette séparation, on le marque en rouge. Ici on a également marqué l'arête droite en rouge mais on l'omettra dans la suite.

Quelques remarques, en convenant que les arbres vides sont noirs :

- la racine est noire et les arbres vides sont noirs ;
- le fils gauche d'un nœud est toujours noir ;
- il ne peut exister d'arêtes liant deux nœuds rouges car par construction les nœuds rouges sont toujours entourés de nœuds noirs ;
- les chemins de la racine aux feuilles dans l'arbre 2-3 sont tous de même longueur et à chacun des nœuds de l'arbre 2-3 correspond exactement un nœud noir, on en déduit donc que tous les chemins de la racine aux arbres vides contiennent le même nombre de nœuds noirs.

Réciproquement un ABR dont les nœuds peuvent avoir ces deux couleurs et qui vérifie ces propriétés est appelé un arbre rouge et noir.

En fait, on a une bijection entre les arbres 2-3 et les arbres rouges et noirs ainsi définis.

Remarque 21.32 La condition « le fils gauche d'un nœud est toujours noir » est spécifique à la variante présentée ici, qui est celle qui correspond exactement aux arbres 2-3 (les arbres rouges et noirs *penchés*, introduits par R. Sedgewick, ici penchés à droite). Dans la définition usuelle des arbres rouges et noirs, un nœud noir peut avoir deux fils rouges (ce qui correspond à un nœud quaternaire d'un arbre 2-3-4), et seules les trois autres conditions sont imposées.

La proposition suivante permet de voir que les arbres rouges et noirs sont de bons ABR :

Théorème 21.33

La hauteur d'un arbre rouge et noir est au plus égale à $2h + 1$ où h est la hauteur de l'arbre 2-3 qui lui est associé.

Ainsi tout arbre rouge et noir de taille $n \geq 1$ est de hauteur au plus $2\log_2(n+1) - 1$, donc $O(\log n)$.

Preuve Un chemin de la racine vers un nœud de l'arbre rouge et noir contient au plus $h + 1$ nœuds noirs, un par nœud de l'arbre 2-3 traversé. Comme la racine est noire et qu'un nœud rouge a toujours un père noir, il contient au plus autant de nœuds rouges que de nœuds noirs. Il a donc au plus $2(h + 1)$ nœuds, soit au plus $2h + 1$ arêtes.

L'arbre 2-3 associé a au plus n nœuds (chacun correspond à un ou deux nœuds de l'arbre rouge et noir), donc $h \leq \log_2(n + 1) - 1$ d'après le théorème sur les arbres 2-3, ce qui donne une hauteur au plus $2\log_2(n + 1) - 1$.

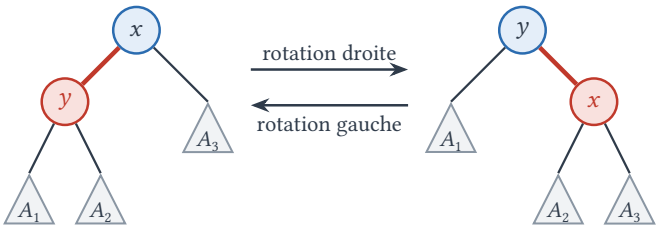
□

Pour l'insertion d'un nœud, on va déstabiliser la coloration en mettant trop de nœuds rouges, puis on va rétablir les contraintes en remontant vers la racine. Cela rappelle l'élimination des nœuds quaternaires des arbres 2-3. Mettre des nœuds rouges a l'avantage de préserver l'équilibrage noir.

Pour rétablir les couleurs, on va alors utiliser une succession de deux opérations élémentaires : les rotations et la bascule.

Rotations

On définit les deux rotations gauche et droite, duales l'une de l'autre, ainsi :



Ici, on a utilisé la couleur bleue pour représenter la couleur du nœud de départ qui peut être soit rouge soit noire mais va être préservée.

Les rotations permettent donc de faire pencher des nœuds rouges à droite ou à gauche. Elles préservent la propriété d'ABR, car l'ordre infixe A_1, y, A_2, x, A_3 est le même dans les deux arbres.

On remarque que la hauteur en nœuds noirs de l'arbre n'a pas changé.

On peut passer d'une des trois représentations convenables des nœuds quaternaires à une autre à l'aide de rotations :

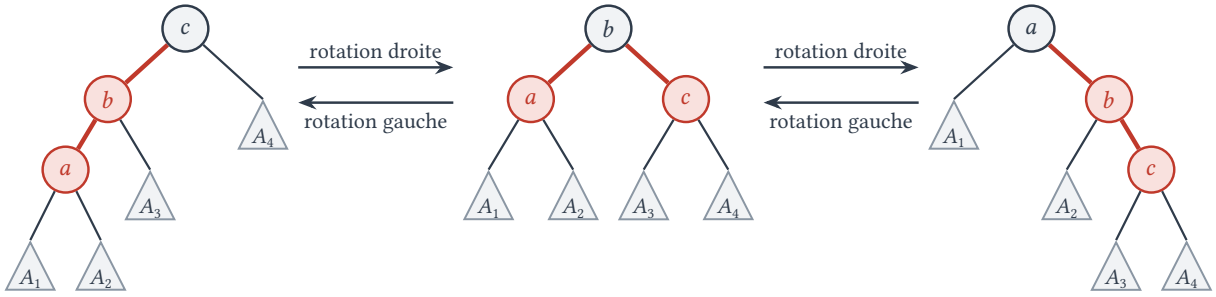


Figure 21.9 — Les trois représentations d'un nœud quaternaire, reliées par des rotations.

Bascule

Une autre opération importante dans les arbres rouges et noirs est la bascule qui permet de faire remonter les défauts de coloration vers la racine tout en préservant l'équilibre en nœuds noirs :

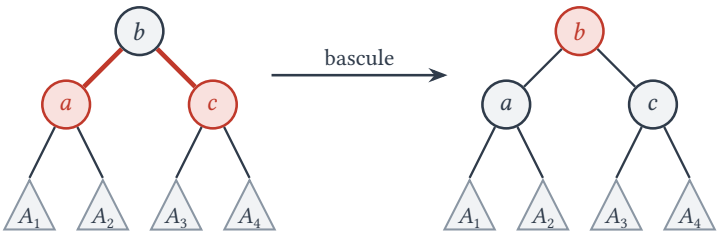


Figure 21.10 — Bascule : les deux fils rouges deviennent noirs et le père devient rouge.

Si on effectue une bascule directement sur la racine de l'arbre considéré, on finit par la noircir ce qui augmente de un la profondeur noire de chaque feuille mais ne change pas l'équilibrage global.

Insertion

Selon les remarques précédentes, on ne va pas hésiter à perdre la propriété des arbres rouges et noirs temporairement. De toutes les propriétés de ces arbres, c'est sûrement le fait qu'ils soient parfaitement équilibrés en nœuds noirs qui est le plus dur à assurer. On ne va donc pas toucher à cette propriété : on va insérer des nœuds rouges à la place d'un arbre vide.

En faisant cela on risque d'avoir deux types de problèmes :

- des arêtes rouge-rouge ;

- des fils gauches rouges.

On va alors traiter les différents cas d'insertion et voir comment on peut les ajuster pour obtenir des arbres rouges et noirs.

Insertion dans un nœud issu d'un nœud binaire

Un nœud noir issu d'un nœud binaire a nécessairement ses deux fils qui sont vides ou non vides simultanément, car ils proviennent directement de la structure parfaitement équilibrée des arbres 2-3.

On a alors les deux cas suivants d'insertion :

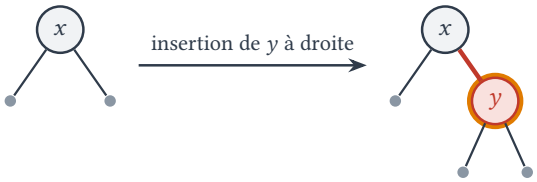


Figure 21.11 — Le nœud inséré (cerclé d'orange) est rouge : on obtient directement un nœud ternaire.

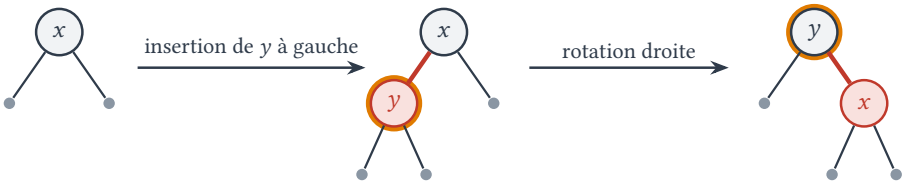


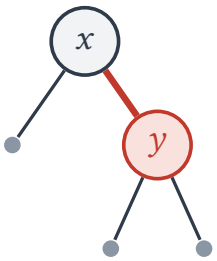
Figure 21.12 — Insertion à gauche puis rotation droite ; le nœud inséré est cerclé d'orange.

Dans les deux cas on se ramène à un nœud ternaire directement.

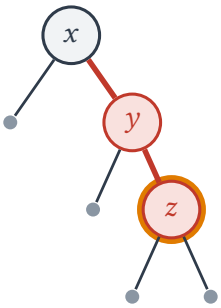
Insertion dans un nœud issu d'un nœud ternaire

On a trois cas d'insertion selon la position de la nouvelle valeur par rapport aux deux valeurs du nœud ternaire.

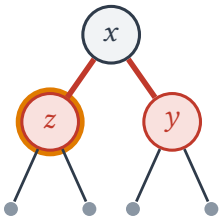
Supposons que l'on insère z dans l'arbre :



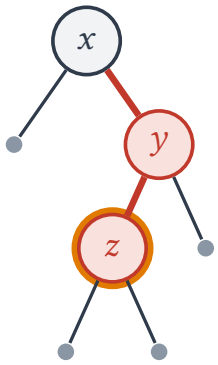
- Si $x \leq y \leq z$:



- Si $z \leq x \leq y$:



- Si $x \leq z \leq y$:



Ici, on ne peut pas rétablir tout de suite la structure mais on peut faire remonter le problème jusqu'à la racine, quitte à faire une bascule finale qui va la noircir. On va donc partir d'un des trois problèmes précédents mais supposer qu'il se situe sur un nœud quelconque avec des sous-arbres rouges et noirs t_1, t_2, t_3 et t_4.

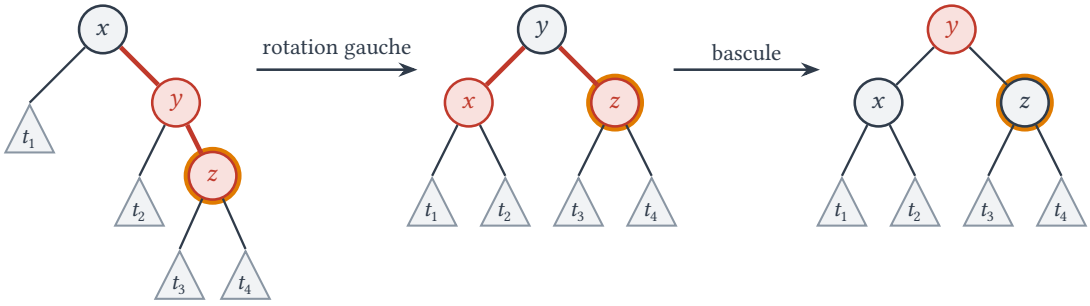


Figure 21.13 — Cas $x \leq y \leq z$: rotation gauche puis bascule (z inséré, cerclé d'orange).

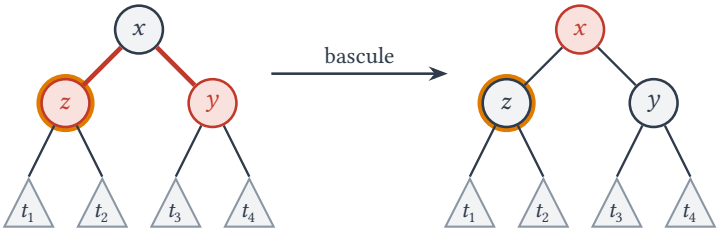


Figure 21.14 — Cas $z \leq x \leq y$: une bascule suffit.

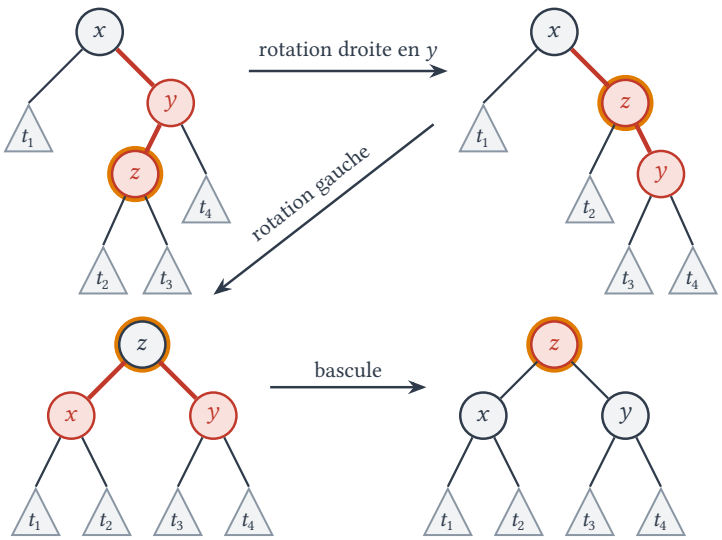


Figure 21.15 — Cas $x \leq z \leq y$: rotation droite en y, rotation gauche, puis bascule.

On remarque qu'on effectue essentiellement les mêmes opérations, et quitte à les inhiber quand on ne peut pas les appliquer, on peut se dire que pour insérer z dans l'arbre a, on le rajoute récursivement à gauche ou à droite, on obtient ainsi a' et on effectue alors

$$(\text{bascule}^? \circ \text{rot\_gauche}^? \circ \text{rot\_droite}^?)(a')$$

où ici, $\text{op}^?$ signifie qu'on effectue l'opération uniquement si elle s'applique :

- pour une rotation droite, il faut que le fils gauche soit rouge et le fils droit noir, pour ne pas tomber dans le cas d'une bascule ;
- pour une rotation gauche, il faut que le fils droit et le fils droit du fils droit soient rouges ;
- pour une bascule, que les deux fils soient rouges.

Implémentation

On commence par définir le type des arbres rouges et noirs :

```
type couleur = R | N
type 'a arn = Nil | Noeud of couleur * 'a arn * 'a * 'a arn
```

On définit ensuite des fonctions qui nous seront utiles vu la remarque précédente pour tester les couleurs à gauche et à droite :

```
let gauche a = match a with
    | Nil -> failwith "Invalide"
    | Noeud(_,g,_,_) -> g
let droite a = match a with
    | Nil -> failwith "Invalide"
    | Noeud(_,_,_,d) -> d
let couleur a = match a with
    | Nil -> N
    | Noeud(c,_,_,_) -> c
let rouge a = couleur a = R
let noir a = not (rouge a)
```

On implémente alors les rotations et la bascule avec le filtrage.

```
let rotation_gauche a = match a with
    | Noeud(c,t1,x,Noeud(R,t2,y,t3)) ->
            Noeud(c,Noeud(R,t1,x,t2),y,t3)
    | _ -> failwith "Invalide"
let rotation_droite a = match a with
```

```
    | Noeud(c,Noeud(R,t1,x,t2),y,t3)
        -> Noeud(c,t1,x,Noeud(R,t2,y,t3))
    | _ -> failwith "Invalide"
let bascule a = match a with
    | Noeud(N,Noeud(R,t1,x,t2),y,Noeud(R,t3,z,t4))
        -> Noeud(R,Noeud(N,t1,x,t2),y,Noeud(N,t3,z,t4))
    | _ -> failwith "Invalide"
let noircit a = match a with
    | Noeud(_,g,x,d) -> Noeud(N,g,x,d)
    | Nil -> Nil
```

Puis, les versions optionnelles qui ne s'effectuent que si c'est nécessaire :

```
let essai_rotgauche a =
    if rouge (droite a) && rouge (droite (droite a))
    then rotation_gauche a else a
let essai_rotdroite a =
    if rouge (gauche a) && noir (droite a)
    then rotation_droite a else a
let essai_bascule a =
    if rouge (gauche a) && rouge (droite a)
    then bascule a else a
```

On peut alors réaliser l'insertion récursivement :

```
let insere a x =
    let rec aux a =
        match a with
        | Nil -> Noeud(R,Nil,x,Nil)
        | Noeud(c,g,y,d) ->
            let a' =
                if x < y
                then Noeud(c,aux g,y,d)
                else Noeud(c,g,y,aux d)
            in
            essai_bascule
                (essai_rotgauche
                    (essai_rotdroite a'))
    in noircit (aux a)
```

✍ **Propriété 21.34**

Soit a un sous-arbre d'un arbre rouge et noir, de hauteur noire b (nombre de nœuds noirs sur chaque chemin vers un arbre vide). Alors `aux a` est un ABR contenant les étiquettes de a et x, de même hauteur noire b, et :

- si la racine de a est noire (ou a vide), `aux a` vérifie toutes les conditions des arbres rouges et noirs, sauf éventuellement que sa racine peut être rouge ;
- si la racine de a est rouge, `aux a` a une racine rouge et vérifie toutes les conditions, sauf éventuellement qu'**un** de ses fils peut être rouge (ce fils ayant lui-même deux fils noirs).

Par conséquent `insere a x` est un arbre rouge et noir, et l'insertion est en $O(\log n)$.

✍ **Preuve (esquisse)** Par induction structurelle sur a. Les rotations et la bascule préservent la propriété d'ABR et la hauteur noire, l'appel récursif aussi par hypothèse d'induction. Le défaut renvoyé par l'appel récursif est situé à la racine du fils modifié ; les figures précédentes traitent exactement les situations possibles :

- un nœud noir dont un fils (issu d'un nœud binaire) devient rouge : c'est le cas binaire, réglé par au plus une rotation droite ;
- un nœud noir dont les deux fils sont rouges : une bascule fait remonter le défaut (la racine devient rouge) ;

- un nœud noir dont le fils droit rouge a un fils rouge : une rotation droite éventuelle (effectuée lors du traitement du fils rouge) puis une rotation gauche et une bascule.

Enfin `noircit` rend la racine noire, ce qui augmente de un la hauteur noire de tous les chemins à la fois. Le calcul suit une branche avec un travail constant par niveau, d'où le $O(h) = O(\log n)$ d'après le théorème précédent.

□

Remarque 21.35 Ce code a été testé en comparant, après chaque insertion d'éléments aléatoires (ou croissants), le parcours infixe de l'arbre avec la liste triée des éléments insérés, et en vérifiant les invariants : racine noire, pas de fils gauche rouge, pas d'arête rouge-rouge, même nombre de nœuds noirs sur toutes les branches, hauteur au plus $2\log_2(n+1) - 1$.

21.5 Tas

21.5.1 Présentation

On va considérer une structure de donnée permettant de réaliser efficacement deux opérations :

- récupérer la plus grande des valeurs ajoutées et la supprimer ;
- ajouter une nouvelle valeur.

Avec des arbres, on cherche à réaliser ces opérations en $O(\log n)$ où n est le nombre de valeurs présentes.

Définition 21.36

On appelle *tas sur l'ensemble* X un arbre binaire étiqueté par X et complet à gauche tel que pour tout nœud de l'arbre, son étiquette est supérieure ou égale à celle de ses fils.

Exemple 21.37

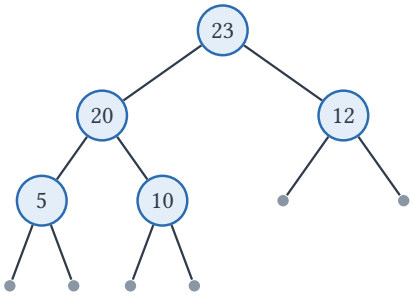


Figure 21.16 — Un tas : chaque nœud a une étiquette supérieure ou égale à celles de ses fils.

Remarque 21.38 On parle de *heap* en anglais. Ici, nous présentons des tas-max permettant de déduire le maximum, on pourrait définir des tas-min, mais quitte à renverser la relation d'ordre, il s'agit de la même structure.

Remarque 21.39 Par transitivité, l'étiquette d'un nœud est supérieure ou égale à toutes les étiquettes de son sous-arbre : le maximum est à la racine. Un tas à n nœuds étant complet, sa hauteur est $\lfloor \log_2 n \rfloor$ (il a au moins 2^h nœuds : les niveaux 0 à $h-1$ sont pleins et le niveau h est non vide).

21.5.2 Opérations

Dans les deux opérations qu'on souhaite implémenter, on va avoir besoin d'un accès rapide à la dernière feuille insérée et à la place, à sa droite, de la prochaine feuille qu'on pourrait insérer.

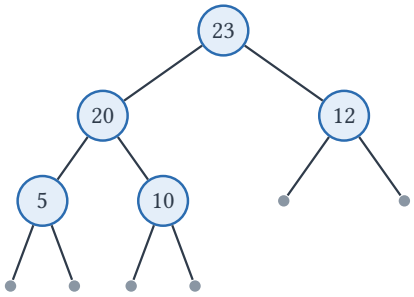
Insertion

Pour insérer un élément dans un tas on adopte la stratégie suivante :

- on le place dans la seule place libre permettant de garantir que l'arbre soit complet à gauche ;
- on l'insère à la bonne place sur la branche qui le relie à la racine en effectuant des échanges successifs jusqu'à ce que la condition de tas soit vérifiée (étape appelée *remontée du tas*).

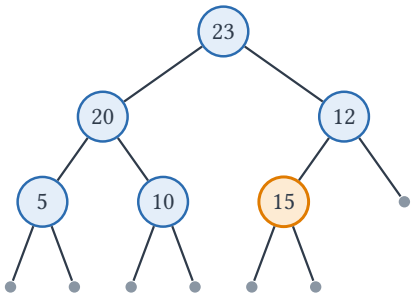
On va illustrer cet algorithme sur un exemple.

On part du tas

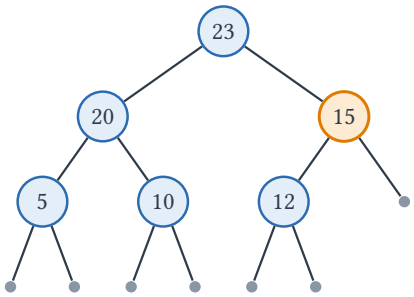


dans lequel on insère la valeur 15.

On commence par ajouter ce nœud dans le premier emplacement libre à droite de 10 :



Comme le fils gauche de 12 a une valeur plus grande que celui-ci on échange les étiquettes pour obtenir :



Maintenant le père de 15 est bien plus grand que celui-ci, on s'arrête car on a obtenu un tas.

✍ **Remarque 21.40** Pendant la remontée, la condition de tas est vérifiée partout sauf éventuellement entre le nœud i contenant la nouvelle valeur v et son père p. Si $e_p < v$, on les échange : v est plus grande que l'ancienne étiquette e_p, elle-même plus grande que l'autre fils de p et que le sous-arbre de i, donc la condition devient vraie autour de p et le seul défaut possible remonte entre p et son père. On s'arrête à la racine ou quand $e_p \geq v$: on a alors un tas.

Extraction du maximum

Pour supprimer la racine :

- soit x l'étiquette du nœud situé juste avant le prochain emplacement libre (il s'agit de la dernière position remplie par une insertion), on place la valeur x à la racine et on supprime le nœud où était situé x, ce qui préserve le caractère complet à gauche ;

- on fait ensuite descendre x en l'échangeant à chaque étape avec le plus grand de ses fils, tant que celui-ci est plus grand que x : choisir le plus grand garantit que le nouveau père respecte bien la condition de tas vis-à-vis de ses deux fils (étape appelée *descente du tas*).

Sur le tas suivant on a $x = 12$

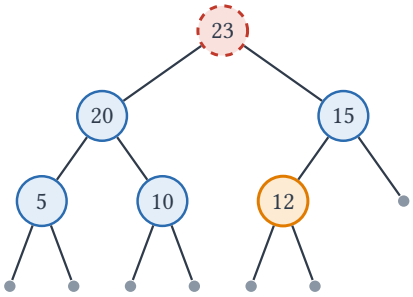
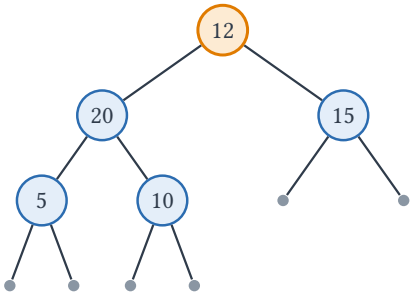
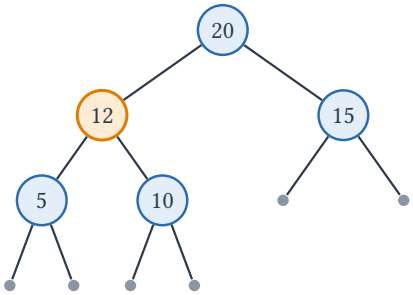


Figure 21.17 — La racine 23 (en rouge) est supprimée et remplacée par $x = 12$ (en orange).

On obtient donc après la première étape l'arbre



On échange alors 12 avec 20 qui est le plus grand des fils :



Maintenant 12 est à la bonne place : on a obtenu un tas.

Complexité

Dans l'hypothèse de pouvoir accéder rapidement (au moins en $O(\log n)$) à la fin du dernier niveau, on peut réaliser ces deux opérations en remontant ou en descendant le long d'une branche, donc en $O(\log n)$ puisque la hauteur du tas est $\lfloor \log_2 n \rfloor$.

21.5.3 Implémentation

On va reposer sur l'implémentation à plat d'un arbre comme décrit précédemment. On va supposer qu'on dispose d'une constante N et que l'on sait que le tas contiendra au maximum N valeurs. Cette implémentation est alors à rapprocher de celle faite pour les files ou les piles dans des tableaux.

```
struct tas {
    int *elements;
    int taille;
    int taille_max;
```

```
};
typedef struct tas tas;
```

```
tas *tas_cree(int N)
{
    tas *t = malloc(sizeof(tas));
    t->elements = malloc(N * sizeof(int));
    t->taille = 0;
    t->taille_max = N;
    return t;
}

void tas_libere(tas *t)
{
    free(t->elements);
    free(t);
}
```

```
void swap(int *t, int i, int j)
{
    int z = t[i];
    t[i] = t[j];
    t[j] = z;
}

int pere(int i)
{
    return (i-1)/2;
}

int gauche(int i)
{
    return 2*i+1;
}

int droite(int i)
{
    return 2*i+2;
}
```

```
void tas_insere(tas *t, int x)
{
    assert(t->taille < t->taille_max);
    int n = t->taille;
    t->taille++;
    t->elements[n] = x;
    while(n > 0 && t->elements[pere(n)] < t->elements[n])
    {
        swap(t->elements, n, pere(n));
        n = pere(n);
    }
}
```

```
bool feuille(tas *t, int i)
{
    return gauche(i) >= t->taille;
}

// Indice du plus grand des fils de i, qu'on suppose ne pas être une feuille
int fils_max(tas *t, int i)
{
    int g = gauche(i);
    int d = droite(i);
    if (d < t->taille && t->elements[d] > t->elements[g])
        return d;
    return g;
}
```

```c
int tas_extrait_max(tas *t)
{
    assert(t->taille > 0);
    int racine = t->elements[0];
    t->taille--;
    t->elements[0] = t->elements[t->taille];

    int n = 0;
    while(!feuille(t, n) && t->elements[n] < t->elements[fils_max(t, n)])
    {
        int f = fils_max(t, n);
        swap(t->elements, n, f);
        n = f;
    }

    return racine;
}
```

21.5.4 Application au tri

Pour trier, il suffit de créer un tas, d'insérer tous les éléments puis de les extraire un à un. On appelle cela le *tri par tas*. C'est d'ailleurs l'origine historique de la structure de donnée. Il est en $O(n \log n)$, puisqu'on fait n insertions et n extractions en $O(\log n)$.

Une manière intéressante de faire cet algorithme consiste à le faire en place, c'est-à-dire sans utiliser un autre tableau pour le tri. Pour cela, on va maintenir dans le même tableau le tas en construction en préfixe et les éléments à insérer en suffixe.

Pour reconstruire le tableau trié, il suffit alors d'extraire les maximums et de les placer à la fin du tableau en faisant ici grossir le suffixe trié et réduire le tas préfixe.

```c
void heapify(int *t, int sz)
{
    for(int i = 1; i < sz; i++)
    {
        int j = i;
        while(j > 0 && t[(j-1)/2] < t[j])
        {
            int p = (j-1)/2;
            swap(t, j, p);
            j = p;
        }
    }
}

void sort(int *t, int sz)
{
    heapify(t, sz);
    for(int i = sz-1; i > 0; i--)
    {
        // le tas occupe t[0..i] : on place son maximum en position i
        swap(t, i, 0);
        // le tas occupe maintenant t[0..i-1] : on fait descendre t[0]
        int j = 0;
        while(2*j+1 < i)
        {
            int c = 2*j+1;
            if (c+1 < i && t[c+1] > t[c])
                c = c+1;
            if (t[j] >= t[c])
                break;
            swap(t, j, c);
            j = c;
        }
    }
}
```

21.5.5 Application aux files de priorité

À la manière des dictionnaires, on peut considérer des couples $(prio, val)$ pour les étiquettes en utilisant uniquement la première composante dans le tas. Ainsi, on réalise la structure de données files de priorité introduite dans un chapitre précédent.

21.6 Exercices

21.6.1 Reconstruction

Exercice

On considère un arbre binaire à 27 nœuds étiquetés par les lettres `A`, `B`, ..., `Z` et `&`.

On connaît l'ordre des étiquettes vues dans un parcours préfixe :

```
I Q J H L E M V O T S B R G Y Z K C A & F P N U D W X
```

et dans un parcours postfixe :

```
H E M L J V Q S G Y R Z B T C P U D N F W & X A K O I
```

Dessiner cet arbre binaire.

Exercice

On rappelle qu'un arbre binaire est strict si chaque nœud qui n'est pas une feuille a exactement deux fils non vides.

1. Décrire et analyser un algorithme récursif qui reconstruit un arbre binaire strict étant donné son ordre préfixe et son ordre postfixe.
2. Montrer que ce n'est pas possible de réaliser un tel algorithme sans savoir que l'arbre est strict.

21.6.2 Lecture infixe d'un ABR

Exercice

1. Montrer qu'un arbre binaire est un ABR si et seulement si la suite de ses étiquettes dans l'ordre infixe est croissante.
2. En déduire une fonction `est_abr : 'a arbre -> bool` en $O(|\, a \,|)$.

Corrigé

1. Par induction structurelle. Pour nil, les deux propriétés sont vraies. Pour $a = (g, x, d)$, la suite infixe est L_g, x, L_d où L_g et L_d sont les suites infixes de g et d. Elle est croissante si et seulement si L_g et L_d sont croissantes, $\max L_g \leq x$ et $x \leq \min L_d$, c'est-à-dire, par hypothèse d'induction, si et seulement si g et d sont des ABR et $\max g \leq x \leq \min d$: c'est la définition d'un ABR.
2. On construit la liste infixe en temps linéaire avec un accumulateur, puis on vérifie qu'elle est croissante.

```
let est_abr a =
    let rec infixe a acc = match a with
        | Nil -> acc
        | Noeud(g, x, d) -> infixe g (x :: infixe d acc)
    in
    let rec croissante l = match l with
        | x :: y :: q -> x <= y && croissante (y :: q)
        | _ -> true
```

```
    in
    croissante (infixe a [])
```

22

TRAVAUX PRATIQUES 22

Arbres - Travaux pratiques

DANS CE CHAPITRE

22.1 Arbres en OCaml
22.2 Arbres non binaires, tries
22.3 Tries
22.4 Arbres binaires en C, ABR, arbres rouges et noirs

Ces travaux pratiques accompagnent le chapitre sur les arbres.

22.1 Arbres en OCaml

22.1.1 Premières fonctions

On va considérer le type des arbres binaires `'a arbre` défini par :

```
type 'a arbre = Nil | Noeud of 'a arbre * 'a * 'a arbre
```

Question

Écrire des fonctions :

```
(* Calcule le nombre de noeuds de a *)
let taille (a : 'a arbre) : int
(* Calcule la hauteur de a *)
let hauteur (a : 'a arbre) : int
(* Indique si a est réduit à une feuille *)
let feuille (a : 'a arbre) : bool
```

Corrigé

```
let rec taille a = match a with
    | Nil -> 0
    | Noeud(g,x,d) -> 1 + taille g + taille d

let rec hauteur a = match a with
    | Nil -> -1
    | Noeud(g,x,d) ->
            1 + max (hauteur g) (hauteur d)

let feuille a = match a with
    | Nil -> false
    | Noeud(g, x, d) -> g = Nil && d = Nil
```

Question

Écrire une fonction `sous_arbres` qui renvoie la liste des sous-arbres non vides d'un arbre.

En déduire des fonctions `noeuds`, `feuilles` et `noeuds_internes` qui renvoient la liste des étiquettes des nœuds correspondants.

Corrigé

```
let rec sous_arbres a = match a with
    | Nil -> []
    | Noeud(g, x, d) ->
       a :: (sous_arbres g @ sous_arbres d)

let rec noeuds a = match a with
    | Nil -> []
    | Noeud(g, x, d) ->
        x :: (noeuds g @ noeuds d)

let rec feuilles a = match a with
    | Nil -> []
    | Noeud(g, x, d) ->
        if feuille a then [x]
        else feuilles g @ feuilles d

let rec noeuds_internes a = match a with
    | Nil -> []
    | Noeud(Nil, _, Nil) -> []
    | Noeud(g, x, d) ->
        x :: (noeuds_internes g @ noeuds_internes d)
```

On peut aussi déduire directement ces fonctions de `sous_arbres`, par exemple `let noeuds a = List.map (fun t -> match t with Noeud(_, x, _) -> x | Nil -> failwith "impossible") (sous_arbres a)`, et de même pour les feuilles avec `List.filter feuille`.

Pour accéder à un nœud de l'arbre, on descend en partant de la racine et en allant à gauche ou à droite. On peut donc représenter un tel chemin par une liste de déplacements :

```
type deplacement = Gauche | Droite
type chemin = deplacement list
```

Question

Écrire une fonction `chemin_noeud : 'a arbre -> chemin -> 'a option` qui renvoie l'étiquette d'un nœud donné par son chemin. On utilise un type `option` en cas de chemin invalide.

Corrigé

```
let rec chemin_noeud a l =
    match a, l with
    | Nil, _ -> None
    | Noeud(g,x,d), Gauche::q -> chemin_noeud g q
    | Noeud(g,x,d), Droite::q -> chemin_noeud d q
    | Noeud(g,x,d), [] -> Some x
```

22.1.2 Parcours

Sur le même modèle que l'exploration par *backtracking*, on va réaliser un parcours en profondeur d'un arbre en explorant à gauche puis à droite ses sous-arbres de manière récursive. Chaque nœud est donc vu trois fois :

- une première fois quand on le découvre avant d'explorer son sous-arbre gauche ;
- entre les deux explorations ;

- enfin quand on a fini d'explorer son sous-arbre droit.

Question

Écrire des fonctions `affiche_avant`, `affiche_milieu` et `affiche_apres` qui parcourent et affichent les étiquettes d'un `string arbre` selon les trois cas précédents. *Note : il s'agit essentiellement du même code à une ligne près.*

Corrigé

```
let rec affiche_avant a = match a with
    | Nil -> ()
    | Noeud(g,x,d) ->
            print_string x;
            affiche_avant g;
            affiche_avant d

let rec affiche_milieu a = match a with
    | Nil -> ()
    | Noeud(g,x,d) ->
            affiche_milieu g;
            print_string x;
            affiche_milieu d

let rec affiche_apres a = match a with
    | Nil -> ()
    | Noeud(g,x,d) ->
            affiche_apres g;
            affiche_apres d;
            print_string x
```

Question

On peut représenter une expression arithmétique comme $(2 + 3) \times 4$ en tant qu'arbre avec des opérations pour les nœuds internes et des valeurs pour les feuilles. Quitte à utiliser des `string` dans les deux cas, qu'obtient-on sur une telle expression avec les trois opérations précédentes ?

Corrigé

L'appel à `affiche_avant` sur l'expression $(2 + 3) \times 4$ va afficher `*+234`.

L'appel à `affiche_milieu` sur l'expression $(2 + 3) \times 4$ va afficher `2+3*4`.

L'appel à `affiche_apres` sur l'expression $(2 + 3) \times 4$ va afficher `23+4*`.

On retrouve ainsi les notions préfixes, infixes et postfixes d'une expression arithmétique. Remarquons que l'écriture infixe obtenue a perdu les parenthèses et ne permet plus de retrouver l'expression.

On peut également évaluer une telle expression :

```
let rec evalue a = match a with
    | Nil -> failwith "Vide"
    | Noeud(Nil,x,Nil) -> int_of_string x
    | Noeud(g, x, d) ->
            let eg = evalue g in
            let ed = evalue d in
            match x with
            | "+" -> eg + ed
            | "*" -> eg * ed
            | _ -> failwith "Ope inconnue"
```

On redonne ici l'implémentation d'une pile et d'une file en `OCaml` :

```
let pile_creer () = ref []
let pile_empile p x = p := x :: !p
let pile_depile p =
    match !p with
    | [] -> failwith "Pile vide"
    | x::q -> p := q; x
let pile_est_vide p = !p = []

let file_creer () = (pile_creer (), pile_creer ())
let file_bascule (pin, pout) = pout := List.rev !pin; pin := []
let file_enfile (pin, pout) x = pile_empile pin x
let file_defile (pin, pout) =
    if pile_est_vide pout
    then file_bascule (pin, pout);
    pile_depile pout
let file_est_vide (pin, pout) = pile_est_vide pin && pile_est_vide pout
```

Ces deux structures de données ont la même interface mais deux comportements différents. On va utiliser un unique type pour les représenter afin de permettre à une fonction d'utiliser soit une pile soit une file.

```
type ('a, 'b) taches = {
    creation : unit -> 'a;
    ajouter : 'a -> 'b -> unit;
    retirer : 'a -> 'b;
    est_vide : 'a -> bool
}

let taches_pile = {
    creation = pile_creer;
    ajouter = pile_empile;
    retirer = pile_depile;
    est_vide = pile_est_vide
    }

let taches_file = {
    creation = file_creer;
    ajouter = file_enfile;
    retirer = file_defile;
    est_vide = file_est_vide
    }
```

Remarque 22.1 Ici, on a une limitation du système de types. On aimerait paramétrer `taches` par `('b t, 'b)` où `t` est lui-même un type paramétré, ainsi on aurait soit `('b pile, 'b)` soit `('b file, 'b)` dans les deux cas. Cependant, ce n'est pas possible : en `OCaml`, un paramètre de type ne peut pas lui-même être un type paramétré. C'est pour cela qu'on a un `('a, 'b)` sans avoir de lien apparent entre `'a` et `'b`.

On pourra alors écrire une fonction prenant un gestionnaire de tâches en paramètre :

```
let f t a =
    let a_traiter = t.creation () in
    for i = 0 to Array.length a - 1 do
        t.ajouter a_traiter a.(i)
    done;
    while not (t.est_vide a_traiter) do
        print_int (t.retirer a_traiter)
    done
```

Ainsi `f taches_pile [|1;2;3|]` va afficher `321` et `f taches_file [|1;2;3|]` va afficher `123`.

Question

En déduire une fonction

```
parcours : ('a, int arbre) taches -> int arbre -> unit
```

qui effectue un parcours de l'arbre en affichant les étiquettes des nœuds visités dans l'ordre induit par le gestionnaire de tâches passé en paramètre.

Corrigé

```
let parcours t a =
    let avisiter = t.creation () in
    t.ajouter avisiter a;
    while not (t.est_vide avisiter) do
        match t.retirer avisiter with
        | Nil -> ()
        | Noeud(g,x,d) ->
                print_int x;
                t.ajouter avisiter g;
                t.ajouter avisiter d
    done
```

Avec `taches_pile` on obtient un parcours en profondeur (qui visite le fils droit avant le fils gauche), avec `taches_file` un parcours en largeur. Sur l'arbre de l'exemple du cours, on obtient respectivement `1365724` et `1234567`.

22.1.3 Arbres binaires stricts

Pour représenter des arbres dont les nœuds internes ont tous deux fils non vides, on va utiliser le type

```
type ('a, 'b) arbre_bin = Feuille of 'a
    | NoeudI of ('a, 'b) arbre_bin * 'b * ('a, 'b) arbre_bin
```

Ce type permet de représenter naturellement des expressions arithmétiques comme :

```
NoeudI(NoeudI( Feuille 2, '+', Feuille 3), '*', Feuille 5)
```

Question

Écrire une fonction `evalue : (int, char) arbre_bin -> int` qui évalue une telle expression.

Corrigé

```
let rec evalue a = match a with
    | Feuille n -> n
    | NoeudI(g, x, d) ->
            let eg = evalue g in
            let ed = evalue d in
            match x with
            | '+' -> eg + ed
            | '*' -> eg * ed
            | _ -> failwith "Ope inconnue"
```

Si on considère le type

```
type ('a, 'b) etiquette = F of 'a | N of 'b
```

On peut passer d'un `('a, 'b) arbre_bin` à un `('a, 'b) etiquette arbre` et, dans certains cas, d'un `('a, 'b) etiquette arbre` à un `('a, 'b) arbre_bin`.

Question

Écrire ainsi deux fonctions de conversion :

```
arbre_bin_vers_arbre : ('a, 'b) arbre_bin -> ('a, 'b) etiquette arbre
arbre_vers_arbre_bin : ('a, 'b) etiquette arbre -> ('a, 'b) arbre_bin option
```

Corrigé

```
let rec arbre_bin_vers_arbre a =
    match a with
    | Feuille n -> Noeud(Nil, F n, Nil)
    | NoeudI(g, x, d) -> Noeud(arbre_bin_vers_arbre g,
                               N x,
                               arbre_bin_vers_arbre d)

let rec arbre_vers_arbre_bin a =
    match a with
    | Noeud(Nil, F f, Nil) -> Some (Feuille f)
    | Noeud(g, N x, d) -> begin
            match arbre_vers_arbre_bin g, arbre_vers_arbre_bin d with
            | Some g', Some d' -> Some (NoeudI(g', x, d'))
            | _ -> None
        end
    | _ -> None
```

22.1.4 Dessin d'arbres

On va réaliser dans cette partie une fonction de dessin d'arbres avec `graphics`. L'idée est de placer la racine et de dessiner les sous-arbres gauche et droit en dessous. Pour cela, il va falloir connaître la taille de ces sous-arbres en pixels.

On pourra consulter la documentation de `graphics` ici : Graphics

Question

Écrire une fonction `pixels : int arbre -> int * int` qui renvoie un couple `(largeur, racine)` où `largeur` est la largeur en pixels d'un arbre dans son affichage et `racine` l'abscisse de la racine dans cet affichage.

Corrigé

```
let rnoeud = 5 (* Le rayon du noeud *)
let marge = 2 (* la marge autour du noeud *)

let rec pixels a = match a with
    | Nil -> 0, 0
    | Noeud(g, _, d) ->
            let lg, _ = pixels g in
            let ld, _ = pixels d in
            lg + ld + 2*(marge + rnoeud), lg + marge + rnoeud
```

Question

En déduire une fonction `dessine : int arbre -> int -> int -> unit` telle que `dessine a x y` dessine l'arbre a en plaçant la racine en (x, y).

Corrigé

```
let rec dessine a x y =
    match a with
    | Nil -> ()
    | Noeud(g, e, d) ->
            (* On précalcule les tailles des sous-arbres *)
            let lg, rg = pixels g in
```

```
            let _, rd = pixels d in
            (* position de la racine gauche relativement à (x,y) *)
            let v_rg = x - lg - marge - rnoeud + rg in
            (* position de la racine droite relative (x, y) *)
            let v_rd = x + marge + rnoeud + rd in
            let dec_y = y - 2 * rnoeud - marge in

            moveto x y; lineto v_rg dec_y;
            moveto x y; lineto v_rd dec_y;
            fill_circle x y rnoeud;

            let s = string_of_int e in
            let tw, th = text_size s in
            moveto (x-tw/2) (y-th/2);
            set_color white;
            draw_string s;
            set_color black;

            dessine g v_rg dec_y;
            dessine d v_rd dec_y
```

Les nœuds sont placés de gauche à droite dans l'ordre infixe, espacés de `2*(marge+rnoeud)` pixels. Avec des valeurs aussi petites pour `rnoeud`, les étiquettes débordent des nœuds : on pourra les augmenter.

22.1.5 Génération aléatoire d'arbres

Dans cette partie, ce qui nous intéresse est de générer aléatoirement des arbres pour en observer la structure. Les étiquettes ne sont pas pertinentes, on pourra donc au choix, soit considérer des `unit arbre`, soit redéfinir un type d'arbres non étiquetés.

On va réaliser ici deux modèles de génération aléatoire d'arbres. Le premier modèle consiste, pour générer un arbre de $n \geq 1$ nœuds, à choisir aléatoirement k dans $[\![0, n-1]\!]$, à générer aléatoirement un arbre g à k nœuds et un arbre d à $n-1-k$ nœuds puis à renvoyer l'arbre `Noeud(g, (), d)`.

Question

Implémenter ce modèle avec une fonction `genere_arbre_1 : int -> unit arbre`.

Tester votre fonction, notamment avec l'affichage de la partie précédente.

Corrigé

```
let rec genere_arbre_1 n =
    if n = 0
    then Nil
    else let k = Random.int n in
        Noeud(genere_arbre_1 k, (),
            genere_arbre_1 (n-1-k))
```

Pour le deuxième modèle, on va choisir un sous-arbre vide uniformément et le remplacer par un nœud.

Question

Écrire une fonction `chemins_vides : 'a arbre -> chemin list` qui étant donné un arbre renvoie la liste des chemins permettant d'aboutir à un sous-arbre vide.

Corrigé

```
let chemins_vides a =
    let rec aux a l =
        match a with
        | Nil -> [ List.rev l ]
```

```
        | Noeud(g, _, d) ->
              aux g (Gauche::l)
            @ aux d (Droite::l)
    in aux a []
```

Un arbre à n nœuds a $n+1$ sous-arbres vides (c'est le théorème sur les arbres stricts appliqué à l'arbre où l'on remplace chaque arbre vide par une feuille) : cette liste est de longueur $n+1$.

Question

Écrire une fonction `remplit : 'a arbre -> chemin -> 'a arbre` qui remplace un sous-arbre vide donné par son chemin par un nœud.

Corrigé

```
let rec remplit a c =
    match a, c with
    | Nil, [] -> Noeud(Nil, (), Nil)
    | Noeud(g,x,d), Gauche::q ->
            Noeud(remplit g q, x, d)
    | Noeud(g,x,d), Droite::q ->
            Noeud(g, x, remplit d q)
    | _ -> failwith "Chemin invalide"
```

Question

En déduire une fonction `genere_arbre_2 : int -> unit arbre` qui prend un entier `n` et construit un arbre à n nœuds en partant de l'arbre vide et en réalisant n remplacements d'un arbre vide par un nœud, le choix de l'arbre vide se faisant uniformément.

Note : on pourra utiliser `List.nth` et `Random.int`. Voir par exemple Random.

Corrigé

```
let rec genere_arbre_2 n =
    if n = 0 then Nil
    else
        let a = genere_arbre_2 (n-1) in
        let l = chemins_vides a in
        let k = Random.int (List.length l) in
        let ch = List.nth l k in
        remplit a ch
```

Remarque 22.2 Le deuxième modèle correspond exactement à la forme d'un arbre binaire de recherche obtenu en insérant successivement les éléments d'une permutation aléatoire uniforme : à chaque étape, la nouvelle valeur tombe avec la même probabilité dans chacun des intervalles délimités par les valeurs déjà présentes, c'est-à-dire dans chacun des sous-arbres vides. On pourra comparer expérimentalement les hauteurs moyennes des arbres obtenus par les deux modèles.

22.2 Arbres non binaires, tries

22.2.1 Arbres non binaires

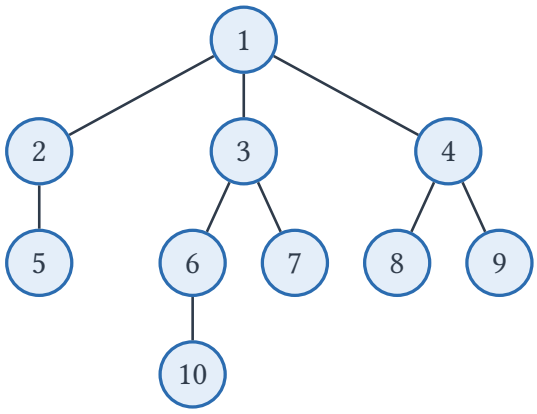
On va considérer ici les arbres d'arité quelconque avec des listes d'enfants pour les nœuds.

```
type 'a arbre = { etiquette : 'a; enfants : 'a arbre list }
```

Afin d'effectuer des tests, on pourra considérer l'arbre suivant :

```
let t = { etiquette = 1;
    enfants = [
        { etiquette = 2; enfants = [
            { etiquette = 5; enfants = [] }
            ] };
        { etiquette = 3; enfants = [
            { etiquette = 6; enfants = [
                { etiquette = 10 ; enfants = []}
            ]};
            { etiquette = 7; enfants = []}
        ] };
        { etiquette = 4; enfants = [
                { etiquette = 8; enfants = []};
                { etiquette = 9; enfants = []}
        ] }
    ] }
```

qui correspond à la représentation :



Comme présenté dans le cours, on peut réaliser des fonctions mutuellement récursives pour travailler avec ces arbres et sur la forêt induite par les enfants d'un nœud :

```
let rec taille_arbre a =
    1 + taille_foret a.enfants
    and taille_foret l = match l with
        | [] -> 0
        | t::q -> taille_arbre t + taille_foret q
```

Question

Écrire une fonction `hauteur_arbre : 'a arbre -> int` qui calcule la hauteur d'un arbre sur le modèle de la fonction précédente. On fera attention au fait qu'il n'est pas possible de représenter l'arbre vide mais la forêt vide jouera le même rôle vis-à-vis de -1.

Corrigé

```
let rec hauteur_arbre a =
    1 + hauteur_foret a.enfants
    and hauteur_foret l = match l with
    | [] -> -1
    | t::q -> max (hauteur_arbre t) (hauteur_foret q)
```

Sur l'arbre `t`, on obtient une taille de 10 et une hauteur de 3.

Question

À l'aide d'un parcours récursif, écrire une fonction `affiche_etiquettes : int arbre -> unit` qui affiche toutes les étiquettes d'un `int arbre`.

Corrigé

```
let rec affiche_etiquettes a =
    Printf.printf "%d\n" a.etiquette;
    affiche_foret a.enfants
    and affiche_foret l = match l with
        | [] -> ()
        | t::q -> affiche_etiquettes t; affiche_foret q
```

22.2.2 Représentation par des arbres binaires

On va maintenant revenir ici sur la représentation d'un `'a arbre` en tant qu'arbre binaire. Pour cela, on va avoir besoin d'étiquettes factices pour les nouveaux nœuds qui ne servent qu'à supporter la structure des nœuds d'arité > 2. On va ainsi les représenter par des `'a option arbre_bin` où l'étiquette `None` sera celle des nœuds factices.

```
type 'a arbre_bin = Noeud of 'a arbre_bin * 'a * 'a arbre_bin | Nil
```

Question

On va commencer par écrire une fonction `peigne : 'a option arbre_bin list -> 'a option arbre_bin` qui va transformer une forêt d'arbres binaires en sa représentation en peigne à l'aide de nœuds d'étiquettes `None`.

Note Dans le cas où la liste est vide, on renverra l'arbre vide et si c'est un singleton, on renverra directement l'arbre qu'il contient.

Corrigé

```
let rec peigne l =
    match l with
    | [] -> Nil
    | [x] -> x
    | a::q -> Noeud(a, None, peigne q)
```

Question

Écrire une fonction `convert : 'a arbre -> 'a option arbre_bin` qui va réaliser la conversion vue en cours.

Corrigé

```
let rec convert a =
    match a.enfants with
    | [] -> Noeud (Nil, Some a.etiquette, Nil)
    | a1::q -> Noeud(convert a1, Some a.etiquette,
                        peigne (List.map convert q))
```

Afin de tester votre fonction, on vous donne une fonction d'affichage textuel d'un tel arbre :

```
let rec print_arbre_bin pref t =
    match t with
    | Nil -> Printf.printf "%snil\n%!" pref
    | Noeud(g,x,d) -> Printf.printf "%s%s\n%!" pref
                (match x with None -> "*" | Some n -> string_of_int n);
        print_arbre_bin (String.make (String.length pref) ' '
                        ^ "|" ^ String.make 5 '-') g;
```

```
        print_arbre_bin (String.make (String.length pref) ' '
                         ^ "|" ^ String.make 5 '-') d
```

L'affichage de `print_arbre_bin "" (convert t)` doit correspondre à la représentation suivante :

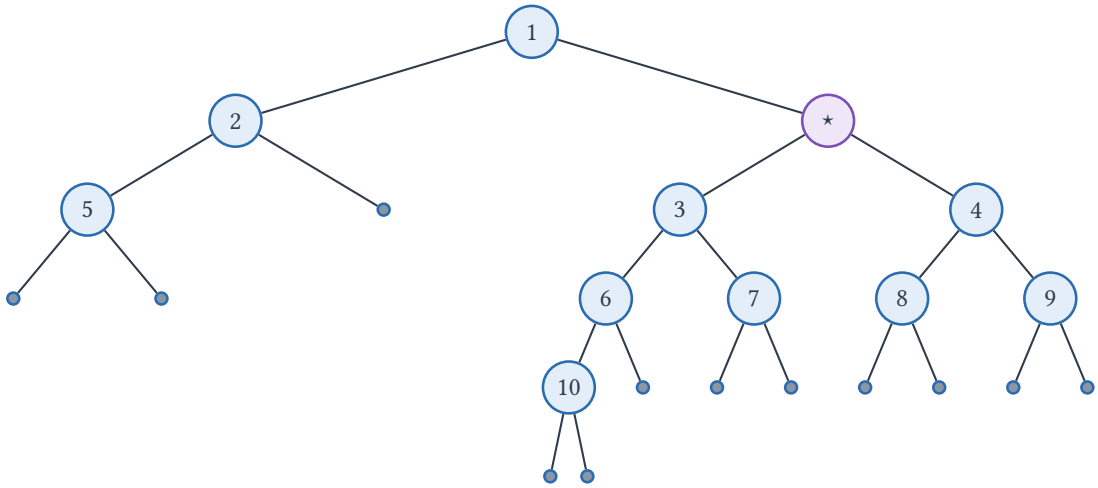


Figure 22.1 — Arbre binaire obtenu par conversion ; ⋆ (violet) est le nœud factice d'étiquette None.

22.3 Tries

On va reprendre les tries vus dans le cours pour représenter un ensemble de mots. Pour cela, on réutilise le type :

```
type trie = {
    mot : bool;
    enfants : (char * trie) list
    }
```

22.3.1 Premières manipulations

Question

Écrire une fonction `taille : trie -> int` qui renvoie le nombre de nœuds d'un trie et une fonction `nb_mots : trie -> int` qui renvoie le nombre de mots d'un trie. On s'inspirera directement des fonctions de la première partie du TP.

Corrigé

```
let rec taille a =
    1 + taille_foret a.enfants
    and taille_foret l = match l with
        | [] -> 0
        | (_, t) :: q -> taille t + taille_foret q

let rec nb_mots a =
    (if a.mot then 1 else 0) + nb_mots_foret a.enfants
    and nb_mots_foret l = match l with
        | [] -> 0
        | (_, t) :: q -> nb_mots t + nb_mots_foret q
```

Question

Écrire une fonction `trie_apres_char` telle que l'appel à `trie_apres_char l c` va renvoyer `Some t'` s'il y a un couple `(c, t')` dans la liste `l` et `None` sinon. On pourra s'en servir avec `trie_apres_char t.enfants c` pour trouver le trie potentiel dont l'arête est étiquetée par `c`.

Corrigé

```
let rec trie_apres_char (l:(char * trie) list) (c:char) =
    match l with
    | [] -> None
    | (c',t)::q -> if c = c' then Some t else trie_apres_char q c
```

Question

En déduire une fonction `contient : trie -> string -> bool` qui teste si un mot est présent dans le trie.

Corrigé

```
let contient (t:trie) (s:string) : bool =
    let rec aux t i =
        if i = String.length s
        then t.mot
        else match trie_apres_char t.enfants s.[i] with
            | None -> false
            | Some t' -> aux t' (i+1)
    in aux t 0
```

Question

Écrire une fonction `enumere : trie -> string list` qui renvoie tous les mots contenus dans un trie par un parcours récursif.

Indication : on pourra utiliser :

- `List.concat` pour passer d'une liste de listes à la concaténée de ses éléments. Ainsi `List.concat [ [1;2]; [3]; [4;5] ]` va renvoyer `[1;2;3;4;5]` ;
- `String.make 1 c` transforme le caractère `c` en chaîne de caractères ;
- on rappelle que `^` permet de concaténer deux chaînes.

Corrigé

```
let enumere (t:trie) =
    let rec aux t acc =
        let l = List.concat (List.map (fun (c,t') ->
                aux t' (acc ^ String.make 1 c)) t.enfants) in
        if t.mot then acc :: l else l
    in aux t ""
```

22.3.2 Ajout d'un mot à un trie

Pour effectuer l'ajout d'un mot à un trie, on va commencer par écrire une fonction qui étant donné une chaîne de caractères `s` $= c_0 \ldots c_{n-1}$ et un entier $i \leq n$ va renvoyer le trie `t` correspondant à la branche

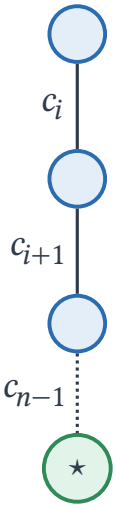


Figure 22.2 — Branche du trie pour le suffixe $c_i \ldots c_{n-1}$; ⋆ marque la fin du mot.

Question

Écrire la fonction `suffixe : int -> string -> trie`.

Corrigé

```
let rec suffixe (i:int) (s:string) : trie =
    let n = String.length s in
    if i = n
    then { mot = true; enfants = [] }
    else { mot = false; enfants = [ (s.[i], suffixe (i+1) s) ] }
```

Question

Écrire une fonction `avec_trie_pour_char : (char * trie) list -> char -> trie -> (char * trie) list` telle que `avec_trie_pour_char l c t` va renvoyer la liste obtenue en remplaçant `(c,t')` par `(c,t)` dans `l` si un tel couple est présent, ou en ajoutant le couple `(c,t)` à la fin sinon.

Corrigé

```
let rec avec_trie_pour_char l c t =
    match l with
    | [] -> [ (c, t) ]
    | (c', t') :: q -> if c = c' then (c, t) :: q
                       else (c', t') :: avec_trie_pour_char q c t
```

Question

Écrire une fonction `ajoute : trie -> string -> trie` qui renvoie le trie obtenu en ajoutant un nouveau mot.

Corrigé

```
let ajoute t s =
    let n = String.length s in
    let rec aux t i =
        if i = n
        then { t with mot = true }
        else let t' = match trie_apres_char t.enfants s.[i] with
                    | None -> suffixe (i+1) s
                    | Some t' -> aux t' (i+1) in
```

```
            { t with enfants = avec_trie_pour_char t.enfants s.[i] t' }
    in aux t 0
```

On va utiliser la fonction précédente pour retrouver le trie donné dans le cours.

```
let rec trie_of_list l = match l with
    | [] -> { mot = false; enfants = [] }
    | s::q -> ajoute (trie_of_list q) s
```

Ainsi `trie_of_list ["bar"; "barbe"; "art"; "ami"; "abri"; "abricot"; "abonne"]` est un trie à 21 nœuds contenant 7 mots.

On va maintenant récupérer une grande liste de mots dans le fichier `wordlist` récupérable à l'adresse https://raw.githubusercontent.com/dwyl/english-words/master/words_alpha.txt à raison d'un mot par ligne :

```
let wordlist = let rec lit_fichier f =
        try
            let s = input_line f in
                s :: lit_fichier f
            with End_of_file -> []
        in
        lit_fichier (open_in "wordlist")
```

Question

En déduire un trie permettant de représenter cette wordlist et comparez :

- le nombre de caractères total ;
- le nombre de nœuds du trie.

Que peut-on en déduire ?

Corrigé

```
let taille_totale = List.fold_left (+) 0 (List.map String.length wordlist) in
taille_totale, taille (trie_of_list wordlist)
```

On constate qu'il y a nettement moins de nœuds que de caractères au total : les préfixes communs ne sont stockés qu'une fois. Attention cependant à ne pas sous-estimer la taille d'un nœud qui dépasse très largement celle d'un caractère. Cependant, le trie apporte la possibilité de chercher un mot m en $O(|\,m\,|)$ (à alphabet fixé) et donc indépendamment du nombre de mots.

Remarquons que `lit_fichier` et `trie_of_list` ne sont pas récursives terminales : sur une très grande liste, on pourra préférer `List.fold_left ajoute { mot = false; enfants = [] } wordlist`.

22.4 Arbres binaires en C, ABR, arbres rouges et noirs

22.4.1 Arbres binaires en C

En C, un nœud d'un arbre binaire va avoir exactement la même structure qu'un maillon d'une liste doublement chaînée : une valeur et deux pointeurs vers d'autres nœuds, ici ce ne sont pas `suivant` et `precedent` mais `gauche` et `droite` :

```
struct noeud {
    struct noeud *gauche;
    struct noeud *droite;
    int valeur;
};
```

```
typedef struct noeud *arbre;
```

Comme pour les listes, l'arbre vide sera représenté par le pointeur `NULL`.

Question

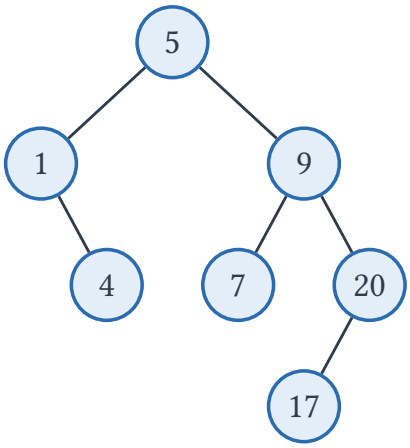
Écrire une fonction `arbre cree_feuille(int x)` qui alloue et crée une feuille de valeur `x`.

Corrigé

```
arbre cree_feuille(int x)
{
    arbre n = malloc(sizeof(struct noeud));
    n->valeur = x;
    n->gauche = n->droite = NULL;
    return n;
}
```

Question

En déduire une représentation en C de l'arbre :




```
Pour cela, on vous suggère de créer autant de feuilles que de nœuds et
d'effectuer les liens ensuite avec les pointeurs.

Cet arbre nous servira d'exemple pour vérifier les fonctions suivantes.
```

Corrigé

```
arbre n5 = cree_feuille(5);
arbre n1 = cree_feuille(1);
arbre n4 = cree_feuille(4);
arbre n9 = cree_feuille(9);
arbre n7 = cree_feuille(7);
arbre n20 = cree_feuille(20);
arbre n17 = cree_feuille(17);
n5->gauche = n1;
n1->droite = n4;
n5->droite = n9;
n9->gauche = n7;
n9->droite = n20;
n20->gauche = n17;

arbre a = n5;
```

Question

Écrire une fonction `int taille(arbre a)` qui renvoie le nombre de nœuds d'un arbre binaire.

Corrigé

```
int taille(arbre a)
{
    if (a == NULL)
        return 0;
    return 1 + taille(a->gauche) + taille(a->droite);
}
```

Question

Écrire une fonction `int hauteur(arbre a)` qui renvoie la hauteur d'un arbre binaire.

Corrigé

```
int max(int a, int b)
{
    if (a > b) return a;
    else return b;
}

int hauteur(arbre a)
{
    if (a == NULL)
        return -1;
    return 1 + max(hauteur(a->gauche), hauteur(a->droite));
}
```

Sur l'arbre exemple, on obtient une taille de 7 et une hauteur de 3.

Question

Écrire des fonctions `void affiche_prefixe(arbre a)`, `affiche_infixe` et `affiche_postfixe` qui affichent les valeurs des nœuds de l'arbre `a` selon le traitement correspondant dans un parcours.

Écrire une fonction `void affiche(arbre a)` qui, à l'aide des trois traitements, permet d'afficher un arbre avec des triplets comme nœuds. Par exemple, pour l'arbre précédent, on pourra afficher :

```
((nil,1,(nil,4,nil)),5,((nil,7,nil),9,((nil,17,nil),20,nil)))
```

Corrigé

```
void affiche_prefixe(arbre a)
{
    if (a != NULL)
    {
        printf("%d ", a->valeur);
        affiche_prefixe(a->gauche);
        affiche_prefixe(a->droite);
    }
}

void affiche_infixe(arbre a)
{
    if (a != NULL)
    {
        affiche_infixe(a->gauche);
```

```
        printf("%d ", a->valeur);
        affiche_infixe(a->droite);
    }
}

void affiche_postfixe(arbre a)
{
    if (a != NULL)
    {
        affiche_postfixe(a->gauche);
        affiche_postfixe(a->droite);
        printf("%d ", a->valeur);
    }
}

void affiche(arbre a)
{
    if (a != NULL)
    {
        printf("(");
        affiche(a->gauche);
        printf(",%d,", a->valeur);
        affiche(a->droite);
        printf(")");
    } else printf("nil");
}
```

Sur l'arbre exemple, on obtient `5 1 4 9 7 20 17` dans l'ordre préfixe, `1 4 5 7 9 17 20` dans l'ordre infixe et `4 1 7 17 20 9 5` dans l'ordre postfixe.

Question

(Optionnel en première lecture) À l'aide d'une file dans un tableau de longueur $|a|$, réaliser une fonction `void affiche_largeur(arbre a)` qui affiche les valeurs des nœuds de l'arbre `a` lors d'un parcours en largeur.

Corrigé

```
void affiche_largeur(arbre a)
{
    if (a == NULL)
        return;
    arbre *file = malloc(sizeof(arbre) * taille(a));
    file[0] = a;
    int debut = 0;
    int fin = 1;
    while(fin > debut)
    {
        arbre r = file[debut];
        debut++;
        printf("%d ", r->valeur);
        if(r->gauche != NULL)
        {
            file[fin] = r->gauche;
            fin++;
        }
        if(r->droite != NULL)
        {
            file[fin] = r->droite;
            fin++;
        }
    }
    free(file);
}
```

Chaque nœud étant enfilé exactement une fois, un tableau de longueur $|a|$ suffit. Sur l'arbre exemple, on obtient `5 1 9 4 7 20 17`.

Question

Écrire une fonction `void libere(arbre a)` qui libère tous les nœuds de l'arbre `a`. On fera attention à l'ordre des libérations avec `free` par rapport aux déréférencements de structure (les `->`).

Corrigé

```
void libere(arbre a)
{
    if (a != NULL)
    {
        libere(a->gauche);
        libere(a->droite);
        free(a);
    }
}
```

22.4.2 Arbres binaires de recherche

On conserve le type précédent, mais on considère maintenant qu'on a des arbres binaires de recherche.

Question

Écrire une fonction `arbre recherche(arbre a, int x)` qui renvoie le premier nœud rencontré d'étiquette `x` dans un ABR ou `NULL` s'il n'est pas présent.

Corrigé

```
arbre recherche(arbre a, int x)
{
    if (a == NULL)
        return NULL;
    if (a->valeur == x)
        return a;
    if (a->valeur < x)
        return recherche(a->droite, x);
    return recherche(a->gauche, x);
}
```

Question

Écrire une fonction `arbre insere(arbre a, int x)` qui renvoie la racine de l'ABR obtenu en insérant une nouvelle feuille d'étiquette `x`.

Corrigé

```
arbre insere(arbre a, int x)
{
    if (a == NULL)
        return cree_feuille(x);

    if (a->valeur >= x)
    {
        a->gauche = insere(a->gauche, x);
    }
    else
    {
```

```
        a->droite = insere(a->droite, x);
    }

    return a;
}
```

Contrairement à la version `OCaml` du cours, cette fonction modifie l'arbre en place : l'ancien arbre n'est plus disponible.

Question

Écrire une fonction `arbre minimum(arbre a)` qui renvoie le nœud de valeur minimale de `a` en allant le plus à gauche dans l'arbre `a`. On supposera `a` non vide.

Corrigé

```
arbre minimum(arbre a)
{
    if (a->gauche == NULL)
        return a;
    return minimum(a->gauche);
}
```

Question

(Optionnel en première lecture) Écrire une fonction `arbre supprime(arbre a, int x)` qui renvoie la racine de l'ABR obtenu en supprimant le premier nœud rencontré d'étiquette `x`.

Corrigé

On suit la méthode du cours : si le nœud a un fils vide, on le remplace par l'autre fils ; sinon on remplace sa valeur par le minimum de son sous-arbre droit, qu'on supprime de ce sous-arbre.

```
arbre supprime(arbre a, int x)
{
    if (a == NULL)
        return a;
    if (a->valeur == x)
    {
        if (a->gauche == NULL)
        {
            arbre a_p = a->droite;
            free(a);
            return a_p;
        }
        if (a->droite == NULL)
        {
            arbre a_p = a->gauche;
            free(a);
            return a_p;
        }
        int min_valeur = minimum(a->droite)->valeur;
        a->droite = supprime(a->droite, min_valeur);
        a->valeur = min_valeur;
    } else {
        if (a->valeur > x)
            a->gauche = supprime(a->gauche, x);
        else
            a->droite = supprime(a->droite, x);
    }
```

```
    return a;
}
```

22.4.3 Arbres rouges et noirs

On va rajouter un champ au type précédent pour indiquer la couleur d'un nœud (il faut inclure `stdbool.h`) :

```
struct noeud {
    struct noeud *gauche;
    struct noeud *droite;
    bool rouge;
    int valeur;
};
typedef struct noeud *arbre;
```

On vous conseille ici de copier le fichier précédent et de faire un nouveau programme.

Question

Reprendre `cree_feuille`, en l'appelant `cree_noeud`, pour que les nouveaux nœuds soient rouges par défaut.

Corrigé

```
arbre cree_noeud(int x)
{
    arbre n = malloc(sizeof(struct noeud));
    n->valeur = x;
    n->rouge = true;
    n->gauche = n->droite = NULL;
    return n;
}
```

Question

Écrire des fonctions `arbre rotation_droite(arbre a)`, `arbre rotation_gauche(arbre a)` et `arbre bascule(arbre a)` qui effectuent une des trois opérations en renvoyant la nouvelle racine.

Corrigé

```
arbre rotation_droite(arbre x)
{
    arbre y = x->gauche;
    x->gauche = y->droite;
    y->droite = x;
    y->rouge = x->rouge;
    x->rouge = true;
    return y;
}

arbre rotation_gauche(arbre y)
{
    arbre x = y->droite;
    y->droite = x->gauche;
    x->gauche = y;
    x->rouge = y->rouge;
    y->rouge = true;
    return x;
}

arbre bascule(arbre a)
{
    a->rouge = true;
```

```
    a->gauche->rouge = false;
    a->droite->rouge = false;
    return a;
}
```

Question

Écrire des fonctions `bool rouge(arbre a)` et `bool noir(arbre a)` qui testent la couleur de la racine. On fera attention au fait que l'arbre vide est noir.

Corrigé

```
bool rouge(arbre a)
{
    if (a == NULL) return false;
    return a->rouge;
}

bool noir(arbre a)
{
    return !rouge(a);
}
```

Question

Écrire des fonctions `arbre essai_rotgauche(arbre a)`, `essai_rotdroite` et `essai_bascule` qui n'effectuent ces opérations que dans les cas nécessaires pour l'insertion.

Corrigé

```
arbre essai_bascule(arbre a)
{
    if(rouge(a->gauche) && rouge(a->droite))
        return bascule(a);
    return a;
}

arbre essai_rotgauche(arbre a)
{
    if(rouge(a->droite) && rouge(a->droite->droite))
        return rotation_gauche(a);
    return a;
}

arbre essai_rotdroite(arbre a)
{
    if(rouge(a->gauche) && noir(a->droite))
        return rotation_droite(a);
    return a;
}
```

Dans `essai_rotgauche`, l'évaluation paresseuse de `&&` garantit que `a->droite->droite` n'est évalué que si `a->droite` est rouge, donc non vide.

Question

En déduire une fonction `arbre insere(arbre a, int x)` qui effectue l'insertion dans un arbre rouge et noir. On pourra définir une fonction auxiliaire `arbre insere_aux(arbre a, int x)` qui effectue l'insertion et qu'on appellera dans `insere` avant de faire le noircissement final de la racine.

Corrigé

```
arbre insere_aux(arbre a, int x)
{
    if (a == NULL)
        return cree_noeud(x);
    if (a->valeur > x)
        a->gauche = insere_aux(a->gauche, x);
    else
        a->droite = insere_aux(a->droite, x);
    return essai_bascule(essai_rotgauche(essai_rotdroite(a)));
}

arbre insere(arbre a, int x)
{
    arbre a2 = insere_aux(a, x);
    a2->rouge = false;
    return a2;
}
```

Pour tester, on pourra écrire une fonction qui vérifie les invariants (racine noire, pas de fils gauche rouge, pas de nœud rouge de fils droit rouge, même nombre de nœuds noirs sur chaque branche) et l'appeler après chaque insertion, par exemple en insérant $0, 1, \ldots, n-1$ dans cet ordre : la hauteur doit rester logarithmique.

TRAVAUX PRATIQUES 23

Arbres binaires

DANS CE CHAPITRE

23.1 Premières fonctions
23.2 Parcours
23.3 Première évaluation d'expression arithmétique
23.4 Arbres binaires stricts
23.5 Dessin d'arbres
23.6 Génération aléatoire d'arbres

23.1 Premières fonctions

On va considérer le type des arbres binaires `'a arbre_bin` défini par :

```
type 'a arbre_bin =
    | Nil
    | Noeud of 'a arbre_bin * 'a * 'a arbre_bin
```

Question 1 Écrire des fonctions permettant de calculer la taille et la hauteur d'un arbre, ainsi qu'une fonction permettant de déterminer si un arbre est réduit à une feuille.

Tests

```
assert (taille Nil = 0);
assert (taille (Noeud(Nil,3,Nil)) = 1);
assert (taille (Noeud(Nil,'a',Nil)) = 1);
assert (taille (Noeud(Nil,'a',Noeud(Nil,'b',Nil))) = 2);
"Valide";;

assert (hauteur Nil = -1);
assert (hauteur (Noeud(Nil,3,Nil)) = 0);
assert (hauteur (Noeud(Nil,'a',Nil)) = 0);
assert (hauteur (Noeud(Nil,'a',Noeud(Nil,'b',Nil))) = 1);
"Valide";;

assert (not (feuille Nil));
assert (feuille (Noeud(Nil,3,Nil)));
assert (feuille (Noeud(Nil,'a',Nil)));
assert (not (feuille (Noeud(Nil,'a',Noeud(Nil,'b',Nil)))));
"Valide";;
```

Corrigé

```
let rec taille (a : 'a arbre_bin) : int = match a with
    | Nil -> 0
    | Noeud(g,x,d) -> 1 + taille g + taille d

let rec hauteur (a : 'a arbre_bin) : int = match a with
    | Nil -> -1
    | Noeud(g,x,d) ->
            1 + max (hauteur g) (hauteur d)

let feuille (a : 'a arbre_bin) : bool = match a with
    | Nil -> false
    | Noeud(g, x, d) -> g = Nil && d = Nil
```

Question 2 Dans cette question, on va identifier un nœud à son étiquette.

Déterminer des fonctions renvoyant, sous forme d'une liste d'étiquettes, les nœuds, les feuilles et les nœuds internes d'un arbre.

Allez regarder les tests si vous avez un doute sur ce qui doit être renvoyé.

Tests

```
assert(noeuds Nil = []);
assert(noeuds (Noeud(Nil,1,Nil)) = [1]);
let a = Noeud(
    Noeud(Nil, 2, Noeud(Nil, 3, Nil)),
    4, Noeud(Noeud(Nil, 5, Nil), 6,
        Noeud(Nil, 7, Nil))) in
assert(List.sort Stdlib.compare (noeuds a)
    = [2;3;4;5;6;7]);
"Valide";;

assert(feuilles Nil = []);
assert(feuilles (Noeud(Nil,1,Nil)) = [1]);
let a = Noeud(
    Noeud(Nil, 2, Noeud(Nil, 3, Nil)),
    4, Noeud(Noeud(Nil, 5, Nil), 6,
        Noeud(Nil, 7, Nil))) in
assert(List.sort Stdlib.compare (feuilles a)
= [3;5;7]);
"Valide";;

assert(noeuds_internes Nil = []);
assert(noeuds_internes (Noeud(Nil,1,Nil)) = []);
let a = Noeud(
    Noeud(Nil, 2, Noeud(Nil, 3, Nil)),
    4, Noeud(Noeud(Nil, 5, Nil), 6,
        Noeud(Nil, 7, Nil))) in
assert(List.sort Stdlib.compare
    (noeuds_internes a)
= [2;4;6]);
"Valide";;
```

Corrigé

```
let rec noeuds (a : 'a arbre_bin) : 'a list =
    match a with
    | Nil -> []
    | Noeud(g, x, d) ->
        x :: (noeuds g @ noeuds d)

let rec feuilles (a : 'a arbre_bin) : 'a list =
    match a with
    | Nil -> []
```

```
    | Noeud(g, x, d) ->
        if feuille a then [x]
        else feuilles g @ feuilles d

let rec noeuds_internes (a : 'a arbre_bin) =
    match a with
    | Nil -> []
    | Noeud(Nil, _, Nil) -> []
    | Noeud(g, x, d) ->
        x :: (noeuds_internes g @ noeuds_internes d)
```

Question 3 Écrire une fonction permettant de déterminer si un arbre binaire est strict.

Tests

```
assert(not (est_strict Nil));
assert(est_strict (Noeud(Nil, 1, Nil)));
let a = Noeud(
    Noeud(Nil, 2, Noeud(Nil, 3, Nil)),
    4, Noeud(Noeud(Nil, 5, Nil), 6,
        Noeud(Nil, 7, Nil))) in
assert(not (est_strict a));
let b = Noeud(
    Noeud(Nil, 2, Nil),
    4, Noeud(Noeud(Nil, 5, Nil), 6,
        Noeud(Nil, 7, Nil))) in
assert(est_strict b);
"Valide";;
```

Corrigé

```
let rec est_strict (a : 'a arbre_bin) : bool =
    match a with
    | Nil -> false
    | Noeud(Nil, x, Nil) -> true
    | Noeud(g, x, d) -> g <> Nil && d <> Nil && est_strict g && est_strict d
```

Question 4 Écrire une fonction qui renvoie les sous-arbres d'un arbre, en dupliquant les sous-arbres identiques.

Tests

```
assert(sous_arbres Nil = [ Nil ]);
let a = Noeud(Nil, 1, Nil) in
assert(List.sort Stdlib.compare (sous_arbres a)
    = [ Nil; Nil; a ]);
let b = Noeud(Nil, 2, Nil) in
let c = Noeud(a, 3, b) in
assert(List.sort Stdlib.compare
    (sous_arbres c) =
    [Nil; Nil; Nil; Nil; a; b; c]);
"Valide";;
```

Corrigé

```
let rec sous_arbres (a : 'a arbre_bin) : 'a arbre_bin list =
    match a with
    | Nil -> [Nil]
    | Noeud(g, x, d) ->
       a :: (sous_arbres g @ sous_arbres d)
```

Pour accéder à un nœud de l'arbre, on descend en partant de la racine et en allant à gauche ou à droite. On peut donc représenter un tel chemin par une liste de déplacements :

```
type deplacement = Gauche | Droite
type chemin = deplacement list
```

Question 5 Écrire une fonction `chemin_noeud : 'a arbre_bin -> chemin -> 'a option` qui renvoie l'étiquette d'un nœud donné par son chemin. On utilise un type `option` en cas de chemin invalide.

Tests

```
let a = Noeud(
    Noeud(Nil, 2, Noeud(Nil, 3, Nil)),
    4, Noeud(Noeud(Nil, 5, Nil), 6,
        Noeud(Nil, 7, Nil))) in
assert(chemin_noeud a [Gauche; Gauche] = None);
assert(chemin_noeud a [] = Some 4);
assert(chemin_noeud a [Droite; Gauche] = Some 5);
assert(chemin_noeud a [Droite; Droite] = Some 7);
assert(chemin_noeud a [Droite; Droite; Droite] = None);
"Valide";;
```

Corrigé

```
let rec chemin_noeud (a : 'a arbre_bin) (l : chemin) : 'a option =
    match a, l with
    | Nil, _ -> None
    | Noeud(g,x,d), Gauche::q -> chemin_noeud g q
    | Noeud(g,x,d), Droite::q -> chemin_noeud d q
    | Noeud(g,x,d), [] -> Some x
```

23.2 Parcours

Sur le même modèle que les fonctions précédentes, on peut explorer un arbre en explorant à gauche puis à droite ses sous-arbres de manière récursive. Chaque nœud est donc vu trois fois :

- une première fois quand on le découvre avant d'explorer son sous-arbre gauche
- entre les deux explorations
- enfin quand on a fini d'explorer son sous-arbre droit.

Question 6 Écrire des fonctions `affiche_avant`, `affiche_milieu` et `affiche_apres` qui parcourent et affichent les étiquettes d'un `string arbre_bin` selon les trois cas précédents. *Note : il s'agit essentiellement du même code à une ligne près.*

Tests

```
let t = Noeud( Noeud(Nil, "1", Nil), "2",
    Noeud(Nil, "3", Noeud(Nil, "4", Nil))) in
affiche_avant t (* doit afficher 2134 *);;

let t = Noeud( Noeud(Nil, "1", Nil), "2",
    Noeud(Nil, "3", Noeud(Nil, "4", Nil))) in
affiche_milieu t (* doit afficher 1234 *);;

let t = Noeud( Noeud(Nil, "1", Nil), "2",
    Noeud(Nil, "3", Noeud(Nil, "4", Nil))) in
affiche_apres t (* doit afficher 1432 *);;
```

Corrigé

```
let rec affiche_avant a = match a with
    | Nil -> ()
    | Noeud(g,x,d) ->
        print_string x;
        affiche_avant g;
        affiche_avant d

let rec affiche_milieu a = match a with
    | Nil -> ()
    | Noeud(g,x,d) ->
        affiche_milieu g;
        print_string x;
        affiche_milieu d

let rec affiche_apres a = match a with
    | Nil -> ()
    | Noeud(g,x,d) ->
        affiche_apres g;
        affiche_apres d;
        print_string x
```

23.3 Première évaluation d'expression arithmétique

On va considérer des expressions comme $(2+3)\times 5$ représentées par des `string arbre_bin` ainsi :

```
let e = Noeud(
    Noeud( Noeud(Nil, "2", Nil),
           "+",
           Noeud(Nil, "3", Nil)),
    "*",
    Noeud(Nil, "5", Nil))
```

Question 7 Dessiner cet arbre. Évaluer les trois fonctions précédentes sur cet arbre. Qu'en déduire ?

Corrigé

La racine est étiquetée par `"*"`, son fils gauche par `"+"` (avec les feuilles `"2"` et `"3"`) et son fils droit est la feuille `"5"`. On obtient :

- `affiche_avant e` : `*+235`, c'est l'écriture **préfixe** de l'expression ;
- `affiche_milieu e` : `2+3*5`, c'est l'écriture **infixe** usuelle, mais les parenthèses ont disparu : on ne peut plus distinguer $(2+3)\times 5$ de $2+(3\times 5)$;
- `affiche_apres e` : `23+5*`, c'est l'écriture **postfixe**, qu'on sait évaluer avec une pile (voir le TP sur les piles et files).

Les écritures préfixe et postfixe permettent de retrouver l'arbre, et donc l'expression, sans parenthèses ; ce n'est pas le cas de l'écriture infixe.

Question 8 Écrire une fonction d'évaluation qui calcule le résultat de cette expression.

On pourra utiliser `int_of_string` pour convertir une chaîne de caractères comme `"42"` en l'entier `42`.

Tests

```
let e = Noeud(Noeud( Noeud(Nil, "2", Nil), "+", Noeud(Nil, "3", Nil)), "*", Noeud(Nil, "5",
Nil)) in
assert(evalue e = 25);
"Valide";;
```

Corrigé

```
let rec evalue a = match a with
    | Nil -> failwith "Vide"
    | Noeud(Nil,x,Nil) -> int_of_string x
    | Noeud(g, x, d) ->
            let eg = evalue g in
            let ed = evalue d in
            match x with
            | "+" -> eg + ed
            | "*" -> eg * ed
            | _ -> failwith "Ope inconnue"
```

23.4 Arbres binaires stricts

Pour représenter des arbres dont les nœuds internes ont tous deux fils non vides, on va utiliser le type :

```
type ('a, 'b) arbre_strict = Feuille of 'a
    | NoeudI of ('a, 'b) arbre_strict * 'b * ('a, 'b) arbre_strict
```

Ce type permet de représenter naturellement des expressions arithmétiques comme :

```
type operations = Plus | Mult
```

```
let e = NoeudI(
    NoeudI(Feuille 2, Plus, Feuille 3),
    Mult,
    Feuille 5)
```

Question 9 Écrire une fonction `evalue : (int, operations) arbre_strict -> int` qui évalue une telle expression.

Tests

```
let e = NoeudI(NoeudI( Feuille 2, Plus, Feuille 3), Mult, Feuille 5) in
assert(evalue e = 25);
"Valide";;
```

Corrigé

```
let rec evalue a = match a with
    | Feuille n -> n
    | NoeudI(g, x, d) ->
            let eg = evalue g in
            let ed = evalue d in
            match x with
            | Plus -> eg + ed
            | Mult -> eg * ed
```

Si on considère le type

```
type ('a, 'b) etiquette = F of 'a | N of 'b
```

on peut passer d'un `('a, 'b) arbre_strict` à un `('a, 'b) etiquette arbre_bin` et, dans certains cas, d'un `('a, 'b) etiquette arbre_bin` à un `('a, 'b) arbre_strict`.

Question 10 Écrire ainsi deux fonctions de conversion :

```
arbre_strict_vers_arbre_bin : ('a, 'b) arbre_strict -> ('a, 'b) etiquette arbre_bin
arbre_bin_vers_arbre_strict : ('a, 'b) etiquette arbre_bin -> ('a, 'b) arbre_strict option
```

Tests

```
let asvab = arbre_strict_vers_arbre_bin in
assert(asvab (Feuille 1) = Noeud(Nil, F 1, Nil));
assert(asvab (NoeudI(Feuille 'a', 1, NoeudI(Feuille 'b', 2, Feuille 'c')))
    = Noeud(Noeud(Nil,F 'a',Nil), N 1, Noeud(Noeud(Nil,F 'b',Nil), N 2, Noeud(Nil, F 'c',
Nil))));
"Valide";;

let abvas = arbre_bin_vers_arbre_strict in
assert(None = abvas Nil);
assert(None = abvas (Noeud(Nil, N 1, Nil)));
assert(None = abvas (Noeud(Nil, N 1, Noeud(Nil, F 2, Nil))));
assert(Some (Feuille 1) = abvas (Noeud(Nil, F 1, Nil)));
assert(Some (NoeudI(Feuille 'a', 1, NoeudI(Feuille 'b', 2, Feuille 'c')))
    = abvas (Noeud(Noeud(Nil,F 'a',Nil), N 1, Noeud(Noeud(Nil,F 'b',Nil), N 2, Noeud(Nil, F
'c', Nil)))));
"Valide";;
```

Corrigé

```
let rec arbre_strict_vers_arbre_bin a =
    match a with
    | Feuille n -> Noeud(Nil, F n, Nil)
    | NoeudI(g, x, d) -> Noeud(arbre_strict_vers_arbre_bin g,
                               N x,
                               arbre_strict_vers_arbre_bin d)

let rec arbre_bin_vers_arbre_strict a =
    match a with
    | Noeud(Nil, F f, Nil) -> Some (Feuille f)
    | Noeud(g, N x, d) -> begin
            match arbre_bin_vers_arbre_strict g, arbre_bin_vers_arbre_strict d with
            | Some g', Some d' -> Some (NoeudI(g', x, d'))
            | _ -> None
        end
    | _ -> None
```

23.5 Dessin d'arbres

On va réaliser dans cette partie une fonction de dessin d'arbres avec la bibliothèque `graphics`. L'idée est de placer la racine et de dessiner les sous-arbres gauche et droit en dessous. Pour cela, il va falloir connaître la taille de ces sous-arbres en pixels.

On pourra consulter la documentation de `graphics` ici : Graphics

Question 11 Écrire une fonction `pixels : 'a arbre_bin -> int * int` qui renvoie un couple `(largeur, racine)` où `largeur` est la largeur en pixels d'un arbre dans son affichage et `racine` l'abscisse de la racine dans cet affichage.

Corrigé

```
let rnoeud = 5 (* Le rayon du noeud *)
let marge = 2 (* la marge autour du noeud *)

let rec pixels a = match a with
    | Nil -> 0, 0
    | Noeud(g, _, d) ->
            let lg, _ = pixels g in
            let ld, _ = pixels d in
            lg + ld + 2*(marge + rnoeud), lg + marge + rnoeud
```

Question 12 En déduire une fonction `dessine : string arbre_bin -> int -> int -> unit` telle que `dessine a x y` dessine l'arbre a en plaçant la racine en (x, y).

On pourra utiliser `draw_string` pour dessiner l'étiquette, et `text_size` pour connaître les dimensions en pixels d'une chaîne.

Corrigé

```
let rec dessine a x y =
    match a with
    | Nil -> ()
    | Noeud(g, e, d) ->
            (* On précalcule les tailles des sous-arbres *)
            let lg, rg = pixels g in
            let _, rd = pixels d in
            (* position de la racine gauche relativement à (x,y) *)
            let v_rg = x - lg - marge - rnoeud + rg in
            (* position de la racine droite relative (x, y) *)
            let v_rd = x + marge + rnoeud + rd in
            let dec_y = y - 2 * rnoeud - marge in

            moveto x y; lineto v_rg dec_y;
            moveto x y; lineto v_rd dec_y;
            fill_circle x y rnoeud;

            (* dimensions en pixels du texte de l'étiquette *)
            let tw, th = text_size e in
            moveto (x-tw/2) (y-th/2);
            set_color white;
            draw_string e;
            set_color black;

            dessine g v_rg dec_y;
            dessine d v_rd dec_y
```

23.6 Génération aléatoire d'arbres

Dans cette partie, ce qui nous intéresse est de générer aléatoirement des arbres pour en observer la structure. Les étiquettes ne sont pas pertinentes, on pourra donc au choix, soit considérer des `unit arbre_bin`, soit redéfinir un type d'arbres non étiquetés.

On va réaliser ici deux modèles de génération aléatoire d'arbres.

23.6.1 Premier modèle

Le premier modèle consiste, pour générer un arbre de $n \geq 1$ nœuds, à choisir aléatoirement k dans $[\![0, n-1]\!]$, à générer aléatoirement un arbre g à k nœuds et un arbre d à $n-1-k$ nœuds puis à renvoyer l'arbre `Noeud(g, (), d)`. Pour $n = 0$, on renvoie l'arbre vide.

Question 13 Implémenter ce modèle avec une fonction `genere_arbre_1 : int -> unit arbre_bin`.

Tester votre fonction, notamment avec l'affichage de la partie précédente.

Corrigé

`Random.int n` renvoie un entier uniforme dans $[\![0, n-1]\!]$.

```
let rec genere_arbre_1 n =
    if n = 0
    then Nil
    else begin
        let k = Random.int n in
        Noeud(genere_arbre_1 k, (), genere_arbre_1 (n - 1 - k))
    end
```

23.6.2 Second modèle

Pour le deuxième modèle, on va choisir un sous-arbre vide uniformément et le remplacer par un nœud.

Question 14 Écrire une fonction `chemins_vides : 'a arbre_bin -> chemin list` qui, étant donné un arbre, renvoie la liste des chemins permettant d'aboutir à un sous-arbre vide.

Corrigé

```
let rec chemins_vides a =
    match a with
    | Nil -> [ [] ]
    | Noeud(g, _, d) ->
        List.map (fun c -> Gauche :: c) (chemins_vides g)
        @ List.map (fun c -> Droite :: c) (chemins_vides d)
```

Un arbre à n nœuds a exactement $n+1$ sous-arbres vides : la liste renvoyée est de longueur $n+1$.

Question 15 Écrire une fonction `remplit : unit arbre_bin -> chemin -> unit arbre_bin` qui remplace un sous-arbre vide, donné par son chemin, par un nœud.

Corrigé

```
let rec remplit a c =
    match a, c with
    | Nil, [] -> Noeud(Nil, (), Nil)
    | Noeud(g, y, d), Gauche :: q -> Noeud(remplit g q, y, d)
    | Noeud(g, y, d), Droite :: q -> Noeud(g, y, remplit d q)
    | _ -> failwith "chemin invalide"
```

Question 16 En déduire une fonction `genere_arbre_2 : int -> unit arbre_bin` qui prend un entier `n` et construit un arbre à n nœuds en partant de l'arbre vide et en réalisant n remplacements d'un arbre vide par un nœud, le choix de l'arbre vide se faisant uniformément.

Note : on pourra utiliser `List.nth` et `Random.int`. Voir par exemple Random.

Corrigé

```
let genere_arbre_2 n =
    let a = ref Nil in
    for _ = 1 to n do
        let vides = chemins_vides !a in
        let c = List.nth vides (Random.int (List.length vides)) in
        a := remplit !a c
    done;
    !a
```

TRAVAUX PRATIQUES 24

Arbres et tries

DANS CE CHAPITRE

24.1 Arbres non binaires, tries
24.2 Tries

24.1 Arbres non binaires, tries

24.1.1 Arbres non binaires

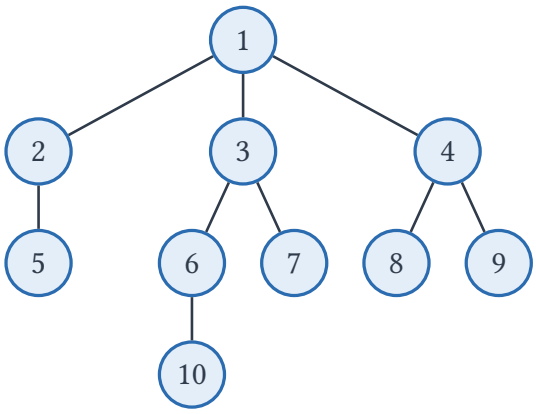
On va considérer ici les arbres d'arité quelconque avec des listes d'enfants pour les nœuds.

```
type 'a arbre = {
    etiquette : 'a;
    enfants : 'a arbre list
}
```

Afin d'effectuer des tests, on pourra considérer l'arbre suivant :

```
let t = { etiquette = 1;
    enfants = [
        { etiquette = 2; enfants = [
            { etiquette = 5; enfants = [] }
            ] };
        { etiquette = 3; enfants = [
            { etiquette = 6; enfants = [
                { etiquette = 10 ; enfants = []}
            ]};
            { etiquette = 7; enfants = []}
        ] };
        { etiquette = 4; enfants = [
                { etiquette = 8; enfants = []};
                { etiquette = 9; enfants = []}
        ] }
    ] }
```

qui correspond à la représentation :



Comme présenté dans le cours, on peut réaliser des fonctions mutuellement récursives pour travailler avec ces arbres et sur la forêt induite par les enfants d'un nœud :

```
let rec taille_arbre a =
    1 + taille_foret a.enfants
and taille_foret l = match l with
    | [] -> 0
    | t::q -> taille_arbre t + taille_foret q
```

Question 1 Écrire une fonction `hauteur_arbre : 'a arbre -> int` qui calcule la hauteur d'un arbre sur le modèle de la fonction précédente. On fera attention au fait qu'il n'est pas possible de représenter l'arbre vide mais la forêt vide jouera le même rôle vis-à-vis de -1.

Corrigé

```
let rec hauteur_arbre a =
    1 + hauteur_foret a.enfants
    and hauteur_foret l = match l with
    | [] -> -1
    | t::q -> max (hauteur_arbre t) (hauteur_foret q)
```

Sur l'arbre d'exemple, on obtient `3`.

Question 2 À l'aide d'un parcours récursif, écrire une fonction `affiche_etiquettes : int arbre -> unit` qui affiche toutes les étiquettes d'un `int arbre`.

Corrigé

```
let rec affiche_etiquettes a =
    Printf.printf "%d\n" a.etiquette;
    affiche_foret a.enfants
    and affiche_foret l = match l with
        | [] -> ()
        | t::q -> affiche_etiquettes t; affiche_foret q
```

24.1.2 Représentation par des arbres binaires

On va maintenant revenir sur la représentation d'un `'a arbre` en tant qu'arbre binaire. Pour cela, on va avoir besoin d'étiquettes factices pour les nouveaux nœuds qui ne servent qu'à supporter la structure des nœuds d'arité > 2. On va ainsi les représenter par des `'a option arbre_bin` où l'étiquette `None` sera celle des nœuds factices.

```
type 'a arbre_bin = Noeud of 'a arbre_bin * 'a * 'a arbre_bin | Nil
```

Question 3 On va commencer par écrire une fonction `peigne : 'a option arbre_bin list -> 'a option arbre_bin` qui va transformer une forêt d'arbres binaires en sa représentation en peigne à l'aide de nœuds d'étiquettes `None`.

Note Dans le cas où la liste est vide, on renverra l'arbre vide et si c'est un singleton, on renverra directement l'arbre qu'il contient.

Corrigé

```
let rec peigne l =
    match l with
    | [] -> Nil
    | [x] -> x
    | a::q -> Noeud(a, None, peigne q)
```

Question 4 Écrire une fonction `convert : 'a arbre -> 'a option arbre_bin` qui va réaliser la conversion vue en classe : le premier enfant devient le fils gauche et les enfants suivants, rassemblés en peigne, le fils droit.

Corrigé

```
let rec convert a =
    match a.enfants with
    | [] -> Noeud (Nil, Some a.etiquette, Nil)
    | a1::q -> Noeud(convert a1, Some a.etiquette,
            peigne (List.map convert q))
```

Afin de tester votre fonction, on vous donne une fonction d'affichage textuel d'un tel arbre :

```
let rec print_arbre_bin pref t =
    match t with
    | Nil -> Printf.printf "%snil\n%!" pref
    | Noeud(g,x,d) -> Printf.printf "%s%s\n%!" pref
                (match x with None -> "*" | Some n -> string_of_int n);
       print_arbre_bin (String.make (String.length pref) ' '
                        ^ "|" ^ String.make 5 '-') g;
       print_arbre_bin (String.make (String.length pref) ' '
                        ^ "|" ^ String.make 5 '-') d
```

L'affichage de `print_arbre_bin "" (convert t)` doit correspondre à la représentation suivante, où ⋆ est un nœud factice d'étiquette `None` :

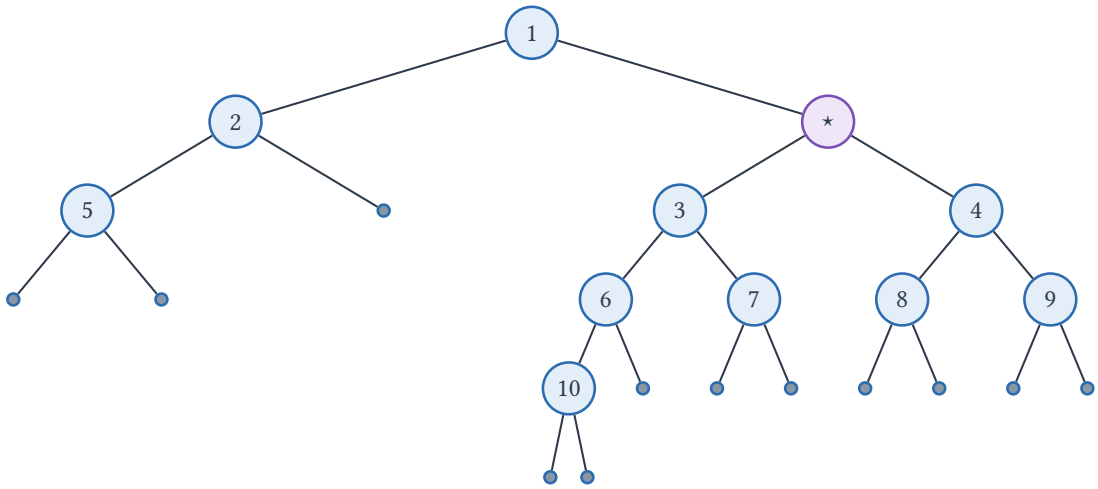


Figure 24.1 — Arbre binaire obtenu par conversion ; ⋆ (violet) est le nœud factice d'étiquette None.

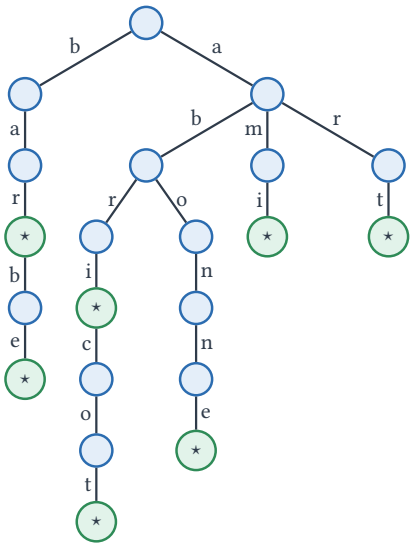
24.2 Tries

On considère les tries vus en classe, on définit ici un type spécifique :

```
type trie = {
    mot : bool;
    enfants : (char * trie) list
    }
```

- Le champ `mot` va permettre d'indiquer si le nœud finit un mot (l'$\star$ de la représentation) ou non.
- Le champ `enfants` présente la liste des enfants avec l'étiquette de l'arête qui y mène.

Par exemple, le trie suivant, vu en cours, représente l'ensemble de mots bar, barbe, art, ami, abri, abricot, abonne :



24.2.1 Premières manipulations

Question 5 Écrire une fonction `taille : trie -> int` qui renvoie le nombre de nœuds d'un trie et une fonction `nb_mots : trie -> int` qui renvoie le nombre de mots d'un trie. On s'inspirera directement des fonctions de la première partie du TP.

Corrigé

```
let rec taille a =
    1 + taille_foret a.enfants
    and taille_foret l = match l with
        | [] -> 0
        | (_, t) :: q -> taille t + taille_foret q
let rec nb_mots a =
    (if a.mot then 1 else 0) + nb_mots_foret a.enfants
    and nb_mots_foret l = match l with
        | [] -> 0
        | (_, t) :: q -> nb_mots t + nb_mots_foret q
```

Sur le trie précédent, on obtient 21 nœuds et 7 mots.

Question 6 Écrire une fonction `trie_apres_char` telle que l'appel à `trie_apres_char l c` renvoie `Some t'` s'il y a un couple `(c, t')` dans la liste `l` et `None` sinon. On pourra s'en servir avec `trie_apres_char t.enfants c` pour trouver le trie potentiel dont l'arête est étiquetée par `c`.

Corrigé

```
let rec trie_apres_char (l:(char * trie) list) (c:char) =
    match l with
    | [] -> None
    | (c',t)::q -> if c = c' then Some t else trie_apres_char q c
```

Question 7 En déduire une fonction `contient : trie -> string -> bool` qui teste si un mot est présent dans le trie.

Corrigé

```
let contient (t:trie) (s:string) : bool =
    let rec aux t i =
        if i = String.length s
        then t.mot
        else match trie_apres_char t.enfants s.[i] with
            | None -> false
            | Some t' -> aux t' (i+1)
    in aux t 0
```

Question 8 Écrire une fonction `enumere : trie -> string list` qui renvoie tous les mots contenus dans un trie par un parcours récursif.

Indication : on pourra utiliser :

- `List.concat` pour passer d'une liste de listes à la concaténée de ses éléments. Ainsi `List.concat [ [1;2]; [3]; [4;5] ]` va renvoyer `[1;2;3;4;5]`.
- `String.make 1 c` transforme le caractère `c` en chaîne de caractères.
- On rappelle que ^ permet de concaténer deux chaînes.

Corrigé

```
let enumere (t:trie) =
    let rec aux t acc =
        let l = List.concat (List.map (fun (c,t') ->
                aux t' (acc ^ String.make 1 c)) t.enfants) in
        if t.mot then acc :: l else l
    in aux t ""
```

24.2.2 Ajout d'un mot à un trie

Pour effectuer l'ajout d'un mot à un trie, on va commencer par écrire une fonction qui, étant donnés une chaîne de caractères $s = c_0 \dots c_{n-1}$ et un entier $i \leq n$, va renvoyer le trie `t` correspondant à la branche :

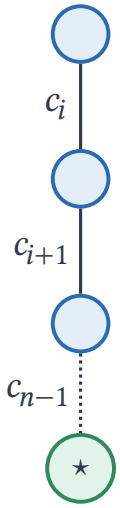


Figure 24.2 — Branche du trie pour le suffixe $c_i \ldots c_{n-1}$; ⋆ marque la fin du mot.

c'est-à-dire le trie ne contenant que le mot $c_i \ldots c_{n-1}$ (le mot vide si $i = n$).

Question 9 Écrire la fonction `suffixe : int -> string -> trie`.

Corrigé

```
let rec suffixe (i:int) (s:string) : trie =
    let n = String.length s in
    if i = n
    then { mot = true; enfants = [] }
    else { mot = false; enfants = [ (s.[i], suffixe (i+1) s) ] }
```

Question 10 Écrire une fonction `avec_trie_pour_char : (char * trie) list -> char -> trie -> (char * trie) list` telle que `avec_trie_pour_char l c t` renvoie la liste obtenue en remplaçant `(c,t')` par `(c,t)` dans `l` si un tel couple est présent, ou en ajoutant le couple `(c,t)` à la fin sinon.

Corrigé

```
let rec avec_trie_pour_char l c t =
    match l with
    | [] -> [ (c, t) ]
    | (c', t') :: q -> if c = c' then (c, t) :: q else (c', t') :: avec_trie_pour_char q c t
```

Question 11 Écrire une fonction `ajoute : trie -> string -> trie` qui renvoie le trie obtenu en ajoutant un nouveau mot.

Corrigé

```
let ajoute t s =
    let n = String.length s in
    let rec aux t i =
        if i = n
        then { t with mot = true }
        else let t' = match trie_apres_char t.enfants s.[i] with
                    | None -> suffixe (i+1) s
                    | Some t' -> aux t' (i+1) in
            { t with enfants = avec_trie_pour_char t.enfants s.[i] t' }
    in aux t 0
```

On va utiliser la fonction précédente pour retrouver le trie donné en début de partie, avec `trie_of_list ["bar"; "barbe"; "art"; "ami"; "abri"; "abricot"; "abonne"]` :

```
let rec trie_of_list l = match l with
    | [] -> { mot = false; enfants = [] }
    | s::q -> ajoute (trie_of_list q) s
```

On va maintenant récupérer une grande liste de mots dans le fichier dictionnaire_fr (un mot par ligne).

```
let dictionnaire = let rec lit_fichier f =
        try
            let s = input_line f in
                s :: lit_fichier f
            with End_of_file -> []
        in
        lit_fichier (open_in "dictionnaire_fr")
```

Question 12 En déduire un trie permettant de représenter ce dictionnaire et comparer :

- le nombre de caractères total ;
- le nombre de nœuds du trie.

Que peut-on en déduire ?

Est-ce vraiment précis comme comparaison ?

Corrigé

Le dictionnaire contient 605 835 mots, soit 7 585 815 caractères en comptant les sauts de ligne, et le trie possède 2 681 196 nœuds : le partage des préfixes divise par presque 3 le nombre de « cases ».

La comparaison n'est pas précise : un caractère occupe un octet alors qu'un nœud du trie est un enregistrement contenant un booléen, une liste de couples, des pointeurs… En mémoire, le trie occupe en réalité bien plus de place que le texte.

On va essayer d'améliorer l'estimation précédente en stockant le trie dans un fichier.

Question 13 Pour cette question, on va écrire un fichier binaire contenant le trie, on parle de sérialisation.

Allez voir les documentations des fonctions suivantes : `open_out_bin`, `output_char`, `output_byte`.

On va écrire un trie ainsi :

- un octet qui contient $mot + arite$ où $mot = 0$ si ce n'est pas un mot et 128 sinon et $arite$ est l'arité du nœud qu'on supposera toujours < 128
- pour chaque enfant
  - le caractère de l'arête
  - l'écriture de l'enfant (écriture récursive donc)

1. En faisant cela, comparez les tailles des deux représentations du dictionnaire.
2. Comparez avec un `zip` du dictionnaire.
3. Est-ce décevant ?

Corrigé

```
let rec ecrit_trie out t =
    let arite = List.length t.enfants in
    assert(arite < 128);
    output_byte out ((if t.mot then 128 else 0) + arite);
    List.iter (fun (c,a) ->
        output_char out c;
        ecrit_trie out a) t.enfants

let sauve_trie nom t =
    let out = open_out_bin nom in
```

```
    ecrit_trie out t;
    close_out out
```

Chaque nœud occupe un octet, plus un octet pour l'arête qui y mène (sauf la racine), soit $2 \times 2\,681\,196 - 1$ octets : le fichier fait environ 5,4 Mo contre 7,6 Mo pour le texte. Un `zip` du dictionnaire fait environ 2 Mo : notre représentation, qui n'exploite que le partage des préfixes, est nettement moins bonne qu'un algorithme de compression généraliste (qui exploite aussi les suffixes communs et les fréquences des lettres, voir le TP sur le codage de Huffman).

25

TRAVAUX PRATIQUES 25

Arbres binaires de recherche

DANS CE CHAPITRE

25.1 Arbres binaires de recherche
25.2 Arbres 2-3 et 2-3-4

25.1 Arbres binaires de recherche

On considère le type suivant :

```
type 'a abr =
    | Noeud of 'a abr * 'a * 'a abr
    | Nil
```

On reprend la définition du cours : un arbre binaire est un arbre binaire de recherche (abr) s'il est vide ou si ses deux sous-arbres sont des abr et si l'étiquette x de sa racine vérifie $\max_{y\in g} y \leq x \leq \min_{y\in d} y$, où g et d sont les sous-arbres gauche et droit (on autorise donc les doublons).

Question 1 Écrire une fonction `etendue` qui à tout arbre `a` **non vide** renvoie le couple `(min_v, max_v)` constitué des valeurs minimale et maximale de ses étiquettes.

Corrigé

```
let rec etendue a =
    match a with
    | Nil -> failwith "Arbre vide"
    | Noeud(Nil,x,Nil) -> (x,x)
    | Noeud(a',x,Nil) | Noeud(Nil,x,a') ->
        let min_a', max_a' = etendue a' in
        (min min_a' x, max max_a' x)
    | Noeud(g,x,d) ->
        let min_g, max_g = etendue g in
        let min_d, max_d = etendue d in
        (min (min min_g min_d) x,
         max (max max_g max_d) x)
```

Question 2 En déduire une fonction `est_abr1` qui renvoie un booléen indiquant si un arbre binaire est un abr.

Corrigé

```
let rec est_abr1 a =
    match a with
    | Nil | Noeud(Nil,_,Nil) -> true
```

```
    | Noeud(g,x,Nil) ->
        let _, max_g = etendue g in
        est_abr1 g && max_g <= x
    | Noeud(Nil,x,d) ->
        let min_d, _ = etendue d in
        est_abr1 d && x <= min_d
    | Noeud(g, x, d) ->
        let _, max_g = etendue g in
        let min_d, _ = etendue d in
        est_abr1 g && est_abr1 d && max_g <= x && x <= min_d
```

Question 3 Expliquer pourquoi cette fonction n'est pas efficace et en déduire une version `est_abr` linéaire dans le nombre de nœuds de l'arbre binaire.

Corrigé

`est_abr1` appelle `etendue` sur chaque sous-arbre, ce qui parcourt à nouveau tout ce sous-arbre : chaque nœud est visité autant de fois qu'il a d'ancêtres, d'où une complexité quadratique dans le pire cas (pour un peigne). On renvoie plutôt, en un seul parcours, le booléen et l'étendue.

```
let rec est_abr_aux a =
    match a with
    | Nil -> failwith "Arbre vide"
    | Noeud(Nil,x,Nil) -> (true, x, x)
    | Noeud(g,x,Nil) ->
        let abr_g, min_g, max_g = est_abr_aux g in
        (abr_g && max_g <= x, min_g, x)
    | Noeud(Nil,x,d) ->
        let abr_d, min_d, max_d = est_abr_aux d in
        (abr_d && x <= min_d, x, max_d)
    | Noeud(g, x, d) ->
        let abr_g, min_g, max_g = est_abr_aux g in
        let abr_d, min_d, max_d = est_abr_aux d in
        (abr_g && abr_d && max_g <= x && x <= min_d,
            min_g, max_d)

let est_abr a =
    match a with
    | Nil -> true
    | _ -> let b, _, _ = est_abr_aux a in b
```

Question 4 Écrire une fonction `contient` qui teste si une valeur est présente dans un abr.

Corrigé

Deux versions possibles, puis une version à éviter :

```
let rec contient x a =
    match a with
    | Nil -> false
    | Noeud(_,y,_) when x = y -> true
    | Noeud(g,y,_) when x < y -> contient x g
    | Noeud(_,y,d) -> contient x d

let rec contient x a =
    match a with
    | Nil -> false
    | Noeud(g,y,d) ->
            x = y || (if x < y then contient x g else contient x d)

(* à éviter : *)
let rec contient x a =
    match a with
```

```
    | Nil -> false
    | Noeud(g,y,d) ->
       if x = y
       then true (* perdu *)
       else if x < y
            then contient x g
            else if x > y
                 then contient x d
                 else false (* impossible a atteindre *)
```

Question 5 Écrire une fonction `insere` qui rajoute un élément dans un abr et renvoie le nouvel abr.

Corrigé

```
let rec insere x a =
    match a with
    | Nil -> Noeud(Nil, x, Nil)
    | Noeud(g,y,d) ->
       if x <= y
       then Noeud(insere x g, y, d)
       else Noeud(g, y, insere x d)
```

Question 6 Écrire une fonction `supprime` qui supprime un élément d'un abr et renvoie le nouvel abr. Si jamais l'élément n'est pas dedans, cette fonction se comporte comme l'identité.

On pourra définir une fonction `plus_a_droite` qui renvoie le nœud le plus à droite d'un abr.

Corrigé

```
(* On va chercher le noeud le plus à droite y et on le court-circuite en
   renvoyant le couple (y, a') où a' est a privé de y *)
let rec plus_a_droite a =
    match a with
    | Nil -> failwith "Vide"
    | Noeud(g,y,Nil) -> (y, g)
    | Noeud(g,y,d) ->
       let z, d' = plus_a_droite d in
       (z, Noeud(g,y,d'))

let rec supprime x a =
    match a with
    | Noeud(Nil, y, Nil) when x = y -> Nil
    | Noeud(g, y, Nil) when x = y -> g
    | Noeud(Nil, y, d) when x = y -> d
    | Noeud(g, y, d) when x = y ->
       let z, g' = plus_a_droite g in
       Noeud(g', z, d)
    | Noeud(g, y, d) ->
       if x < y then Noeud(supprime x g, y, d)
                else Noeud(g, y, supprime x d)
    | _ -> a
```

25.2 Arbres 2-3 et 2-3-4

On définit les deux types suivants :

```
type 'a a23 =
    | Bin23 of 'a a23 * 'a * 'a a23
    | Ter23 of 'a a23 * 'a * 'a a23 * 'a * 'a a23
    | Nil23

type 'a a234 =
```

```
    | Bin234 of 'a a234 * 'a * 'a a234
    | Ter234 of 'a a234 * 'a * 'a a234 * 'a * 'a a234
    | Quat234 of 'a a234 * 'a * 'a a234
        * 'a * 'a a234 * 'a * 'a a234
    | Nil234
```

Question 7 Écrire une fonction `insere23` qui insère un élément dans un arbre 2-3 en utilisant un arbre 2-3-4 de manière transitoire.

Corrigé

On commence par des fonctions de conversion :

```
let rec vers234 a =
    match a with
    | Nil23 -> Nil234
    | Bin23(g,x,d) -> Bin234(vers234 g, x, vers234 d)
    | Ter23(g,x,m,y,d) -> Ter234(vers234 g, x, vers234 m, y, vers234 d)

let rec vers23 a =
    match a with
    | Nil234 -> Nil23
    | Bin234(g,x,d) -> Bin23(vers23 g, x, vers23 d)
    | Ter234(g,x,m,y,d) -> Ter23(vers23 g, x, vers23 m, y, vers23 d)
    | _ -> failwith "Quaternaire"
```

On insère alors dans une feuille, ce qui peut créer un nœud quaternaire, que l'on fait remonter en l'éclatant en deux nœuds binaires et en insérant sa clé médiane dans le père (`repare234`). Si la racine devient quaternaire, on l'éclate et la hauteur augmente de 1.

```
let elim_racine_quat a =
    match a with
    | Quat234(g,x,m,y,n,z,d) -> Bin234( Bin234(g,x,m), y, Bin234(n,z,d) )
    | _ -> a

let rec repare234 a =
    match a with
    | Bin234(Quat234(g,x,m,y,n,z,d),t,d') ->
            Ter234(Bin234(g,x,m), y, Bin234(n,z,d), t, d')
    | Bin234(g',t, Quat234(g,x,m,y,n,z,d)) ->
            Ter234(g', t, Bin234(g,x,m), y, Bin234(n, z,d))
    | Ter234(Quat234(g,x,m,y,n,z,d),x',m',y',d') ->
        Quat234(Bin234(g,x,m),y,Bin234(n,z,d),x',m',y',d')
    | Ter234(g',x',Quat234(g,x,m,y,n,z,d),y',d') ->
        Quat234(g',x',Bin234(g,x,m),y,Bin234(n,z,d),y',d')
    | Ter234(g',x',m',y',Quat234(g,x,m,y,n,z,d)) ->
        Quat234(g',x',m',y',Bin234(g,x,m),y,Bin234(n,z,d))
    | _ -> a

let rec insere23_aux x a =
    match a with
    | Nil23 -> Bin234(Nil234, x, Nil234)
    | Bin23(Nil23,y,Nil23) ->
        if x <= y
        then Ter234(Nil234,x,Nil234,y,Nil234)
        else Ter234(Nil234,y,Nil234,x,Nil234)
    | Bin23(g, y, d) -> repare234
        (if x <= y then Bin234(insere23_aux x g, y, vers234 d)
                   else Bin234(vers234 g, y, insere23_aux x d))
    | Ter23(Nil23,y,Nil23,z,Nil23) ->
        if x <= y
        then Quat234(Nil234, x, Nil234, y, Nil234, z, Nil234)
        else if x <= z
            then Quat234(Nil234, y, Nil234, x, Nil234, z, Nil234)
            else Quat234(Nil234, y, Nil234, z, Nil234, x, Nil234)
```

```
    | Ter23(g,y,m,z,d) -> repare234
        (if x <= y
        then Ter234(insere23_aux x g, y, vers234 m, z, vers234 d)
        else if x <= z
            then Ter234(vers234 g, y, insere23_aux x m, z, vers234 d)
            else Ter234(vers234 g, y, vers234 m, z, insere23_aux x d))


let insere23 a x = vers23 (elim_racine_quat (insere23_aux x a))
```

26

TRAVAUX PRATIQUES 26

Tas et files de priorité en OCaml

DANS CE CHAPITRE

26.1 Tas
26.2 Tri par tas
26.3 Files de priorité

26.1 Tas

On va définir ici des fonctions dans un fichier `tas.ml`, cela permettra ensuite de l'utiliser dans un autre fichier avec `Tas` (c'est ce module qui est utilisé dans le TP sur la compression de Huffman). C'est la raison pour laquelle on ne va pas préfixer les noms de fonctions par `tas`.

On réalise ici des tas **max**, rangés dans un tableau : la racine est à l'indice 0 et les fils du nœud d'indice i sont aux indices $2i + 1$ et $2i + 2$.

On propose ici une interface assez simple mais deux améliorations sont proposées librement :

- faire en sorte de ne pas avoir à donner tout de suite un élément du bon type avant le premier ajout ;
- faire en sorte de ne pas avoir à donner une capacité initiale en ayant un tas redimensionnable comme les tableaux dynamiques.

On définit le type

```
type 'a t = {
    elements : 'a array;
    mutable taille : int
}
```

Question 1 Écrire une fonction

```
val cree : int -> 'a -> 'a t
```

telle que `cree taille_max defaut` crée un tas de capacité maximale `taille_max` et dont les éléments sont initialisés à `defaut`.

Corrigé

```
let cree taille_max defaut =
    {
        elements = Array.make taille_max defaut;
        taille = 0
    }
```

Remarque 26.1 C'est une restriction pour le typage d'avoir à passer une valeur par défaut. Vous pouvez vous en passer en changeant le type des éléments du tableau en `'a option` et en mettant `None` partout au début.

Question 2 Écrire des fonctions

```
val est_vide : 'a t -> bool
val taille : 'a t -> int
```

qui indiquent respectivement si un tas est vide et son nombre d'éléments.

Corrigé

```
let est_vide tas = tas.taille = 0

let taille tas = tas.taille
```

Question 3 Écrire des fonctions

```
val pere : int -> int
val gauche : int -> int
val droite : int -> int
```

qui indiquent les indices respectifs du père, du fils gauche et du fils droit d'un indice donné.

Corrigé

```
let pere a =
    if a = 0
    then failwith "racine"
    else (a+1)/2 - 1

let gauche a = 2*(a+1) - 1
let droite a = 2*(a+1)
```

On a bien $\text{pere}(i) = \left\lfloor \frac{i-1}{2} \right\rfloor$, $\text{gauche}(i) = 2i+1$ et $\text{droite}(i) = 2i+2$.

Question 4 Écrire une fonction

```
val maximum : 'a t -> 'a
```

qui renvoie le maximum d'un tas non vide.

Corrigé

On en profite pour définir deux fonctions utilitaires.

```
let maximum tas = tas.elements.(0)

let nth tas i = tas.elements.(i)

let swap v x y =
    let t = v.(x) in
    v.(x) <- v.(y);
    v.(y) <- t
```

Question 5 Écrire une fonction

```
val ajoute : 'a -> 'a t -> unit
```

qui rajoute un élément dans le tas.

Corrigé

On place l'élément dans la première case libre puis on le fait remonter tant qu'il est plus grand que son père.

```
let ajoute x tas =
    let pos = ref tas.taille in
    tas.elements.(tas.taille) <- x;
    tas.taille <- tas.taille + 1;
    while !pos > 0 && nth tas (pere !pos) < nth tas !pos do
        let p = pere !pos in
        swap tas.elements p !pos;
        pos := p
    done
```

Question 6 Écrire une fonction

```
val max_pere_fils : 'a t -> int -> int
```

telle que `max_pere_fils tas i` renvoie l'indice du plus grand élément entre `i`, `gauche i` et `droite i` (parmi ceux qui sont dans le tas). Ce sera utile pour la fonction suivante.

Corrigé

```
let max_pere_fils tas a =
    let max_courant = ref a in

    if tas.taille > gauche a
        && nth tas (gauche a) > nth tas !max_courant
    then max_courant := gauche a;
    if tas.taille > droite a
        && nth tas (droite a) > nth tas !max_courant
    then max_courant := droite a;
    !max_courant
```

Question 7 Écrire une fonction

```
val extrait_maximum : 'a t -> 'a
```

qui extrait et renvoie le maximum d'un tas non vide.

Corrigé

On remplace la racine par le dernier élément, puis on le fait descendre en l'échangeant avec le plus grand de ses fils tant que nécessaire.

```
let extrait_maximum tas =
    assert (tas.taille > 0);
    tas.taille <- tas.taille - 1;
    let x = tas.elements.(0) in
    if tas.taille > 0
    then begin
        let pos = ref 0 in
        tas.elements.(0) <- tas.elements.(tas.taille);
        while max_pere_fils tas !pos <> !pos do
            let max_fils = max_pere_fils tas !pos in
            swap tas.elements !pos max_fils;
            pos := max_fils
        done
```

```
    end;
    x
```

26.2 Tri par tas

Question 8 Écrire une fonction

```
val tri_par_tas : 'a list -> 'a list
```

qui trie une liste dans l'ordre croissant à l'aide d'un tas.

Corrigé

```
let tri_par_tas l =
    match l with
    | [] -> []
    | x :: _ ->
        let tas = cree (List.length l) x in
        List.iter (fun y -> ajoute y tas) l;
        (* on extrait les maxima : on construit la liste
           triée en commençant par la fin *)
        let res = ref [] in
        while not (est_vide tas) do
            res := extrait_maximum tas :: !res
        done;
        !res
```

Chaque ajout et chaque extraction coûte $O(\log n)$, le tri coûte donc $O(n \log n)$.

Comme on l'a vu en classe, on peut réaliser un tri par tas en place dans un tableau.

Question 9 Écrire des fonctions

```
val vers_tas : 'a array -> unit
val depuis_tas : 'a array -> unit
```

qui vont respectivement :

- prendre un tableau et le transformer *en place* en tableau d'un tas ;
- prendre un tableau d'un tas et le transformer *en place* en tableau trié dans l'ordre croissant.

Corrigé

```
(* descente de l'élément d'indice i dans le tas formé
   des n premières cases de t *)
let rec descend t n i =
    let m = ref i in
    if gauche i < n && t.(gauche i) > t.(!m) then m := gauche i;
    if droite i < n && t.(droite i) > t.(!m) then m := droite i;
    if !m <> i
    then begin
        swap t i !m;
        descend t n !m
    end

let vers_tas t =
    let n = Array.length t in
    (* les feuilles sont déjà des tas : on traite les nœuds
       internes du dernier vers la racine *)
    for i = n/2 - 1 downto 0 do
        descend t n i
    done
```

```
let depuis_tas t =
    let n = Array.length t in
    for k = n-1 downto 1 do
        (* le maximum du tas t.(0..k) va à sa place définitive *)
        swap t 0 k;
        descend t k 0
    done
```

Pour trier un tableau `t` en place, il suffit d'appeler `vers_tas t` puis `depuis_tas t`.

26.3 Files de priorité

On va définir ici un module dans `filePriorite.ml` qui permettra donc d'accéder à ces fonctions avec `FilePriorite`.

Comme il faut utiliser `Tas`, vous pouvez passer par `dune` pour gérer les modules (voir le TP sur la compression de Huffman) ou compiler à la main.

On considère le type :

```
type ('a, 'b) t = ('b * 'a) Tas.t
```

Un élément de type `'a` de priorité de type `'b` est ainsi stocké dans le tas sous la forme du couple `(priorite, element)` : comme les couples sont comparés d'abord selon leur première composante, la racine du tas est un élément de priorité maximale.

Question 10 Écrire une fonction

```
val cree : int -> 'a -> 'b -> ('a, 'b) t
```

telle que `cree taille_max defval defprio` crée une file de priorité de taille maximale `taille_max` et dont les éléments sont initialisés à la valeur `defval` de priorité `defprio`.

Remarque 26.2 Là aussi, on peut faire mieux avec une fonction `cree : unit -> ('a, 'b) t` si on le souhaite.

Question 11 Écrire une fonction

```
val ajoute : ('a, 'b) t -> 'a -> 'b -> unit
```

qui ajoute un élément avec une priorité donnée.

Question 12 Écrire une fonction

```
val retire : ('a, 'b) t -> 'a
```

qui retire un élément quelconque de priorité maximale.

Corrigé des questions 10 à 12

```
type ('a, 'b) t = ('b * 'a) Tas.t

let cree taille_max defval defprio =
    Tas.cree taille_max (defprio, defval)

let ajoute file x p =
    Tas.ajoute (p, x) file

let retire file =
```

```
    let _, x = Tas.extrait_maximum file in
    x
```

On remarque qu'à priorités égales, ce sont les éléments eux-mêmes qui départagent les couples : il faut donc que le type `'a` soit comparable avec `<` (pas de fonctions par exemple).

27

TRAVAUX PRATIQUES 27

Arbres binaires en C

DANS CE CHAPITRE

27.1 Arbres binaires
27.2 Parcours

27.1 Arbres binaires

On va considérer le type suivant pour des arbres binaires en C :

```c
struct noeud {
    struct noeud *gauche;
    struct noeud *droite;
    int valeur;
};

typedef struct noeud arbre;
```

L'arbre vide est alors représenté par la valeur `NULL`, et un arbre est manipulé par un pointeur `arbre *` vers sa racine, alloué sur le tas.

Pour construire des arbres de test, on pourra utiliser la fonction suivante :

```c
arbre *noeud(arbre *g, int v, arbre *d)
{
    arbre *a = malloc(sizeof(arbre));
    a->gauche = g;
    a->valeur = v;
    a->droite = d;
    return a;
}
```

Par exemple, l'arbre de racine 5, de fils gauche 1 (lui-même de fils droit 4) et de fils droit 9 (de fils 7 et 20) s'obtient avec :

```c
arbre *a = noeud(noeud(NULL, 1, noeud(NULL, 4, NULL)), 5,
                 noeud(noeud(NULL, 7, NULL), 9, noeud(NULL, 20, NULL)));
```

Question 1

Écrire une fonction de prototype

```c
int taille(arbre *a);
```

qui renvoie la taille d'un arbre binaire, c'est-à-dire son nombre de nœuds.

Corrigé

```c
int taille(arbre *a)
{
    if (a == NULL)
        return 0;
    return 1 + taille(a->gauche) + taille(a->droite);
}
```

Question 2

Écrire une fonction de prototype

```c
int hauteur(arbre *a);
```

qui renvoie la hauteur d'un arbre binaire.

Corrigé

Avec la convention du cours, l'arbre vide est de hauteur -1 (une feuille est donc de hauteur 0).

```c
int max(int a, int b)
{
    return a > b ? a : b;
}

int hauteur(arbre *a)
{
    if (a == NULL)
        return -1;
    return 1 + max(hauteur(a->gauche), hauteur(a->droite));
}
```

Sur l'exemple, on obtient une taille de 6 et une hauteur de 2.

Question 3

Écrire une fonction de prototype

```c
arbre *parfait(int n);
```

qui renvoie un arbre binaire parfait de hauteur `n`. On mettra des valeurs `1` comme étiquettes partout pour simplifier.

Corrigé

Un arbre parfait de hauteur $n \geq 0$ a pour sous-arbres deux arbres parfaits de hauteur $n-1$; l'arbre parfait de hauteur -1 est l'arbre vide.

```c
arbre *parfait(int n)
{
    if (n == -1)
        return NULL;
    return noeud(parfait(n - 1), 1, parfait(n - 1));
}
```

On vérifie par récurrence que `parfait(n)` a $2^{n+1} - 1$ nœuds. Les deux appels récursifs sont nécessaires : avec un seul appel dont on réutiliserait le résultat pour les deux fils, les deux sous-arbres seraient partagés, et la libération de l'arbre libérerait deux fois les mêmes nœuds.

Question 4

Écrire une fonction qui libère toute la mémoire occupée par un arbre binaire.

Corrigé

```c
void libere(arbre *a)
{
    if (a != NULL)
    {
        libere(a->gauche);
        libere(a->droite);
        free(a); // en dernier : après free, on ne peut plus lire a->gauche
    }
}
```

27.2 Parcours

Question 5

Écrire des fonctions réalisant un affichage des étiquettes des nœuds suivant les trois ordres : préfixe, postfixe et infixe.

Corrigé

Seule la position de l'affichage par rapport aux deux appels récursifs change.

```c
void affiche_prefixe(arbre *a)
{
    if (a != NULL)
    {
        printf("%d ", a->valeur);
        affiche_prefixe(a->gauche);
        affiche_prefixe(a->droite);
    }
}

void affiche_infixe(arbre *a)
{
    if (a != NULL)
    {
        affiche_infixe(a->gauche);
        printf("%d ", a->valeur);
        affiche_infixe(a->droite);
    }
}

void affiche_postfixe(arbre *a)
{
    if (a != NULL)
    {
        affiche_postfixe(a->gauche);
        affiche_postfixe(a->droite);
        printf("%d ", a->valeur);
    }
}
```

Sur l'exemple, on obtient respectivement `5 1 4 9 7 20`, `1 4 5 7 9 20` et `4 1 7 20 9 5`.

Comme les nœuds sont mutables, on peut faire des modifications en place.

Question 6

Écrire une fonction qui incrémente toutes les étiquettes dans un arbre.

Corrigé

```c
void incremente(arbre *a)
{
    if (a != NULL)
    {
        a->valeur = a->valeur + 1;
        incremente(a->gauche);
        incremente(a->droite);
    }
}
```

La fonction ne renvoie rien : elle modifie directement les nœuds accessibles depuis le pointeur `a`.

CHAPITRE 28

Dictionnaires et Ensembles finis

DANS CE CHAPITRE

28.1 Principe d'un dictionnaire
28.2 Implémentation immuable : ABR
28.3 Implémentation mutable : les tables de hachage

Note J'ai repris l'organisation de ce chapitre cette année.

Les arbres binaires de recherche, y compris les arbres auto-équilibrants (arbres 2-3, arbres rouges et noirs), sont présentés dans le chapitre sur les arbres ; ce chapitre se contente d'en déduire une implémentation des dictionnaires.

28.1 Principe d'un dictionnaire

28.1.1 Définition

Un dictionnaire est une structure de données abstraite permettant de représenter une association entre des clés et des valeurs.

Par exemple, on pourrait imaginer que les clés soient des chaînes de caractères et les valeurs des nombres.

Plus précisément, si le type des clés est `k` et celui des valeurs `v`, un dictionnaire est la donnée d'un type `t` (dépendant de `k` et de `v`) et des opérations :

- création d'un dictionnaire vide
- ajout d'une correspondance entre une clé `c` et une valeur `x`
- suppression d'une correspondance associée à une clé
- test de la présence d'une correspondance vis-à-vis d'une clé
- récupération de la valeur associée à une clé

On remarque qu'il est possible d'avoir une structure mutable ou non. Dans une structure mutable, le dictionnaire est modifié par les opérations alors que dans une structure immuable, les opérations renverront un nouveau dictionnaire.

Un point crucial des dictionnaires est d'avoir une relation d'égalité, ou même une relation d'ordre sur les clés.

28.1.2 Interfaces immuable et mutable

En OCaml, on peut écrire ces deux interfaces ainsi. On a choisi de fusionner le test de présence et la récupération de la valeur en une seule opération renvoyant une option : la clé est présente si et seulement si le résultat est de la forme `Some v`.

```
(* Interface immuable *)
type ('k, 'v) dico
val vide : ('k, 'v) dico
val ajout : ('k, 'v) dico -> 'k -> 'v -> ('k, 'v) dico
val suppression : ('k, 'v) dico -> 'k -> ('k, 'v) dico
val recherche : ('k, 'v) dico -> 'k -> 'v option
```

```
(* Interface mutable *)
type ('k, 'v) dico
val dico_creer : unit -> ('k, 'v) dico
val dico_ajout : ('k, 'v) dico -> 'k -> 'v -> unit
val dico_supprime : ('k, 'v) dico -> 'k -> unit
val dico_cherche : ('k, 'v) dico -> 'k -> 'v option
```

La spécification attendue est la suivante : une recherche de `c` renvoie `Some v` si la dernière opération portant sur `c` est un ajout de `(c, v)`, et `None` s'il n'y en a pas ou si c'est une suppression. En particulier, ajouter une clé déjà présente **remplace** la valeur associée, et supprimer une clé absente ne fait rien.

On a des interfaces analogues pour les ensembles finis : il suffit de retirer les valeurs.

28.1.3 Lien avec les ensembles finis

On retrouve beaucoup de points communs entre cette structure de données abstraite et celle d'ensembles finis. En effet, il suffit de considérer un ensemble fini de couples (clé, valeur) et de ne faire porter les comparaisons entre éléments de l'ensemble uniquement sur la première composante, la clé.

Ainsi, dans l'ensemble `{ ("riri", 12), ("fifi", 14) }` on pourra récupérer le couple de la forme `("fifi", *)` pour obtenir la valeur qui lui est associée.

Si on sait implémenter des ensembles finis, on peut en déduire une implémentation des dictionnaires.

Pour cela, il suffit de considérer une interface sur les ensembles finis permettant de définir sa propre fonction de comparaison, égalité ou ordre selon les cas. On va donc chercher à implémenter la structure, immuable ou mutable, suivante :

- création d'un ensemble vide
- ajout d'un élément
- suppression d'un élément
- test et récupération d'un élément

La récupération de l'élément est cruciale dans le cas des dictionnaires. En effet, dans l'exemple précédent, on obtiendra le couple `("fifi", 14)` en testant la valeur `"fifi"` grâce à la fonction de comparaison ne portant que sur la première composante.



Remarque 28.1 Réciproquement, **si on sait implémenter des dictionnaires, on en déduit une implémentation des ensembles finis** : un ensemble d'éléments de type `'a` est un dictionnaire de type `('a, unit) dico`, l'élément étant la clé et la valeur associée étant toujours `()`.

28.2 Implémentation immuable : ABR

Dans cette implémentation, on utilise les arbres binaires de recherche (ABR) vus dans le chapitre sur les arbres et dont on sait qu'ils peuvent représenter des ensembles finis de valeurs **pour lesquelles on a un ordre total**.

Pour implémenter un dictionnaire, on considère un ensemble C totalement ordonné de *clés* et un ensemble V de valeurs. On va représenter un dictionnaire par un ABR étiqueté par $C \times V$. Pour effectuer des opérations sur l'arbre, on se contente de regarder la première composante. Lors d'une recherche, on peut renvoyer la valeur.

On a alors ajout, suppression et recherche en $O(h)$ où h est la hauteur de l'arbre, soit $O(\log n)$ où n est le nombre d'entrées **sous réserve de bon équilibrage des ABR** (sinon h peut valoir $n-1$) : c'est le cas par exemple avec les arbres rouges et noirs.

L'implémentation se déduit alors très naturellement de celle des arbres binaires de recherche :

```
type ('a, 'b) dico = Nil
  | Noeud of ('a, 'b) dico * ('a * 'b) * ('a, 'b) dico

let rec recherche dico cle =
  match dico with
  | Nil -> None
  | Noeud(_, (y,v), _) when cle = y -> Some v
  | Noeud(g, (y,_), _) when cle < y -> recherche g cle
  | Noeud(_, _, d) -> recherche d cle
```

Pour l'ajout, on procède comme pour l'insertion dans un ABR, à une différence près : si la clé est déjà présente, on remplace la valeur associée au lieu d'ajouter un nouveau nœud, conformément à la spécification des dictionnaires.

```
let vide = Nil

let rec ajout dico cle v =
  match dico with
  | Nil -> Noeud(Nil, (cle, v), Nil)
  | Noeud(g, (y, w), d) ->
    if cle = y then Noeud(g, (cle, v), d)
    else if cle < y then Noeud(ajout g cle v, (y, w), d)
    else Noeud(g, (y, w), ajout d cle v)
```

La suppression est exactement la suppression dans un ABR (remplacement d'un nœud ayant deux enfants par la plus petite entrée de son sous-arbre droit), en ne comparant que les clés. Comme pour l'insertion, ces opérations reconstruisent uniquement les $O(h)$ nœuds du chemin de la racine à la clé et partagent tous les autres sous-arbres avec l'arbre initial : l'ancienne version du dictionnaire reste utilisable, c'est une structure immuable.

Enfin, un parcours infixe d'un ABR énumère ses entrées par ordre croissant des clés (voir l'exercice du chapitre sur les arbres). On obtient donc en $O(n)$ la liste triée des associations, ce qu'une table de hachage ne permet pas.

```
let vers_liste dico =
  let rec aux dico acc =
    match dico with
    | Nil -> acc
    | Noeud(g, e, d) -> aux g (e :: aux d acc)
  in
  aux dico []
```

Pour garantir une hauteur en $O(\log n)$, on peut utiliser les arbres 2-3 ou les arbres rouges et noirs : il suffit d'y remplacer les étiquettes par des couples (clé, valeur) comparés selon leur première composante.

28.3 Implémentation mutable : les tables de hachage

28.3.1 Hachage

Le principe du hachage (*hashing* en anglais) est, étant donné un ensemble de valeurs E et un entier n, de réaliser une **bonne** fonction

$$h : E \to [\![0, n-1]\!]$$

Que veut-on dire par **bonne** fonction ? Naturellement, avec E infini ou de cardinal très grand devant n, il est illusoire d'imaginer pouvoir réaliser une fonction injective. Cependant, on peut faire en sorte que deux valeurs proches aient des valeurs associées, on parle de *hachés*, différents. Cela représenterait ainsi une forme d'injectivité locale.

Quand $x \neq y$ et $h(x) = h(y)$, on dit que la paire $\{x, y\}$ est une *collision* pour h.

Exemple 28.2

Si $E \subset \mathbb{N}$, on peut considérer le reste modulo n comme fonction de hachage. Des éléments consécutifs auront ainsi des hachés distincts.

Toutes les paires $\{kn + r, ln + r\}$, où $k \neq l$ et $r \in [\![0, n-1]\!]$, sont des collisions.

Ainsi, cette fonction est très prévisible.

Remarque 28.3 Dans un contexte cryptographique, par exemple pour stocker un mot de passe, on souhaite éviter de remonter du haché à la valeur. En effet, en cas de compromission des accès, il suffirait d'*inverser* h pour retrouver le mot de passe de l'utilisateur.

La fonction de l'exemple précédent est ainsi peu sécurisée.

Il existe des fonctions de hachage cryptographiques comme `SHA-256` (`SHA-1` est aujourd'hui considérée comme cassée) qui sont sécurisées mais peuvent être mal utilisées : en hachant tous les mots du dictionnaire, on peut constituer sa propre table de correspondance entre hachés et mots de passe (les *rainbow tables* en sont une version compacte). Une manière simple de s'en prémunir est de ne pas stocker les hachés des mots de passe mais ceux de la concaténation d'un mot de passe et d'une donnée générée par le serveur pour chaque utilisateur : le sel.

28.3.2 Tables de hachage pour un ensemble fini

L'idée première des tables de hachage pour stocker des éléments de E est d'utiliser un tableau de n cases et une fonction de hachage $h : E \to [\![0, n-1]\!]$. Ainsi, pour stocker $x \in E$, on le place dans le tableau à l'adresse $h(x)$.

Naturellement, cette idée est très naïve en raison des collisions éventuelles. Tout l'intérêt des tables de hachage est donc de gérer les collisions. Il y a plusieurs stratégies plus ou moins efficaces en machine.

Pour chacune de ces stratégies, il sera donc nécessaire de faire plus de travail qu'un simple accès dans une case. En général, on a des stratégies qui sont linéaires dans le nombre d'éléments en collision. En pire cas, ce nombre est celui du nombre d'éléments. Mais, en pratique, si les collisions sont peu probables, trois éléments entrant en collision le seront encore moins, ce qui fait qu'on peut supposer que le nombre de collisions est faible devant le nombre d'éléments. On pourra donc faire l'hypothèse que les accès sont en $O(1)$.

✍ Plus précisément, sous l'hypothèse (idéale) que la fonction de hachage répartit les éléments uniformément et indépendamment dans les n cases, le nombre moyen d'éléments en collision avec un élément donné, pour m éléments stockés, est de l'ordre du *facteur de charge* $\alpha = m/n$. Le coût moyen d'un accès est alors en $O(1 + \alpha)$, donc en $O(1)$ tant que $m = O(n)$, ce que l'on garantit en agrandissant la table lorsqu'elle devient trop chargée.

On va montrer ici l'implémentation de différentes stratégies en utilisant un même cadre : on cherche à représenter un ensemble fini d'éléments de type `'a` où on a une fonction de hachage `hache : 'a -> int` et un test de comparaison `eq : 'a -> 'a -> bool` testant l'égalité.

On considère aussi définie une constante `n` et l'assurance que `hache x` soit dans $[\![0, n-1]\!]$.

28.3.3 Listes de collisions

La stratégie la plus simple de gestion des collisions est de ne pas stocker des éléments dans chaque case mais une liste chaînée d'éléments.

Cette stratégie a le désavantage de fragmenter la mémoire car il faut descendre le long des maillons.

```
type 'a htbl = 'a list array

let htbl_creer () = Array.make n []

let htbl_cherche t x =
```

```
  let found = ref None in
  List.iter (fun y -> if eq x y then found := Some y) t.(hache x);
  !found

let htbl_ajout t x =
  let h = hache x in
  if not (List.exists (eq x) t.(h)) then
    t.(h) <- x :: t.(h)

let htbl_supprime t x =
  let h = hache x in
  t.(h) <- List.filter (fun y -> not (eq x y)) t.(h)
```

28.3.4 Sondage linéaire

Pour éviter les listes chaînées, on peut stocker les éléments directement dans le tableau. Chaque case est alors *libre* ou *occupée* par un élément. Pour ajouter `x`, on calcule son haché i, c'est-à-dire `hache x`. Si la case i est libre, on y range `x`. Sinon, on teste successivement les cases $i+1 \bmod n$, $i+2 \bmod n$, ... jusqu'à trouver une case libre, dans laquelle on range `x`. On parle de *sondage linéaire* (ou de recherche linéaire), et plus généralement d'*adressage ouvert* (*open addressing*) puisque l'élément n'est pas forcément rangé à l'adresse donnée par son haché.

Pour rechercher `x`, on sonde de même les cases à partir de `hache x` : si on trouve `x`, il est présent, et si on tombe sur une case libre, il est absent, car il aurait sinon été rangé au plus tard dans cette case.

Exemple 28.4

Avec $n = 4$ et `hache x = x mod 4`, on ajoute 1492, 1515 puis 1939 à une table vide. On a `hache 1492 = 0` et 1492 est rangé en case 0. On a `hache 1515 = 3` et 1515 est rangé en case 3. Enfin `hache 1939 = 3`, la case 3 est occupée, la case 0 aussi en revenant au début du tableau, et 1939 est rangé en case 1.

case	0	1	2	3
contenu	1492	1939	libre	1515

La suppression est plus délicate : on ne peut pas simplement libérer la case. Dans l'exemple précédent, si on supprime 1515 en libérant la case 3, une recherche de 1939 commence en case 3, la trouve libre, et conclut à tort que 1939 est absent.

On marque donc la case d'un élément supprimé comme une *pierre tombale* : lors d'une recherche, on passe par-dessus comme pour une case occupée, mais lors d'un ajout, on peut la réutiliser comme une case libre.

```
type 'a case = Vide | PierreTombale | Element of 'a

type 'a htbl_lin = 'a case array

let htbl_lin_creer () = Array.make n Vide

(* Renvoie Some (i, y) si t.(i) contient y égal à x, None sinon *)
let htbl_lin_trouve t x =
  let h = hache x in
  let rec aux k =
    if k = n then None
    else
      let i = (h + k) mod n in
      match t.(i) with
      | Vide -> None
      | Element y when eq x y -> Some (i, y)
      | _ -> aux (k + 1)
  in
  aux 0

let htbl_lin_cherche t x =
  match htbl_lin_trouve t x with
```

```
  | None -> None
  | Some (_, y) -> Some y

let htbl_lin_supprime t x =
  match htbl_lin_trouve t x with
  | None -> ()
  | Some (i, _) -> t.(i) <- PierreTombale

let htbl_lin_ajout t x =
  if htbl_lin_trouve t x = None then begin
    let h = hache x in
    let rec aux k =
      if k = n then failwith "Erreur : table pleine"
      else
        let i = (h + k) mod n in
        match t.(i) with
        | Element _ -> aux (k + 1)
        | Vide | PierreTombale -> t.(i) <- Element x
    in
    aux 0
  end
```

La correction repose sur l'invariant suivant : si un élément `x` est rangé dans la case $(\texttt{hache x} + k) \bmod n$, aucune des cases $(\texttt{hache x} + j) \bmod n$ pour $j < k$ n'est `Vide`. L'ajout le préserve, car il range `x` dans la première case non occupée par un élément, et la suppression aussi, car elle ne crée jamais de case `Vide`. La recherche de `x` ne peut donc pas s'arrêter avant d'avoir atteint sa case.

Remarque 28.5

- Avec cette stratégie, on stocke au plus n éléments : le facteur de charge α est toujours inférieur à 1. Les éléments ont tendance à former des blocs de cases consécutives occupées, ce qui allonge les recherches quand α se rapproche de 1. En pratique, on maintient α assez petit, par exemple $\alpha \leq 2/3$, et le coût moyen d'un accès est alors en $O(1)$ (admis).
- Les pierres tombales ne disparaissent jamais et ralentissent les recherches. On s'en débarrasse en reconstruisant la table, ce qu'on fait de toute manière lors d'un redimensionnement.
- L'intérêt de cette stratégie est que les cases parcourues sont contiguës en mémoire, ce qui est bien plus efficace en machine que de suivre des maillons dispersés.

28.3.5 Redimensionnement

On a supposé jusqu'ici que la taille n de la table était fixée. Si le nombre m d'éléments stockés devient grand devant n, le facteur de charge $\alpha = m/n$ augmente et les accès ne sont plus en temps constant. Comme on ne connaît pas en général m à l'avance, on fait grossir la table au fur et à mesure, comme pour un tableau dynamique : lorsque α dépasse un seuil, on alloue une table deux fois plus grande et on y réinsère tous les éléments.

Il faut alors que la fonction de hachage dépende de la taille de la table. On se donne pour cela une fonction renvoyant un entier positif quelconque, comme `Hashtbl.hash` en OCaml, et on prend son reste modulo la taille courante. Les éléments changent donc de case lors d'un redimensionnement : on ne peut pas recopier le tableau tel quel, il faut **réinsérer** chaque élément.

Propriété 28.6

Si on double la taille de la table dès que le nombre d'éléments atteint la taille avant un ajout, une suite de m opérations partant d'une table de taille n_0 effectue des redimensionnements de coût total $O(n_0 + m)$. Le coût amorti d'un redimensionnement par opération est donc en $O(1)$, et on a toujours $\alpha \leq 1$.

Preuve Un redimensionnement depuis une table de taille s coûte $O(s)$: il faut allouer la nouvelle table et réinsérer les s éléments présents. Les tailles successives sont $n_0, 2n_0, 4n_0, \ldots, 2^p n_0$. Le dernier redimensionnement, depuis la taille $2^{p-1}n_0$, a lieu quand la table contient $2^{p-1}n_0$ éléments, donc après au moins autant d'ajouts, d'où $2^{p-1}n_0 \leq m$. Le coût total est alors en

$$O\left(\sum_{i=0}^{p-1} 2^i n_0\right) = O(2^p n_0) = O(n_0 + m)$$

Enfin, avant un ajout, soit le nombre d'éléments est strictement inférieur à la taille, soit il lui est égal et on double la taille : dans les deux cas, après l'ajout, le nombre d'éléments est au plus égal à la taille.

□

28.3.6 Implémentation d'un dictionnaire

Comme expliqué au début de ce chapitre, on obtient un dictionnaire à partir d'une table de hachage pour les ensembles finis en stockant des couples `(cle, valeur)`, la fonction de hachage et le test d'égalité ne portant que sur la clé. Avec des listes de collisions, chercher la valeur associée à une clé dans une liste de couples, c'est exactement ce que fait `List.assoc_opt`, et `List.remove_assoc` supprime le couple associé à une clé.

On en déduit une implémentation de l'interface mutable avec redimensionnement. Ici, `hache_cle` renvoie un entier positif quelconque, par exemple `let hache_cle = Hashtbl.hash`, et on compare les clés avec `=` comme pour les ABR.

```
type ('k, 'v) dico = {
  mutable donnees : ('k * 'v) list array;
  mutable nb : int; (* nombre d'associations *)
}

let indice t c = hache_cle c mod Array.length t.donnees

let dico_creer () = { donnees = Array.make 8 []; nb = 0 }

let dico_cherche t c = List.assoc_opt c t.donnees.(indice t c)

let redimensionne t =
  let anciennes = t.donnees in
  t.donnees <- Array.make (2 * Array.length anciennes) [];
  Array.iter
    (List.iter (fun (c, v) ->
         let i = indice t c in
         t.donnees.(i) <- (c, v) :: t.donnees.(i)))
    anciennes

let dico_supprime t c =
  let i = indice t c in
  if List.mem_assoc c t.donnees.(i) then begin
    t.donnees.(i) <- List.remove_assoc c t.donnees.(i);
    t.nb <- t.nb - 1
  end

let dico_ajout t c v =
  dico_supprime t c;
  if t.nb = Array.length t.donnees then redimensionne t;
  let i = indice t c in
  t.donnees.(i) <- (c, v) :: t.donnees.(i);
  t.nb <- t.nb + 1
```

On a supprimé l'ancienne association avant l'ajout pour qu'une clé n'apparaisse qu'une fois, ce qui garantit que `nb` est bien le nombre de clés et qu'ajouter une clé présente remplace sa valeur. D'après la propriété précédente, on a toujours $\alpha \leq 1$ et, sous l'hypothèse d'un hachage uniforme, chaque opération est en $O(1)$ en moyenne et en amorti.

Remarque 28.7 Le module `Hashtbl` d'OCaml fournit une telle table de hachage mutable, avec redimensionnement automatique : `Hashtbl.create`, `Hashtbl.find_opt`, `Hashtbl.remove` et `Hashtbl.replace`. **Attention**, `Hashtbl.add` n'a pas la même spécification que `dico_ajout` : si la clé est déjà présente, il ajoute une nouvelle association qui masque l'ancienne sans la supprimer, et celle-ci réapparaît après un `Hashtbl.remove`. C'est `Hashtbl.replace` qui remplace la valeur.

29

TRAVAUX PRATIQUES 29

Arbres rouges et noirs en C

29.1 Arbres rouges et noirs

Ce TP fait suite à la partie « Arbres binaires en C » des TP sur les arbres, dont on reprend le type et la fonction `cree_feuille`, et met en œuvre les arbres rouges et noirs du cours sur les arbres.

✍ **Rappels** On utilise ici les arbres rouges et noirs obtenus en codant les arbres 2-3 comme dans le cours : un nœud ternaire (a, b) devient un nœud noir a dont le fils **droit** b est rouge. Ainsi :

- la racine est noire, et l'arbre vide est considéré comme noir ;
- un nœud rouge n'a jamais de fils rouge ;
- le fils gauche d'un nœud n'est jamais rouge (les liens rouges penchent à droite) ;
- tous les chemins de la racine à l'arbre vide contiennent le même nombre de nœuds noirs.

Les trois opérations de rééquilibrage sont :

- la **rotation droite** d'un nœud x de fils gauche y : y prend la place et la couleur de x, et x, devenu fils droit de y, est colorié en rouge ; le fils droit de y devient le fils gauche de x ;
- la **rotation gauche**, symétrique ;
- la **bascule** d'un nœud dont les deux fils sont rouges : les fils deviennent noirs et le nœud devient rouge, ce qui fait remonter le défaut vers la racine (c'est l'éclatement d'un nœud quaternaire de l'arbre 2-3).

Ces opérations préservent la propriété d'arbre binaire de recherche et le nombre de nœuds noirs sur chaque chemin.

On va rajouter un champ au type précédent pour indiquer la couleur d'un nœud :

```
struct noeud {
    struct noeud *gauche;
    struct noeud *droite;
    bool rouge;
    int valeur;
};
typedef struct noeud *arbre;
```

On vous conseille ici de copier le fichier précédent et de faire un nouveau programme.

Question 1

Reprendre `cree_feuille` pour que les feuilles soient rouges par défaut.

Corrigé

```
arbre cree_noeud(int x)
{
    arbre n = malloc(sizeof(struct noeud));
```

```
    n->valeur = x;
    n->rouge = true;
    n->gauche = n->droite = NULL;
    return n;
}
```

Un nouveau nœud est rouge : l'insérer ne change pas le nombre de nœuds noirs sur les chemins.

Question 2

Écrire des fonctions `arbre rotation_droite(arbre a)`, `arbre rotation_gauche(arbre a)` et `arbre bascule(arbre a)` qui effectuent une des trois opérations en renvoyant la nouvelle racine.

Corrigé

```
arbre rotation_droite(arbre x)
{
    arbre y = x->gauche;
    x->gauche = y->droite;
    y->droite = x;
    y->rouge = x->rouge;
    x->rouge = true;
    return y;
}

arbre rotation_gauche(arbre y)
{
    arbre x = y->droite;
    y->droite = x->gauche;
    x->gauche = y;
    x->rouge = y->rouge;
    y->rouge = true;
    return x;
}

arbre bascule(arbre a)
{
    a->rouge = true;
    a->gauche->rouge = false;
    a->droite->rouge = false;
    return a;
}
```

Question 3

Écrire des fonctions utilitaires `bool rouge(arbre a)` et `bool noir(arbre a)` qui testent la couleur de la racine. On fera attention au fait que l'arbre vide est noir.

Corrigé

```
bool rouge(arbre a)
{
    if (a == NULL) return false;
    return a->rouge;
}

bool noir(arbre a)
{
    return !rouge(a);
}
```

Question 4

Écrire des fonctions `arbre essai_rotgauche(arbre a)`, `essai_rotdroite` et `essai_bascule` qui n'effectuent ces opérations que dans les cas nécessaires pour l'insertion.

Corrigé

```
arbre essai_bascule(arbre a)
{
    if(rouge(a->gauche) && rouge(a->droite))
        return bascule(a);
    return a;
}

arbre essai_rotgauche(arbre a)
{
    if(rouge(a->droite) && rouge(a->droite->droite))
        return rotation_gauche(a);
    return a;
}

arbre essai_rotdroite(arbre a)
{
    if(rouge(a->gauche) && noir(a->droite))
        return rotation_droite(a);
    return a;
}
```

- `essai_rotdroite` fait pencher à droite un fils gauche rouge (si le fils droit est aussi rouge, c'est la bascule qui s'en chargera) ;
- `essai_rotgauche` traite deux rouges consécutifs à droite, qui deviennent les deux fils rouges d'un même nœud ;
- `essai_bascule` éclate alors ce nœud quaternaire.

L'évaluation paresseuse de `&&` garantit qu'on n'accède à `a->droite->droite` que si `a->droite` est rouge, donc non vide.

Question 5

En déduire une fonction `arbre insere(arbre a, int x)` qui effectue l'insertion dans un arbre rouge et noir. On pourra définir une fonction auxiliaire `arbre insere_aux(arbre a, int x)` qui effectue l'insertion et qu'on appellera dans `insere` avant de faire le noircissement final de la racine.

Corrigé

```
arbre insere_aux(arbre a, int x)
{
    if (a == NULL)
        return cree_noeud(x);
    if (a->valeur > x)
        a->gauche = insere_aux(a->gauche, x);
    else
        a->droite = insere_aux(a->droite, x);
    return essai_bascule(essai_rotgauche(essai_rotdroite(a)));
}

arbre insere(arbre a, int x)
{
    arbre a2 = insere_aux(a, x);
    a2->rouge = false;
    return a2;
}
```

Les opérations de rééquilibrage sont effectuées en remontant, sur chaque nœud du chemin d'insertion : l'insertion est donc en $O(h)$ où h est la hauteur de l'arbre, c'est-à-dire en $O(\log n)$.

Pour tester, on peut écrire une fonction qui vérifie récursivement les propriétés rappelées au début (ABR, pas de fils gauche rouge, pas de rouge fils d'un rouge) et renvoie le nombre de nœuds noirs sur les chemins, en vérifiant qu'il est le même à gauche et à droite. Par exemple, après l'insertion de 1, 2, …, 2000, l'arbre obtenu vérifie ces propriétés et a une hauteur de 15, contre 1999 pour un ABR sans rééquilibrage.

CHAPITRE 30

Graphes

DANS CE CHAPITRE

30.1 Graphes orientés
30.2 Graphes non orientés
30.3 Graphes classiques
30.4 Parcours
30.5 Tri topologique
30.6 Plus courts chemins

Note Une grande partie est issue verbatim de mon ancien poly. À affiner.

30.1 Graphes orientés

30.1.1 Définition

Définition 30.1

Un graphe orienté est un couple $G = (S, A)$ où S est un ensemble fini et A est une relation **irréflexive** sur S, c'est-à-dire $A \subset \{(x, y) \mid x, y \in S, x \neq y\}$.

Les éléments de S sont appelés les **sommets** du graphe G et les éléments de A les **arêtes**. Si $(x, y) \in A$, on dit que x est la source de l'arête et y est son but. Quand le contexte n'est pas ambigu, on notera $x \to y$.

Exemple 30.2

- $G = (\{a, b, c\}, \{(a, b), (b, c), (c, b)\})$
- $D_n = (\llbracket 1, n \rrbracket, \{(a, b) \mid a \text{ divise } b, a \neq b\})$

Remarque 30.3 Ici, on a exclu les graphes infinis ainsi que les graphes avec des boucles ou des arêtes parallèles. On est dans le contexte des **graphes finis simples**.

On représente graphiquement un graphe sous la forme d'un diagramme sagittaire où les sommets sont des points et où une arête (a, b) est une flèche allant du point a au point b.

Exemple 30.4

- G est représenté par

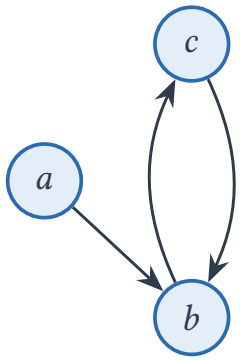


Figure 30.1 — Diagramme sagittaire du graphe G : arêtes $a \to b$, $b \to c$ et $c \to b$.

```
* $D_{10}$ est représenté par
```

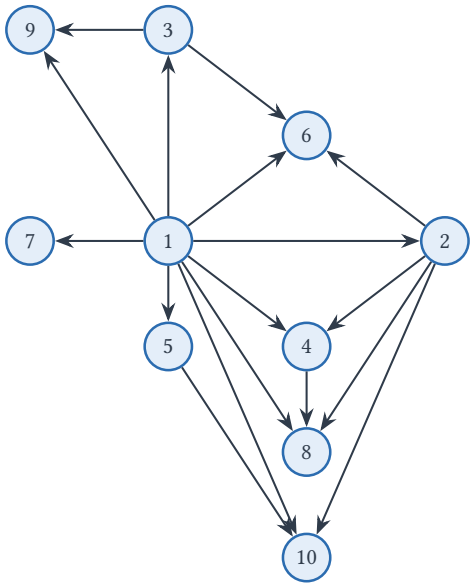


Figure 30.2 — Graphe D_{10} : une flèche $x \to y$ quand x divise strictement y.

30.1.2 Voisins et degrés

Définition 30.5

Soit $G = (S, A)$ un graphe orienté et $x \in S$.

On appelle **voisins sortants**, ou **successeurs**, de x les éléments de

$$v_+(x) = \{y \in S \mid (x, y) \in A\}$$

et on appelle **degré sortant** de x le cardinal de cet ensemble $d_+(x) = \mid v_+(x) \mid$.

De même, on parle de **voisins entrants**, ou **prédécesseurs** pour les éléments de

$$v_-(x) = \{z \in S \mid (z, x) \in A\}$$

et on parle de **degré entrant** pour $d_-(x) = \mid v_-(x) \mid$.

Théorème 30.6

Soit $G = (S, A)$ un graphe, on a $\sum_{x \in S} d_+(x) = \sum_{x \in S} d_-(x) = \mid A \mid$

Preuve On peut partitionner A en regroupant les arêtes de même source, on a ainsi :

$$A = \bigcup_{x\in S}\{(x,y) \mid y \in S, (x,y) \in A\} = \bigcup_{x\in S}\{(x,y) \mid y \in v_+(x)\}$$

Cette réunion étant disjointe, en prenant le cardinal de cette égalité, on obtient directement $|A| = \sum_{x\in S} d_+(x)$. L'autre égalité est symétrique en considérant les arêtes de même but.

□

30.1.3 Chemin

Définition 30.7

Soit $G = (S, A)$ un graphe orienté et $x, y \in S$.

Une suite finie $\varphi = (s_0, s_1, ..., s_p)$ où $p \in \mathbb{N}, s_0 = x, s_p = y$ et

$$\forall i \in [\![1, p]\!], (s_{i-1}, s_i) \in A$$

est appelée un **chemin**, de longueur p, de x vers y. On a donc

$$s_0 \to s_1 \to \ldots \to s_p$$

On notera $\varphi : x \rightsquigarrow y$ pour signifier que φ est un tel chemin et $x \rightsquigarrow y$ pour signifier qu'il existe un chemin de x vers y. On dit alors que y est accessible depuis x.

Théorème 30.8

$\rightsquigarrow$ est la plus petite relation sur S contenant $\to$ et qui soit réflexive et transitive.

Preuve Montrons d'abord que $\rightsquigarrow$ convient. Le chemin (x) de longueur 0 donne $x \rightsquigarrow x$, donc $\rightsquigarrow$ est réflexive. Si $x \to y$, le chemin (x, y) donne $x \rightsquigarrow y$, donc $\to \subset \rightsquigarrow$. Enfin, si $\varphi : x \rightsquigarrow y$ et $\psi : y \rightsquigarrow z$, la concaténation de φ et ψ (en ne répétant pas y) est un chemin de x à z, donc $\rightsquigarrow$ est transitive.

Réciproquement, on considère $\to \subset R \subset S^2$ telle que R soit réflexive et transitive.

Soient $x, y \in S$ et $\varphi = (s_0, ..., s_p) : x \rightsquigarrow y$.

Si $p = 0$, $x = y$ et xRy par réflexivité. Sinon, comme $s_{i-1} \to s_i$ on a $s_{i-1}Rs_i$ pour tout $i \in [\![1, p]\!]$ et, par transitivité, $x = s_0Rs_p = y$. Donc $x \rightsquigarrow y \Rightarrow xRy$.

Ainsi $\rightsquigarrow \subset R$.

□

On dit que $\rightsquigarrow$ est la fermeture réflexive et transitive de $\to$.

Remarque 30.9 Vocabulaire additionnel :

- Si tous les sommets sont distincts, sauf éventuellement le premier et le dernier, on dit que le chemin est **élémentaire**.
- Si toutes les arêtes sont distinctes, on dit que le chemin est **simple**.
- Si $x = y$, on dit que le chemin est **fermé**.
- Un chemin simple dont tous les sommets sont distincts est appelé une **chaîne**.
- Un chemin élémentaire fermé simple de longueur au moins 1 est appelé un **cycle**. *Comme le chemin vide issu de x est fermé, il est nécessaire de considérer des chemins non vides.*
- Un graphe contenant au moins un cycle est dit **cyclique**. Dans le cas contraire, on dit qu'il est **acyclique**.

Remarque 30.10 Si $\varphi : x \rightsquigarrow y$ et $\psi : y \rightsquigarrow z$ on note $\varphi\psi : x \rightsquigarrow z$ la concaténée des deux chemins.

Définition 30.11

On note $\leftrightarrow$ la plus grande relation d'équivalence incluse dans $\rightsquigarrow$.

Plus précisément, on a

$$x \leftrightarrow y \Leftrightarrow (x \rightsquigarrow y \wedge y \rightsquigarrow x)$$

Les classes d'équivalence pour $\leftrightarrow$ sont appelées les composantes **fortement connexes** du graphe. S'il n'y a qu'une classe, on dit que le graphe est **fortement connexe**.

30.1.4 Sous-graphe

Définition 30.12

Soit $G = (S, A)$ et $X \subset S$. On appelle **sous-graphe** induit par X le graphe (X, A_X) où

$$A_X = \{(a, b) \in A \mid a \in X \wedge b \in X\}$$

Un graphe G' est un **sous-graphe** de G quand c'est le sous-graphe de G induit par les sommets de G' (ils sont alors nécessairement des sommets de G).

Remarque 30.13 Les composantes fortement connexes sont les sous-graphes fortement connexes maximaux pour l'inclusion.

30.1.5 Implémentation

Énumération de sommets

Si on considère un graphe $G = (S, A)$, il est assez naturel de représenter ses sommets dans un tableau. Pour cela, on fixe naturellement un ordre sur ces sommets et on va associer à chaque sommet son indice dans le tableau.

Par exemple, si $S = \{a, b, c, d\}$ on va pouvoir considérer `[ a, b, c, d ]` et ainsi associer à `a` son indice 0 dans le tableau. L'ordre est arbitraire, on aurait pu considérer `[ b, c, a, d ]` et l'indice de `a` aurait alors été 2.

Ce qui compte, c'est de pouvoir travailler directement sur les indices et pas sur les éléments. Une manière de s'en convaincre est d'imaginer le graphe d'un réseau social où un sommet correspond au profil d'une personne, et contient donc beaucoup (*trop*) d'informations. Il est bien plus raisonnable de lui associer un identifiant unique et d'utiliser cet identifiant ensuite.

Quand on va implémenter des graphes, on peut donc supposer que les sommets sont les entiers de `0` à `n-1` où $|S| = n$. Il sera toujours possible de retrouver la correspondance avec les sommets eux-mêmes.

Remarque 30.14 Si on est bloqué par le coût de l'opération permettant d'obtenir l'indice d'un sommet, qui est en $O(|S|)$ par une recherche linéaire, on peut très bien définir un dictionnaire associant à chaque sommet son indice en $O(1)$.

Il se trouve qu'on ne va pas forcément s'intéresser à ces questions mais plus aux raisonnements sur la structure du graphe.

Matrice d'adjacence

Définition 30.15

Soit $G = (S, A)$ un graphe et $S = \{s_1, \ldots, s_n\}$ une énumération sans répétition des sommets de G. La matrice $M_G = (m_{ij})_{1 \leq i,j \leq n} \in \mathcal{M}_n(\mathbb{R})$ définie par

$$\forall i, j \in [\![1, n]\!], m_{ij} = \begin{cases} 1 & \text{si } s_i \to s_j \\ 0 & \text{sinon} \end{cases}$$

est appelée une **matrice d'adjacence** de G.

Remarque 30.16 Il s'agit bien d'**une** matrice d'adjacence et pas de **la** matrice car elle dépend de l'énumération des sommets.

Exemple 30.17

Si on considère le graphe

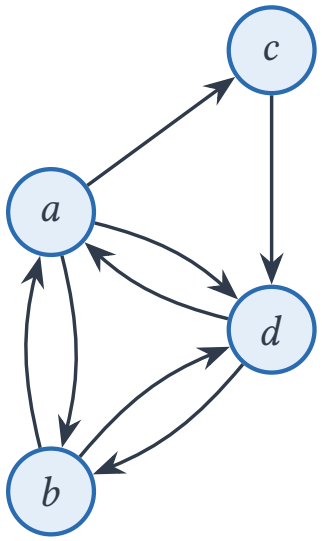


Figure 30.3 — Graphe orienté à quatre sommets a, b, c, d.

```
On aura pour l'énumération $a,b,c,d$ la matrice :

$$
\begin{pmatrix}
0 & 1 & 1 & 1 \\
1 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 \\
1 & 1 & 0 & 0
\end{pmatrix}
$$

En fait, la ième ligne correspond au ième sommet de l'énumération. Ici, la
dernière ligne $1 1 0 0$ correspond au sommet d et donne dans l'ordre $a,b,c,d$
la présence ou non d'une arête $d \rightarrow *$.

Si on considère l'énumération $b,a,d,c$ on aura la matrice :

$$
\begin{pmatrix}
0 & 1 & 1 & 0 \\
1 & 0 & 1 & 1 \\
1 & 1 & 0 & 0 \\
0 & 0 & 1 & 0
\end{pmatrix}
$$
```

On en déduit ainsi une implémentation directe d'un graphe en représentant la matrice d'adjacence par un tableau de tableaux.

Python

```
m = [ [ 0, 1, 1, 1 ],
      [ 1, 0, 0, 1 ],
      [ 0, 0, 0, 1 ],
      [ 1, 1, 0, 0 ] ]
```

OCaml

```
let m = [| [| 0; 1; 1; 1 |];
           [| 1; 0; 0; 1 |];
           [| 0; 0; 0; 1 |];
           [| 1; 1; 0; 0 |] |]
```

C

```
int m[4][4] = { { 0, 1, 1, 1 },
                { 1, 0, 0, 1 },
                { 0, 0, 0, 1 },
                { 1, 1, 0, 0 } };
```

Remarque 30.18 Il y a le même lien entre les différentes matrices d'adjacence d'un graphe donné et la matrice d'un endomorphisme dans les différentes permutations d'une même base, à chaque fois on obtient la nouvelle matrice en permutant de même ses lignes et ses colonnes.

Cette représentation permet d'accéder, en lecture comme en écriture, à une arête en temps constant. Cependant, pour récupérer les voisins d'un sommet, il est nécessaire de parcourir toute la ligne correspondante, donc en $O(|\,S\,|)$.

Remarque 30.19 Dans la majorité des cas, $|\,A\,|$ est petit par rapport à $|\,S\,|^2$. Ainsi, la matrice contient très peu de 1 et beaucoup de 0, on dit qu'elle est *creuse*. Une telle matrice *creuse* peut être efficacement représentée par l'ensemble fini des arêtes (i, j). Par exemple, avec une table de hachage. Cependant, dans le cas des graphes, il est possible d'avoir une structure creuse plus efficace.

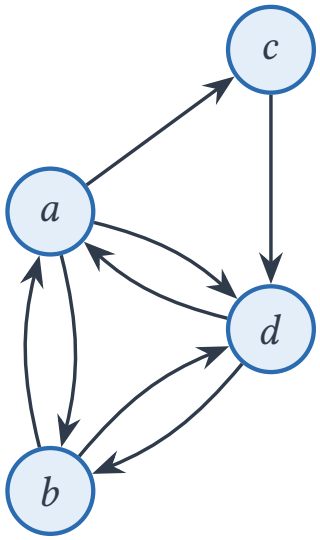
À titre d'exemple, un graphe issu d'un réseau social contenant 300 000 arêtes pour 8 000 sommets a un rapport $|\,A\,| / |\,S\,|^2$ est de 4 pour 1000 !

Listes d'adjacences

La donnée de $v_+(x)$ pour chaque sommet $x \in S$ suffit à reconstruire A.

Exemple 30.20

Si on reprend le graphe précédent,




```
on a $S = \{ a,b,c,d\}$
et $A = \{ (a, b), (a, c), (a,d), (b,a), (b, d) , (c,d), (d, a), (d, b) \}$.

Or $v_+(a) = \{ b, c, d \}, v_+(b) = \{ a, d \}, v_+(c)
= \{ d \}$ et $v_+(d) = \{ a, b \}$.
```

```
Si on pose $V_+(x) = \enscomp{ (x,y) }{ y \in v_+(x) }$ on a directement
$A = \bigcup_{x \in S} V_+(x)$.
```

On en déduit ainsi une représentation d'un graphe où on place dans un tableau chaque $v_+(x)$ représenté par une liste chaînée ou un tableau dynamique.

Remarque 30.21 Pour représenter $v_+(x)$ on a plusieurs choix : on peut utiliser directement les valeurs des sommets ou utiliser des indices dans une énumération. C'est le second choix qu'on fera en général car il est plus simple. Cependant, en cas de suppression d'un sommet, les indices changent, et il faut renuméroter dans les listes.

Pour le graphe précédent, on pourra donc considérer une énumération `[ a, b, c, d ]` et la représentation sous forme de liste d'adjacence pourra être :

Python

```python
l = [ [ 1, 2, 3 ], [ 0, 3 ], [ 3 ], [ 0, 1 ] ]
```

OCaml

```ocaml
let l = [|
    [ 1; 2; 3 ]; [ 0; 3 ]; [ 3 ]; [ 0; 1 ]
    |]
```

C

```c
struct graphe {
    int n;          // nombre de sommets
    int *degre;     // degre[x] = d_+(x)
    int **voisins; // voisins[x][0], ..., voisins[x][degre[x] - 1]
};
typedef struct graphe graphe;

int v0[] = { 1, 2, 3 };
int v1[] = { 0, 3 };
int v2[] = { 3 };
int v3[] = { 0, 1 };
int *voisins[] = { v0, v1, v2, v3 };
int degre[] = { 3, 2, 1, 2 };
graphe l = { 4, degre, voisins };
```

En C, on stocke chaque $v_+(x)$ dans un tableau de taille $d_+(x)$, ce qui impose de mémoriser aussi les degrés. Ici, les tableaux sont alloués statiquement ; pour un graphe construit à l'exécution, on les allouerait avec `malloc`.

L'accès en lecture ou en écriture à une arête $x \to y$ est alors en $O(|\ A\ |)$, plus précisément en $O(d_+(x))$, et on peut parcourir les voisins sortants de x en $O(d_+(x))$ également. Accéder à la liste elle-même se fait en $O(1)$.

Pour obtenir les voisins entrants, il est par contre nécessaire de tester la présence de x dans chacune des autres listes, on obtient donc un algorithme en $O(|\ S\ | + |\ A\ |)$: on parcourt chaque case du tableau des listes puis chaque maillon de listes d'adjacence.

Il est possible d'améliorer cela en utilisant une structure plus efficace pour stocker les ensembles. Cela peut être un dictionnaire reposant sur une table de hachage. L'avantage de cela est que pour tester l'appartenance $y \in v_+(x)$ on sera en $O(1)$ avec un dictionnaire alors qu'on sera en $O(d_+(x))$ avec une liste.

Comparaison

opération	Matrice	Listes	Dictionnaire
complexité spatiale	$O(\mid S \mid^2)$	$O(\mid S \mid + \mid A \mid)$	$O(\mid S \mid + \mid A \mid)$
arête test	$O(1)$	$O(\mid A \mid)$	$O(1)$
arête ajout	$O(1)$	$O(1)$	$O(1)$
arête suppression	$O(1)$	$O(\mid A \mid)$	$O(1)$
sommet ajout	$O(\mid S \mid^2)$	$O(\mid S \mid)$	$O(\mid S \mid)$
sommet suppression	$O(\mid S \mid^2)$	$O(\mid S \mid + \mid A \mid)$	$O(\mid S \mid)$
voisins/degré +	$O(\mid S \mid)$	$O(1)$	$O(1)$
voisins/degré -	$O(\mid S \mid)$	$O(\mid S \mid + \mid A \mid)$	$O(\mid S \mid)$

Notons qu'il est possible d'améliorer certaines complexités en utilisant des tableaux dynamiques, notamment les ajouts et suppressions de sommets.

Remarque 30.22 Il semble a priori clair qu'on devrait utiliser des dictionnaires pour manipuler des graphes en voyant ce tableau. Pourtant, on utilisera presque exclusivement des listes d'adjacence. Pourquoi ? En grande partie parce que les opérations rendues efficaces sont rares et ne justifient pas la difficulté accrue de manipulation. En effet, l'opération la plus importante est de pouvoir énumérer les voisins, et une liste chaînée le permet facilement et efficacement.

30.2 Graphes non orientés

30.2.1 Définition et adaptation du vocabulaire

Définition 30.23

Un graphe non orienté est un couple $G = (S, A)$ où S est un ensemble fini et A est un ensemble de paires d'éléments de S, c'est-à-dire d'ensembles à deux éléments $\{x, y\}$ où $x, y \in S$.

Les éléments de S sont appelés les **sommets** du graphe G et les éléments de A les **arêtes**. Si $\{x, y\} \in A$, on note $x \smile y$.

Remarque 30.24 Si $G = (S, A)$ est un graphe orienté, on peut en déduire deux graphes non orientés :

- un graphe par défaut $G^- = (S, A^-)$ où $\forall x, y \in S, x \smile y \Leftrightarrow (x \to y \wedge y \to x)$;
- un graphe par excès $G^+ = (S, A^+)$ où $\forall x, y \in S, x \smile y \Leftrightarrow (x \to y \vee y \to x)$.

On a également A symétrique $\Leftrightarrow G^+ = G^-$. Cela correspond à un graphe où chaque arête est dans les deux sens ($x \to y$ et $y \to x$) et, donc, on peut oublier l'orientation.

De la même manière, on peut associer à un graphe non orienté G son graphe orienté symétrique obtenu en doublant chaque arête. C'est-à-dire en posant $x \to y$ si $x \smile y$. On a donc $\forall x, y \in S, x \to y \Leftrightarrow y \to x$ et le graphe est bien symétrique.

On reprend directement l'essentiel du vocabulaire des graphes orientés symétriques avec des simplifications :

Définition 30.25

Soit $G = (S, A)$ un graphe et $x \in S$.

- On appelle *voisins* de x les éléments de
$$v(x) = \{y \in S \mid x \smile y\}$$
- On appelle *degré* de x l'entier $d(x) = \mid v(x) \mid$.

Théorème 30.26

$\sum_{x \in S} d(x) = 2 \mid A \mid$

✍ **Preuve** Il suffit d'appliquer le théorème analogue au graphe orienté symétrique associé : chaque sommet y a $d(x)$ voisins sortants et chaque arête $\{x, y\}$ donne exactement deux arêtes (x, y) et (y, x).

□

On étend directement la notion de **chemin** mais il faut faire attention au fait que **simple** n'a pas le même sens entre un graphe orienté symétrique et un graphe non orienté. En effet, $x \to y \to x$ est simple pour un graphe orienté alors que $x \smile y \smile x$ ne l'est pas vu qu'il s'agit de la même arête.

30.2.2 Connexité

Définition 30.27

On définit $\overset{*}{\smile}$ comme étant la fermeture réflexive et transitive de $\smile$.

Théorème 30.28

- $\overset{*}{\smile}$ est une relation d'équivalence dont les classes sont appelées les **composantes connexes** du graphe.
- $x \overset{*}{\smile} y \Leftrightarrow$ il existe un chemin de x à y.

Définition 30.29

Un graphe n'ayant qu'une classe d'équivalence pour $\overset{*}{\smile}$ est dit **connexe**. Cela signifie qu'il existe un chemin entre toute paire de sommets.

30.2.3 Graphe acyclique connexe

Définition 30.30

On appelle **arbre** un graphe acyclique connexe. Quand on a distingué un sommet, on parle d'**arbre enraciné**.

30.2.4 Graphe biparti

Définition 30.31

Soit $G = (S, A)$ un graphe, on dit que le graphe est **biparti** lorsqu'il existe une partition $S = S_1 \cup S_2$ telle que

$$\forall \{x, y\} \in A, x \in S_1 \Leftrightarrow y \in S_2$$

Par exemple, le graphe suivant est biparti avec $S_1 = \{a, e, f, g\}$ et $S_2 = \{b, c, d\}$.

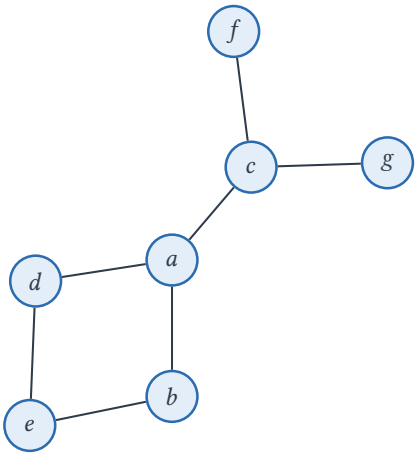


Figure 30.4 — Un graphe biparti.

On peut s'en rendre compte en coloriant les sommets.

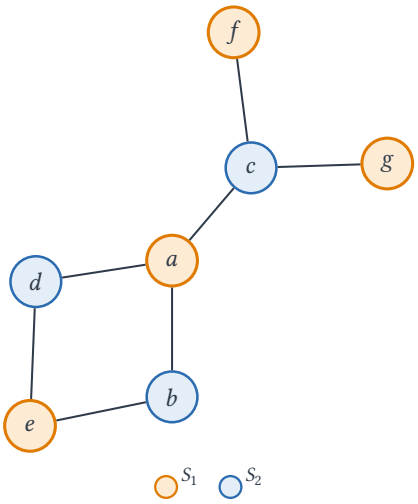


Figure 30.5 — Le même graphe colorié : $S_1 = \{a, e, f, g\}$ en orange, $S_2 = \{b, c, d\}$ en bleu.

En fait, être biparti est équivalent à pouvoir être colorié en deux couleurs de sorte que deux sommets reliés soient de couleurs différentes.

30.3 Graphes classiques

On présente ici brièvement des graphes ou des familles de graphes classiques qui serviront, notamment pour les exemples.

30.3.1 Graphes complets

Définition 30.32

Soit $n \in \mathbb{N}^*$, on appelle **graphe complet** à n sommets le graphe non orienté K_n de sommets $[\![0, n-1]\!]$ et tel que

$$\forall x, y \in [\![0, n-1]\!], x \neq y \Leftrightarrow x \frown y$$

K_5 :

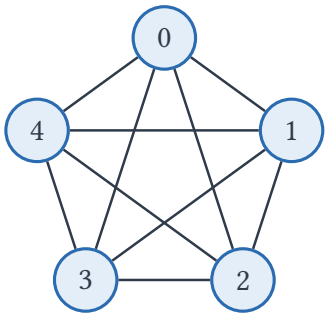


Figure 30.6 — Le graphe complet K_5.

K_8 :

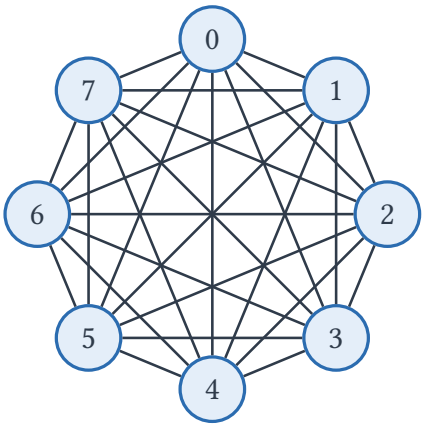


Figure 30.7 — Le graphe complet K_8.

Définition 30.33

Soit $n, p \in \mathbb{N}^*$, on appelle **graphe biparti complet** à (n, p) sommets le graphe non orienté $K_{n,p}$ de sommets $[\![0, n+p-1]\!]$ et tel que

$$\forall x \in [\![0, n-1]\!], \forall y \in [\![n, n+p-1]\!], x \smile y$$

$K_{3,4}$:

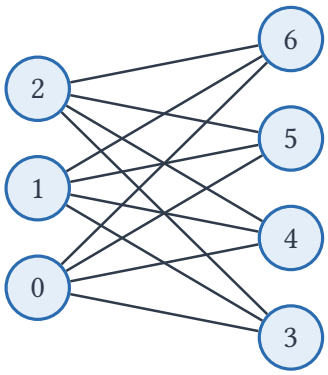


Figure 30.8 — Le graphe biparti complet $K_{3,4}$.

30.3.2 Cycles

Définition 30.34

Soit $n \geq 3$, on appelle **cycle** de longueur n le graphe non orienté C_n de sommets $[\![0, n-1]\!]$ et tel que

$$\forall x, y \in [\![0, n-1]\!], y \equiv x \pm 1 \ [n] \Leftrightarrow x \frown y$$

C_5 :

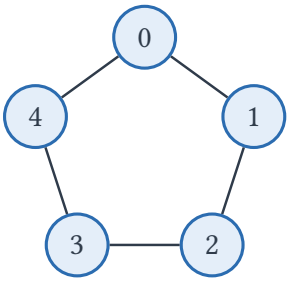


Figure 30.9 — Le cycle C_5.

C_8 :

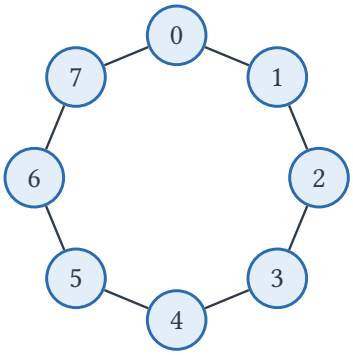


Figure 30.10 — Le cycle C_8.

Définition 30.35

Soit $n \geq 2$, on appelle **cycle orienté** de longueur n le graphe orienté C_n^o de sommets $[\![0, n-1]\!]$ et tel que

$$\forall x, y \in [\![0, n-1]\!], y \equiv x + 1 \ [n] \Leftrightarrow x \to y$$

C_5^o :

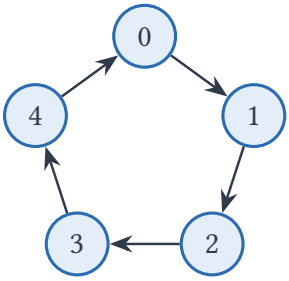


Figure 30.11 — Le cycle orienté C_5^o.

C_8^o :

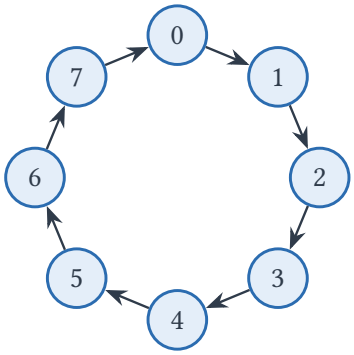


Figure 30.12 — Le cycle orienté C_8^o.

30.3.3 Grilles et hypercubes

On va manipuler ici des vecteurs de dimension d à coordonnées entières dans $[\![0, n-1]\!]$.

On sait que $[\![0, n-1]\!]^d$ est en bijection avec $[\![0, n^d - 1]\!]$ et on considère la bijection explicite suivante :

$$\varphi(x) = \sum_{i=0}^{d-1} x_i n^i \quad \text{où} \quad x = (x_0, ..., x_{d-1})$$

Soit $x = (x_0, ..., x_{d-1})$ et $y = (y_0, ..., y_{d-1}) \in [\![0, n-1]\!]^d$, on note $d_1(x, y) = \sum_{i=0}^{d-1} | x_i - y_i |$ qui compte la somme des différences de coordonnées. Ainsi $d_1((1, 2, 3), (3, 2, 2)) = | 1 - 3 | + | 2 - 2 | + | 3 - 2 | = 3$. Comme les coordonnées sont entières, on a le théorème suivant qui indique que les vecteurs à distance 1 sont ceux qui diffèrent exactement de 1 dans une unique coordonnée :

Théorème 30.36

Soient $x, y \in [\![0, n-1]\!]^d$, on a

$$d_1(x, y) = 1 \Leftrightarrow \exists i \in [\![0, d-1]\!], \{\forall j \in [\![0, d-1]\!], j \neq i \Rightarrow x_j = y_j$$
$$| x_i - y_i | = 1$$

On en déduit un graphe de grille des vecteurs à distance 1 :

Définition 30.37

Soient $n, d \in \mathbb{N}^*$, on appelle **grille de dimension d et de taille n** le graphe G_n^d dont les sommets sont dans $[\![0, n-1]\!]^d$ et où

$$\forall x, y \in [\![0, n-1]\!]^d, x \smile y \Leftrightarrow d_1(x, y) = 1$$

G_3^2 :

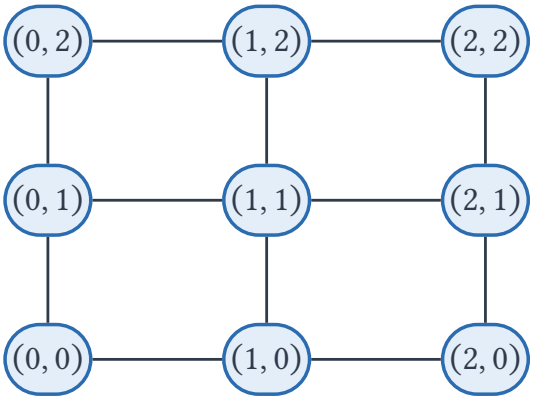


Figure 30.13 — La grille G_3^2.

G_2^3 :

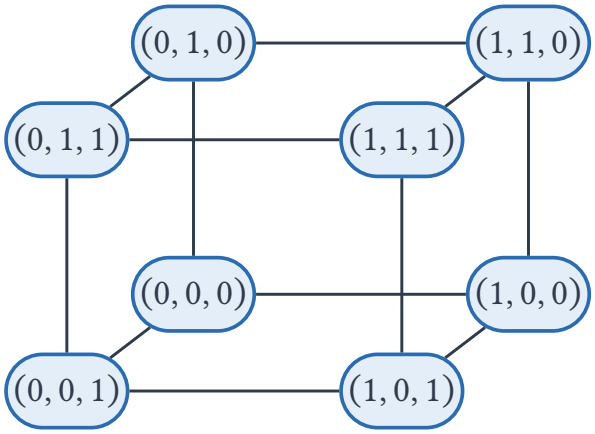


Figure 30.14 — La grille G_2^3, c'est-à-dire l'hypercube H_3.

Définition 30.38

Soit $n \in \mathbb{N}^*$, on appelle **hypercube de dimension** n le graphe $H_n = G_2^n$.

On vient de voir l'hypercube de dimension 3 (G_2^3) dans l'exemple précédent.

Remarque 30.39 L'hypercube de dimension n est intéressant car il permet de représenter chaque nombre de $[\![0, 2^n - 1]\!]$ comme un sommet du graphe (via la bijection φ : les coordonnées sont les bits). Par exemple, toute chaîne hamiltonienne, c'est-à-dire qui contient chaque sommet du graphe, correspond à un ordre d'énumération des nombres qui vérifie qu'un nombre et son successeur ne diffèrent que sur un bit.

30.4 Parcours

30.4.1 Principe

La grande majorité des algorithmes sur les graphes consistent à parcourir les sommets de voisins en voisins pour effectuer des traitements, la manière dont on les parcourt pouvant changer selon les différentes applications.

Citons, par exemple, le fait de déterminer les composantes connexes d'un graphe ou de trouver le plus court chemin entre deux sommets.

On va se placer dans le cadre d'un graphe orienté représenté par listes d'adjacence et en supposant que les sommets sont identifiés par leur indice. Dans ce cadre, si on a, en fait, un graphe non orienté, il sera représenté par son graphe orienté symétrique comme on l'a vu plus haut.

On va également considérer qu'on veut effectuer un traitement, ou une visite, pour ces sommets ou les arêtes empruntées.

30.4.2 Parcours en profondeur récursif

On présente ici une première manière élémentaire de les parcourir en tirant parti de la récursivité : on considère une fonction `parcours` et l'appel à `parcours` pour le sommet `x` va effectuer des appels récursifs à `parcours` pour chaque sommet y de $v_+(x)$.

Le problème est qu'on ne veut pas traiter deux fois un sommet et on veut que les appels terminent. Pour cela, on introduit une notion d'état associé à chaque sommet. Un sommet peut être

- **Inconnu**, cela correspond au fait qu'il n'est pas encore apparu en tant que voisin.
- **Découvert**, il est apparu mais n'a pas encore été traité complètement.
- **Traité** (ou **Visité**), il a non seulement été traité, mais également tous les sommets parcourus grâce à lui.

Pour maintenir cet état dans le code, le plus simple est de considérer un tableau d'entiers `etat` où `etat[i]` donne l'état du sommet `i`.

Définition 30.40

Ce parcours est appelé un **parcours en profondeur** récursif. En anglais, on parle de **depth-first search** et on utilise couramment l'acronyme **DFS**.

Première version

On adapte directement le principe précédent en un programme.

Python

```python
INCONNU = 0
DECOUVERT = 1
TRAITE = 2

def parcours(ladj, x, etat):
    if etat[x] != TRAITE:
        print(x)
        for y in ladj[x]:
            if etat[y] == INCONNU:
                etat[y] = DECOUVERT
                parcours(ladj, y, etat)
        etat[x] = TRAITE

def lance_parcours(ladj, x):
    # État initial inconnu pour tous
    etat = [ INCONNU ] * len(ladj)
    etat[x] = DECOUVERT
    parcours(ladj, x, etat)
```

OCaml

```
type etat = Inconnu | Decouvert | Traite

let rec parcours ladj x etat =
    if etat.(x) <> Traite
    then begin
        Printf.printf "%d\n" x;
        List.iter (fun y ->
            if etat.(y) = Inconnu
            then begin
                etat.(y) <- Decouvert;
                parcours ladj y etat
            end
        ) ladj.(x);
        etat.(x) <- Traite
    end
```

```
let lance_parcours ladj x =
    (* État initial inconnu pour tous *)
    let etat = Array.make (Array.length ladj) Inconnu in
    etat.(x) <- Decouvert;
    parcours ladj x etat
```

C

```c
#include <stdio.h>
#include <stdlib.h>

enum etat { INCONNU, DECOUVERT, TRAITE };

void parcours(graphe g, int x, int *etat) {
    if (etat[x] != TRAITE) {
        printf("%d\n", x);
        for (int i = 0; i < g.degre[x]; i++) {
            int y = g.voisins[x][i];
            if (etat[y] == INCONNU) {
                etat[y] = DECOUVERT;
                parcours(g, y, etat);
            }
        }
        etat[x] = TRAITE;
    }
}

void lance_parcours(graphe g, int x) {
    // État initial inconnu pour tous
    int *etat = malloc(g.n * sizeof(int));
    for (int i = 0; i < g.n; i++) {
        etat[i] = INCONNU;
    }
    etat[x] = DECOUVERT;
    parcours(g, x, etat);
    free(etat);
}
```

Ici, on se contente d'afficher les sommets rencontrés. En OCaml, on utilise un type somme pour les trois états ; en C, une énumération et le type `graphe` défini plus haut.

Exemple 30.41

On va réaliser un parcours en profondeur du graphe suivant en partant du sommet 0 :

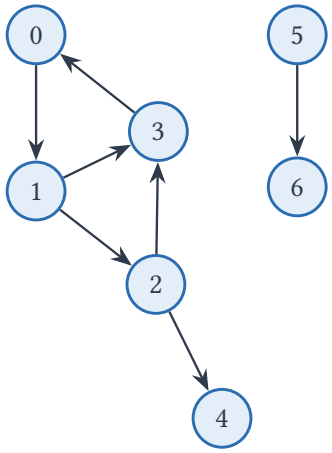
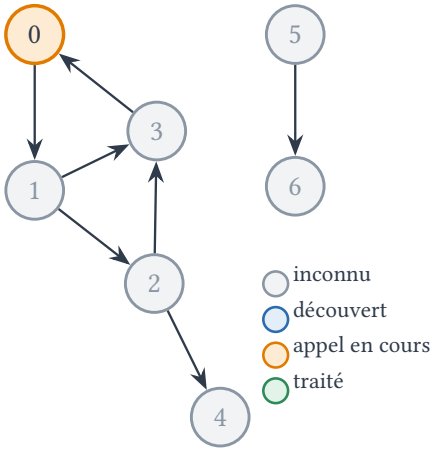


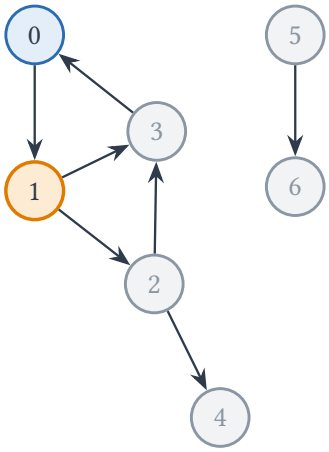
Figure 30.15 — Graphe orienté parcouru en profondeur depuis 0.

```
On indique les sommets inconnus en gris, les sommets découverts en bleu
(en orange celui dont l'appel est en cours) et les sommets traités en vert.
```

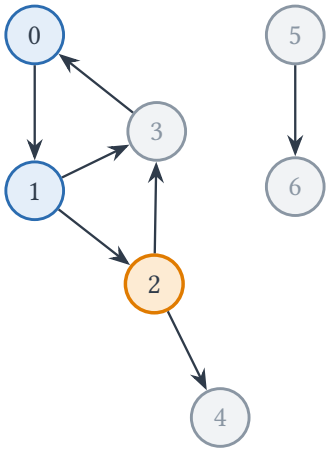
```
On affiche 0 et on relance le
parcours du voisin 1.
```



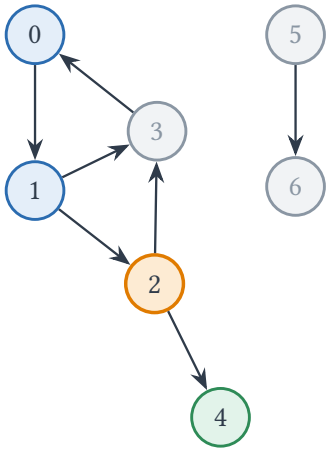

```
On affiche 1 et on relance le parcours du voisin 2.
```



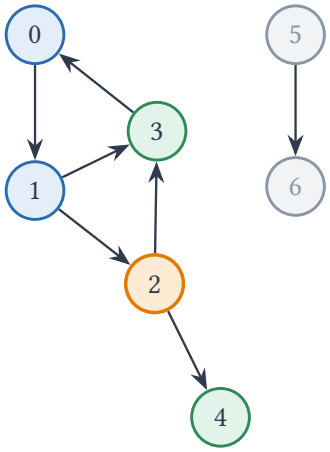

```
On affiche 2 et on relance le
parcours de son voisin 4 (on suppose ici que 4 apparaît avant 3 dans
la liste d'adjacence de 2).
```



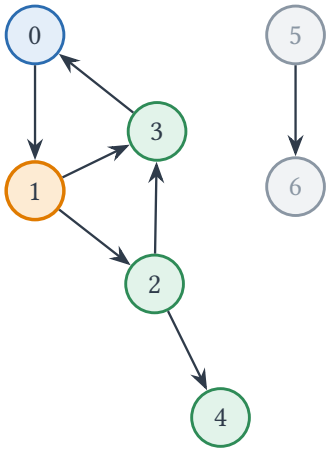
```
On affiche 4 et comme il n'a pas de voisins, on marque 4 comme traité
et on sort de l'appel.
```



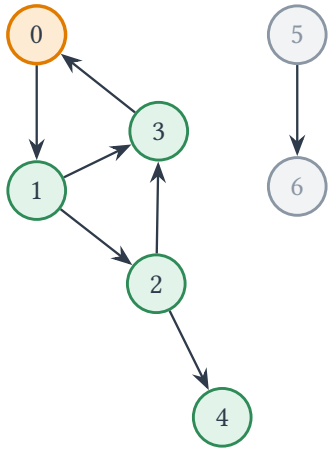

```
On revient à 2 et on relance le
parcours du voisin 3. On affiche 3, et on le marque comme traité
car son unique voisin 0 est déjà découvert.
```



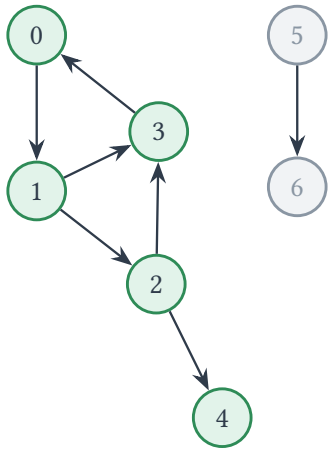

```
On revient à 2 et cette fois-ci il n'y a plus de voisins, on peut
marquer 2 comme étant traité.
```



```
On revient à 1, son autre voisin 3 est déjà traité, on peut marquer 1
comme étant traité.
```




```
On revient à 0 et on peut marquer 0 comme étant traité.
```




```
On a donc une vague de descente dans le graphe qui marque les sommets comme
découverts et ensuite on remonte la pile d'appels récursifs en marquant les
sommets comme étant traités.

Les sommets sont ici affichés dans l'ordre 0 1 2 4 3. On remarque que les
sommets 5 et 6 sont inaccessibles, ils restent ainsi inconnus tout le long du
parcours.
```

Arbre de parcours

On peut noter les arêtes empruntées lors du parcours précédent pour découvrir un nouveau sommet, et donc relancer un appel récursif. On obtient alors une structure arborescente qui est appelée l'arbre du parcours en profondeur.

Exemple 30.42

Dans le parcours précédent, on obtient l'arbre suivant :

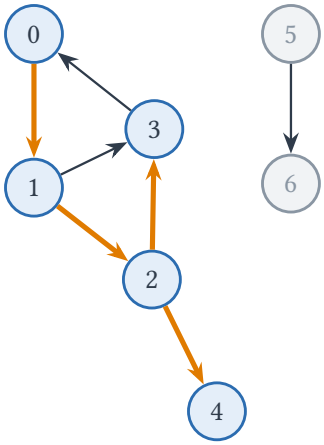


Figure 30.16 — Arêtes de l'arbre du parcours en profondeur (en orange) ; 5 et 6 restent inconnus.

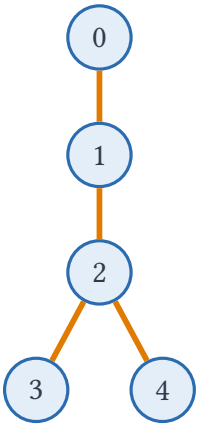


Figure 30.17 — L'arbre du parcours en profondeur issu de 0.

Pour obtenir cet arbre, on va construire un tableau `parent` pendant le parcours. Lorsqu'on emprunte une arête $x \to y$ pour découvrir y, on note $parent[y] = x$. Par défaut, $parent[y]$ est indéfini (`None` ou une valeur d'indice invalide comme -1).

Théorème 30.43

Si $parent[y]$ est défini à l'issue du parcours depuis x, alors il existe un chemin $\varphi : x \rightsquigarrow y$ où les arêtes $s \to s'$ empruntées lors du chemin vérifient toutes $parent[s'] = s$.

Preuve Quand on pose $parent[y] = s$, le sommet s est en cours de traitement, donc découvert avant y. Ainsi, en remontant les parents depuis y, on obtient des sommets découverts de plus en plus tôt : la remontée ne boucle pas et s'arrête sur le seul sommet découvert sans parent, à savoir x. En renversant cette suite, on obtient le chemin voulu, puisque $parent[y] = s$ n'est posé que si $s \to y$.

□

Comme $parent[y]$ est défini pour tous les sommets découverts autres que x et que les sommets découverts finissent tous par être traités, on en déduit que les sommets traités sont accessibles depuis x. La réciproque fait l'objet du théorème suivant :

Théorème 30.44

À l'issue du parcours depuis x, les sommets traités sont les sommets accessibles depuis x, c'est-à-dire les éléments de

$$\{y \in S \mid x \rightsquigarrow y\}$$

Preuve On vient de voir que les sommets traités sont accessibles depuis x. Réciproquement, montrons par récurrence sur p que si $\varphi : x \rightsquigarrow y$ est de longueur p, alors y est traité. Pour $p = 0$, $y = x$ est traité à la fin de l'appel initial. Si $p \geq 1$, on écrit $\varphi = \psi\,(z \to y)$ où ψ est de longueur $p - 1$: par hypothèse de récurrence, z est traité, donc l'appel sur z a parcouru tous ses voisins. Au moment où l'on examine y dans cet appel, soit y est inconnu et on lance un appel sur y qui se termine en le marquant traité, soit y est déjà découvert ou traité. Or tout sommet découvert est découvert juste avant l'appel sur lui et est traité à la fin de cet appel, qui termine avant la fin du parcours. Dans tous les cas, y est traité.

□

On présente le calcul de ces chemins dans le programme suivant :

Python

```python
INCONNU = 0
DECOUVERT = 1
TRAITE = 2

def parcours(ladj, x, etat, parent):
    if etat[x] != TRAITE:
        print(x)
        for y in ladj[x]:
            if etat[y] == INCONNU:
                parent[y] = x
                etat[y] = DECOUVERT
                parcours(ladj, y, etat, parent)
        etat[x] = TRAITE

def arbre_parcours(ladj, x):
    etat = [ INCONNU ] * len(ladj)
    parent = [ None ] * len(ladj)
    etat[x] = DECOUVERT
    parcours(ladj, x, etat, parent)
    return parent

def chemin(parent, y):
    # Renvoie le chemin x -> ... -> y
    # en sens inverse
    p = [ y ]
    while parent[y] != None:
        y = parent[y]
        p.append(y)
    return p
```

OCaml

```ocaml
type etat = Inconnu | Decouvert | Traite

let rec parcours ladj x etat parent =
    if etat.(x) <> Traite
    then begin
        Printf.printf "%d\n" x;
        List.iter (fun y ->
            if etat.(y) = Inconnu
            then begin
                parent.(y) <- x;
                etat.(y) <- Decouvert;
                parcours ladj y etat parent
            end
        ) ladj.(x);
        etat.(x) <- Traite
    end

(* -1 signifie que le parent est indéfini *)
let arbre_parcours ladj x =
    let n = Array.length ladj in
    let etat = Array.make n Inconnu in
```

```
    let parent = Array.make n (-1) in
    etat.(x) <- Decouvert;
    parcours ladj x etat parent;
    parent

(* Renvoie le chemin x -> ... -> y, cette fois dans le bon sens *)
let chemin parent y =
    let rec remonte y acc =
        if parent.(y) = -1
        then y :: acc
        else remonte parent.(y) (y :: acc)
    in
    remonte y []
```

C

```c
#include <stdio.h>
#include <stdlib.h>

enum etat { INCONNU, DECOUVERT, TRAITE };

void parcours(graphe g, int x, int *etat, int *parent) {
    if (etat[x] != TRAITE) {
        printf("%d\n", x);
        for (int i = 0; i < g.degre[x]; i++) {
            int y = g.voisins[x][i];
            if (etat[y] == INCONNU) {
                parent[y] = x;
                etat[y] = DECOUVERT;
                parcours(g, y, etat, parent);
            }
        }
        etat[x] = TRAITE;
    }
}

// Renvoie un tableau parent alloué, -1 signifiant indéfini
int *arbre_parcours(graphe g, int x) {
    int *etat = malloc(g.n * sizeof(int));
    int *parent = malloc(g.n * sizeof(int));
    for (int i = 0; i < g.n; i++) {
        etat[i] = INCONNU;
        parent[i] = -1;
    }
    etat[x] = DECOUVERT;
    parcours(g, x, etat, parent);
    free(etat);
    return parent;
}

// Écrit dans p le chemin x -> ... -> y en sens inverse
// et renvoie sa longueur (p doit pouvoir contenir n sommets)
int chemin(int *parent, int y, int *p) {
    int k = 0;
    p[k++] = y;
    while (parent[y] != -1) {
        y = parent[y];
        p[k++] = y;
    }
    return k;
}
```

Composantes connexes

On peut déduire directement du théorème précédent, le corollaire suivant dans le cas non orienté (représenté par son graphe orienté symétrique, dans lequel $x \rightsquigarrow y \Leftrightarrow x \overset{*}{\smile} y$) :

Théorème 30.45

Dans un graphe non orienté, les sommets traités depuis un parcours en profondeur issu d'un sommet sont exactement les sommets de sa composante connexe.

On en déduit alors un algorithme pour obtenir les composantes connexes d'un graphe non orienté :

Algorithme 30.46 — : Composantes connexes

Entrées :

Un graphe non orienté $G = (S, A)$

Déroulement :

- Tant qu'il y a des sommets non traités
  - On choisit un sommet non traité x
  - On effectue un parcours récursif depuis x (sans réinitialiser les états)
  - Tous les sommets nouvellement traités par ce parcours forment une composante connexe

On va en déduire le programme suivant :

Python

```python
INCONNU = 0
DECOUVERT = 1
TRAITE = 2

def parcours(ladj, x, etat, composante):
    if etat[x] != TRAITE:
        composante.append(x)
        for y in ladj[x]:
            if etat[y] == INCONNU:
                etat[y] = DECOUVERT
                parcours(ladj, y, etat, composante)
        etat[x] = TRAITE

def composantes_connexes(ladj):
    composantes = []
    etat = [ INCONNU ] * len(ladj)
    for i in range(len(ladj)):
        if etat[i] == INCONNU:
            composante = [ ]
            etat[i] = DECOUVERT
            parcours(ladj, i, etat, composante)
            composantes.append( composante )
    return composantes
```

OCaml

```ocaml
type etat = Inconnu | Decouvert | Traite

let rec parcours ladj x etat composante =
    if etat.(x) <> Traite
    then begin
        composante := x :: !composante;
        List.iter (fun y ->
            if etat.(y) = Inconnu
            then begin
                etat.(y) <- Decouvert;
                parcours ladj y etat composante
            end
        ) ladj.(x);
        etat.(x) <- Traite
    end

let composantes_connexes ladj =
    let n = Array.length ladj in
```

```
    let composantes = ref [] in
    let etat = Array.make n Inconnu in
    for i = 0 to n - 1 do
        if etat.(i) = Inconnu
        then begin
            let composante = ref [] in
            etat.(i) <- Decouvert;
            parcours ladj i etat composante;
            composantes := List.rev !composante :: !composantes
        end
    done;
    List.rev !composantes
```

C

```c
#include <stdlib.h>

enum etat { INCONNU, DECOUVERT, TRAITE };

void parcours(graphe g, int x, int *etat, int *composante, int num) {
    if (etat[x] != TRAITE) {
        composante[x] = num;
        for (int i = 0; i < g.degre[x]; i++) {
            int y = g.voisins[x][i];
            if (etat[y] == INCONNU) {
                etat[y] = DECOUVERT;
                parcours(g, y, etat, composante, num);
            }
        }
        etat[x] = TRAITE;
    }
}

// Remplit composante[x] avec le numéro de la composante de x
// et renvoie le nombre de composantes
int composantes_connexes(graphe g, int *composante) {
    int nb = 0;
    int *etat = malloc(g.n * sizeof(int));
    for (int i = 0; i < g.n; i++) {
        etat[i] = INCONNU;
    }
    for (int i = 0; i < g.n; i++) {
        if (etat[i] == INCONNU) {
            etat[i] = DECOUVERT;
            parcours(g, i, etat, composante, nb);
            nb++;
        }
    }
    free(etat);
    return nb;
}
```

✍ En C, plutôt que de construire une liste de listes, on numérote les composantes dans un tableau : deux sommets sont dans la même composante si et seulement s'ils ont le même numéro.

Détection de cycles

Si on est en train de traiter le sommet x et qu'on rencontre une arête $x \to y$ où y est découvert mais non traité, c'est qu'on est dans l'appel récursif de y et donc que x est un de ses descendants : $y \rightsquigarrow x$. En rajoutant la nouvelle arête, $y \rightsquigarrow x \to y$, on en déduit un cycle dans le cas d'un graphe orienté.

Pour un graphe non orienté représenté par un graphe orienté symétrique, il faut faire attention à ne pas prendre un aller-retour $x \to y \to x$ pour un cycle. On demande donc à avoir la condition :

```
etat[y] == DECOUVERT and parent[x] != y
```

On en déduit le programme suivant :

Python

```python
INCONNU = 0
DECOUVERT = 1
TRAITE = 2

def parcours(ladj, x, etat, parent, est_oriente):
    if etat[x] != TRAITE:

        for y in ladj[x]:
            if etat[y] == INCONNU:
                parent[y] = x
                etat[y] = DECOUVERT
                cycle = parcours(ladj, y, etat, parent, est_oriente)
                if cycle != None:
                    return cycle
            elif etat[y] == DECOUVERT and (est_oriente or parent[x] != y):
                # y est un ancêtre de x : on renvoie le cycle y -> ... -> x -> y
                cycle = [ y ]
                z = x
                while z != y:
                    cycle.append(z)
                    z = parent[z]
                cycle.append(y)
                cycle.reverse()
                return cycle

        etat[x] = TRAITE
    return None

def detecte_cycle(ladj, x, est_oriente):
    etat = [ INCONNU ] * len(ladj)
    parent = [ None ] * len(ladj)
    etat[x] = DECOUVERT
    cycle = parcours(ladj, x, etat, parent, est_oriente)
    if cycle != None:
        print('Cyclique : ', cycle)
    else:
        print('Acyclique')
```

OCaml

```ocaml
type etat = Inconnu | Decouvert | Traite

exception Cycle of int list

let rec parcours ladj x etat parent est_oriente =
    if etat.(x) <> Traite
    then begin
        List.iter (fun y ->
            if etat.(y) = Inconnu
            then begin
                parent.(y) <- x;
                etat.(y) <- Decouvert;
                parcours ladj y etat parent est_oriente
            end
            else if etat.(y) = Decouvert && (est_oriente || parent.(x) <> y)
            then begin
                (* y est un ancêtre de x : on construit le cycle
                   y -> ... -> x -> y en remontant depuis x *)
                let rec remonte z acc =
                    if z = y
                    then y :: acc
                    else remonte parent.(z) (z :: acc)
                in
                raise (Cycle (remonte x [y]))
            end
        ) ladj.(x);
        etat.(x) <- Traite
```

```
    end

let detecte_cycle ladj x est_oriente =
    let n = Array.length ladj in
    let etat = Array.make n Inconnu in
    let parent = Array.make n (-1) in
    etat.(x) <- Decouvert;
    try
        parcours ladj x etat parent est_oriente;
        print_endline "Acyclique"
    with Cycle c ->
        print_string "Cyclique :";
        List.iter (Printf.printf " %d") c;
        print_newline ()
```

C

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

enum etat { INCONNU, DECOUVERT, TRAITE };

// Renvoie la longueur du cycle écrit dans le tableau cycle
// (de taille n + 1), ou 0 si aucun cycle n'est trouvé
int parcours(graphe g, int x, int *etat, int *parent, bool est_oriente,
             int *cycle) {
    if (etat[x] != TRAITE) {
        for (int i = 0; i < g.degre[x]; i++) {
            int y = g.voisins[x][i];
            if (etat[y] == INCONNU) {
                parent[y] = x;
                etat[y] = DECOUVERT;
                int k = parcours(g, y, etat, parent, est_oriente, cycle);
                if (k != 0) {
                    return k;
                }
            } else if (etat[y] == DECOUVERT
                       && (est_oriente || parent[x] != y)) {
                // y est un ancêtre de x : on écrit le cycle y -> ... -> x -> y
                // à l'envers puis on le retourne
                int k = 0;
                cycle[k++] = y;
                for (int z = x; z != y; z = parent[z]) {
                    cycle[k++] = z;
                }
                cycle[k++] = y;
                for (int a = 0, b = k - 1; a < b; a++, b--) {
                    int tmp = cycle[a];
                    cycle[a] = cycle[b];
                    cycle[b] = tmp;
                }
                return k;
            }
        }
        etat[x] = TRAITE;
    }
    return 0;
}

void detecte_cycle(graphe g, int x, bool est_oriente) {
    int *etat = malloc(g.n * sizeof(int));
    int *parent = malloc(g.n * sizeof(int));
    int *cycle = malloc((g.n + 1) * sizeof(int));
    for (int i = 0; i < g.n; i++) {
        etat[i] = INCONNU;
        parent[i] = -1;
    }
```

```
    etat[x] = DECOUVERT;
    int k = parcours(g, x, etat, parent, est_oriente, cycle);
    if (k != 0) {
        printf("Cyclique :");
        for (int i = 0; i < k; i++) {
            printf(" %d", cycle[i]);
        }
        printf("\n");
    } else {
        printf("Acyclique\n");
    }
    free(etat);
    free(parent);
    free(cycle);
}
```

En OCaml, on interrompt le parcours avec une exception dès qu'un cycle est trouvé, `List.iter` ne permettant pas de sortir de la boucle.

Classification des arêtes

On considère ici un **parcours complet** d'un graphe orienté G : comme pour les composantes connexes, tant qu'il reste des sommets inconnus, on en choisit un et on lance un parcours depuis celui-ci, sans réinitialiser les états. À la fin, tous les sommets sont traités.

Les arêtes de découverte $parent[y] \to y$ forment alors une **forêt**, la forêt du parcours : chaque sommet a au plus un parent et, en remontant les parents, on aboutit à une **racine**, c'est-à-dire un sommet depuis lequel on a lancé un parcours. On parle d'**ancêtre** et de **descendant** dans cette forêt, en convenant qu'un sommet est ancêtre et descendant de lui-même.

Lors du parcours, chaque arête $x \to y$ est examinée exactement une fois, par l'appel sur x. On classe les arêtes selon la position de y par rapport à x dans la forêt.

Définition 30.47

Soit $x \to y$ une arête de G. On dit que c'est une arête :

- **de liaison** si $parent[y] = x$, c'est-à-dire si elle a servi à découvrir y ;
- **arrière** si y est un ancêtre de x ;
- **avant** si y est un descendant de x et que ce n'est pas une arête de liaison ;
- **transverse** sinon, c'est-à-dire quand aucun des deux sommets n'est un ancêtre de l'autre.

Comme $x \neq y$, y ne peut pas être à la fois un ancêtre et un descendant de x : ces quatre cas sont exclusifs.

Exemple 30.48

On reprend le graphe de l'exemple du parcours en profondeur

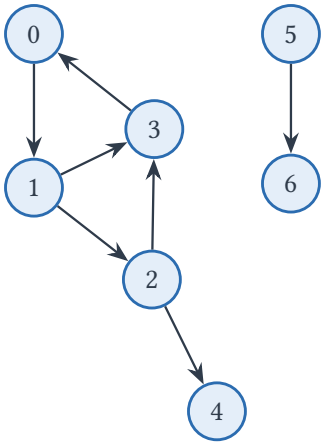


Figure 30.18 — Graphe orienté parcouru en profondeur depuis 0.

```
avec les listes d'adjacence `[1]`, `[2, 3]`, `[4, 3]`, `[0]`, `[]`, `[6]`,
`[]`, et on effectue un parcours complet en lançant des parcours depuis 0
puis depuis 5.

* Les arêtes de liaison sont $0 \rightarrow 1$, $1 \rightarrow 2$,
  $2 \rightarrow 4$, $2 \rightarrow 3$ et $5 \rightarrow 6$.
* $3 \rightarrow 0$ est une arête arrière : 0 est un ancêtre de 3.
* $1 \rightarrow 3$ est une arête avant : 3 est un descendant de 1, mais il
  a été découvert depuis 2.
* Il n'y a pas d'arête transverse.

Si la liste d'adjacence de 1 était `[3, 2]`, on découvrirait 3 depuis 1, puis
2 depuis 1 : l'arête $2 \rightarrow 3$ deviendrait transverse et
$1 \rightarrow 3$ une arête de liaison. La classification dépend donc de
l'ordre des listes d'adjacence et pas seulement du graphe.
```

Pour relier cette classification au déroulement du parcours, on utilise la pile des appels récursifs.

Lemme 30.49

À tout moment d'un parcours complet :

- les sommets découverts non traités sont exactement ceux dont l'appel est en cours, et, dans l'ordre de leur découverte, ils forment une branche $s_0, s_1, \ldots, s_k$ de la forêt : s_0 est une racine et $parent[s_i] = s_{i-1}$ pour $i \in [\![1, k]\!]$;
- y est un descendant de x si et seulement si $y = x$ ou y est découvert à un moment où x est découvert non traité.

Preuve Un sommet est marqué découvert juste avant l'appel sur lui et traité à la fin de cet appel, d'où la première affirmation. Les appels en cours forment la pile d'appels : le plus ancien a été lancé sur une racine par la boucle du parcours complet, et chacun des autres a été lancé par l'appel situé juste en dessous dans la pile, lors de l'examen d'une arête $s_{i-1} \to s_i$, ce qui a posé $parent[s_i] = s_{i-1}$.

Si $y \neq x$ est découvert alors que x est découvert non traité, x est dans la pile $s_0, \ldots, s_k$ et y est découvert par l'appel au sommet de la pile : $parent[y] = s_k$ et y prolonge la branche, donc x est un ancêtre de y.

Réciproquement, on raisonne par récurrence sur le nombre de remontées de parents nécessaires pour aller de y à x. Si $y = x$, il n'y a rien à montrer. Sinon, $z = parent[y]$ est un descendant de x et y est découvert par l'appel sur z, donc alors que z est découvert non traité. Si $z = x$, c'est terminé. Sinon, par hypothèse de récurrence, z a été découvert alors que x était dans la pile : x est en dessous de z dans la pile et y reste tant que l'appel sur z n'est pas terminé. Ainsi x est découvert non traité quand y est découvert.

□

Propriété 30.50

Au moment où l'appel sur x examine l'arête $x \to y$:

- y est inconnu si et seulement si l'arête est de liaison ;
- y est découvert non traité si et seulement si l'arête est arrière ;
- y est traité si et seulement si l'arête est avant ou transverse.

Preuve Les trois cas de gauche recouvrent toutes les possibilités et ceux de droite sont exclusifs : il suffit donc de montrer les implications de gauche à droite.

- Si y est inconnu, on pose $parent[y] = x$: l'arête est de liaison.
- Si y est découvert non traité, y est dans la pile d'appels, dont x est le sommet. D'après le lemme, y est un ancêtre de x : l'arête est arrière.
- Si y est traité, il n'est pas dans la pile, donc ce n'est pas un ancêtre de x. De plus, $parent[y] = x$ ne peut être posé que lors de l'examen de l'arête $x \to y$, qui n'a lieu qu'une fois, maintenant, et ne le pose pas puisque y n'est pas inconnu. L'arête est donc avant ou transverse.

□

Remarque 30.51 Le programme de détection de cycles vu plus haut s'arrête exactement à la première arête arrière rencontrée, et le cycle qu'il renvoie est formé de cette arête et de la branche de la forêt qui va de y à x. On va voir qu'un graphe orienté est cyclique si et seulement si un parcours complet rencontre une arête arrière : pour détecter un cycle n'importe où dans le graphe, et pas seulement parmi les sommets accessibles depuis x, il suffit donc de relancer ce parcours depuis chaque sommet encore inconnu.

Temps d'entrée et de sortie

✍ On ajoute au parcours complet une **horloge** : un compteur, initialement nul, qu'on incrémente à chaque fois qu'un sommet est découvert et à chaque fois qu'un sommet devient traité.

Définition 30.52

Le **temps d'entrée** $t_e(x)$ d'un sommet x est la valeur de l'horloge juste après son incrémentation lors de la découverte de x. Le **temps de sortie** $t_s(x)$ est sa valeur juste après son incrémentation quand x devient traité.

Chaque sommet est découvert une fois et traité une fois : les temps d'entrée et de sortie sont les entiers de 1 à $2\,|S|$, deux à deux distincts, et $t_e(x) < t_s(x)$ pour tout sommet x.

On calcule ces temps lors d'un parcours complet. Ici, c'est `parcours` qui marque le sommet comme découvert, au moment où on note son temps d'entrée.

Python

```python
INCONNU = 0
DECOUVERT = 1
TRAITE = 2

def parcours(ladj, x, etat, horloge, entree, sortie):
    # renvoie la valeur de l'horloge à la fin du traitement de x
    etat[x] = DECOUVERT
    horloge += 1
    entree[x] = horloge
    for y in ladj[x]:
        if etat[y] == INCONNU:
            horloge = parcours(ladj, y, etat, horloge, entree, sortie)
    etat[x] = TRAITE
    horloge += 1
    sortie[x] = horloge
    return horloge
```

```python
def dates(ladj):
    # parcours complet
    n = len(ladj)
    etat = [ INCONNU ] * n
    entree = [ None ] * n
    sortie = [ None ] * n
    horloge = 0
    for x in range(n):
        if etat[x] == INCONNU:
            horloge = parcours(ladj, x, etat, horloge, entree, sortie)
    return entree, sortie
```

OCaml

```ocaml
type etat = Inconnu | Decouvert | Traite

let rec parcours ladj x etat horloge entree sortie =
    etat.(x) <- Decouvert;
    incr horloge;
    entree.(x) <- !horloge;
    List.iter (fun y ->
        if etat.(y) = Inconnu
        then parcours ladj y etat horloge entree sortie
    ) ladj.(x);
    etat.(x) <- Traite;
    incr horloge;
    sortie.(x) <- !horloge

(* Parcours complet : renvoie les tableaux des dates d'entrée et de sortie *)
let dates ladj =
    let n = Array.length ladj in
    let etat = Array.make n Inconnu in
    let entree = Array.make n 0 in
    let sortie = Array.make n 0 in
    let horloge = ref 0 in
    for x = 0 to n - 1 do
        if etat.(x) = Inconnu
        then parcours ladj x etat horloge entree sortie
    done;
    entree, sortie
```

C

```c
#include <stdlib.h>

enum etat { INCONNU, DECOUVERT, TRAITE };

void parcours(graphe g, int x, int *etat, int *horloge,
              int *entree, int *sortie) {
    etat[x] = DECOUVERT;
    *horloge += 1;
    entree[x] = *horloge;
    for (int i = 0; i < g.degre[x]; i++) {
        int y = g.voisins[x][i];
        if (etat[y] == INCONNU) {
            parcours(g, y, etat, horloge, entree, sortie);
        }
    }
    etat[x] = TRAITE;
    *horloge += 1;
    sortie[x] = *horloge;
}

// Parcours complet : remplit les tableaux entree et sortie (de taille n)
void dates(graphe g, int *entree, int *sortie) {
    int *etat = malloc(g.n * sizeof(int));
    for (int i = 0; i < g.n; i++) {
        etat[i] = INCONNU;
```

```
    }
    int horloge = 0;
    for (int x = 0; x < g.n; x++) {
        if (etat[x] == INCONNU) {
            parcours(g, x, etat, &horloge, entree, sortie);
        }
    }
    free(etat);
}
```

Exemple 30.53

Sur le graphe de l'exemple précédent, avec les listes d'adjacence `[1]`, `[2, 3]`, `[4, 3]`, `[0]`, `[]`, `[6]`, `[]`, on obtient :

sommet	0	1	2	3	4	5	6
t_e	1	2	3	6	4	11	12
t_s	10	9	8	7	5	14	13

Si on écrit une parenthèse ouvrante étiquetée par x au temps $t_e(x)$ et une parenthèse fermante au temps $t_s(x)$, on obtient le mot bien parenthésé

$$(_0\,(_1\,(_2\,(_4\,)_4\,(_3\,)_3\,)_2\,)_1\,)_0\,(_5\,(_6\,)_6\,)_5$$

qui reflète exactement la forêt du parcours. C'est un fait général :

Théorème 30.54 — (des parenthèses)

Soient $x \neq y$ deux sommets. Après un parcours complet, on est dans un et un seul des trois cas suivants :

- les intervalles $[t_e(x), t_s(x)]$ et $[t_e(y), t_s(y)]$ sont disjoints, et aucun des deux sommets n'est un descendant de l'autre ;
- $t_e(x) < t_e(y) < t_s(y) < t_s(x)$ et y est un descendant de x ;
- $t_e(y) < t_e(x) < t_s(x) < t_s(y)$ et x est un descendant de y.

Preuve Les trois cas sont clairement exclusifs. Quitte à échanger x et y, on suppose $t_e(x) < t_e(y)$. Comme x est découvert avant y, x n'est pas un descendant de y d'après le lemme.

- Si $t_s(x) < t_e(y)$, les intervalles sont disjoints et y est découvert après que x est traité : d'après le lemme, y n'est pas un descendant de x.
- Sinon, $t_e(x) < t_e(y) < t_s(x)$: y est découvert alors que x est découvert non traité, donc y est un descendant de x d'après le lemme. De plus, x est alors dans la pile d'appels, en dessous de y, et y reste jusqu'à la fin de l'appel sur y : on a donc $t_s(y) < t_s(x)$.

□

On en déduit une caractérisation de la classification des arêtes à l'aide des temps d'entrée et de sortie.

Corollaire 30.55

Soit $x \to y$ une arête. Après un parcours complet :

- elle est de liaison ou avant si et seulement si $t_e(x) < t_e(y) < t_s(y) < t_s(x)$;
- elle est arrière si et seulement si $t_e(y) < t_e(x) < t_s(x) < t_s(y)$;
- elle est transverse si et seulement si $t_s(y) < t_e(x)$.

En particulier, l'arête est arrière si et seulement si $t_s(x) < t_s(y)$.

Preuve Les deux premiers points découlent directement du théorème des parenthèses. Si l'arête est transverse, les intervalles sont disjoints d'après ce même théorème. Or, d'après la propriété précédente, y est déjà traité quand l'appel sur x examine l'arête, donc $t_s(y) < t_s(x)$ et ainsi $t_s(y) < t_e(x)$.

Les quatre types d'arêtes recouvrant tous les cas et les trois conditions de droite étant exclusives, on a bien des équivalences. Enfin, dans les trois cas autres que celui d'une arête arrière, on a $t_s(y) < t_s(x)$.

□

Théorème 30.56

Un graphe orienté G est acyclique si et seulement si un parcours complet de G ne rencontre aucune arête arrière, et ceci quels que soient l'ordre dans lequel on choisit les sommets de départ et l'ordre des listes d'adjacence.

Preuve Si $x \to y$ est une arête arrière, y est un ancêtre de x : les arêtes de liaison donnent un chemin $y \rightsquigarrow x$ le long d'une branche de la forêt, dont les sommets sont distincts. En le complétant avec l'arête $x \to y$, on obtient un cycle.

Réciproquement, s'il n'y a aucune arête arrière, le corollaire donne $t_s(y) < t_s(x)$ pour toute arête $x \to y$. Si $s_0 \to s_1 \to \ldots \to s_p = s_0$ était un cycle, on aurait $t_s(s_0) > t_s(s_1) > \ldots > t_s(s_p) = t_s(s_0)$, ce qui est absurde.

□

Corollaire 30.57

Si G est un graphe orienté acyclique, les sommets triés dans l'ordre décroissant de leurs temps de sortie lors d'un parcours complet forment un tri topologique de G.

Preuve D'après le théorème précédent, il n'y a pas d'arête arrière, donc $t_s(y) < t_s(x)$ pour toute arête $x \to y$.

□

On retrouve ainsi le théorème de la partie Tri topologique, sans avoir à lancer les parcours depuis les sommets de degré entrant nul.

Remarque 30.58 Pour un graphe non orienté, représenté par son graphe orienté symétrique, il n'y a pas d'arête transverse. En effet, si $x \to y$ était transverse, on aurait $t_s(y) < t_e(x)$; mais l'arête $y \to x$ est examinée par l'appel sur y, donc avant $t_s(y)$, à un moment où x est encore inconnu : x deviendrait un descendant de y, ce qui est contradictoire. De plus, $x \to y$ est une arête avant si et seulement si $y \to x$ est une arête arrière qui n'est pas le retour d'une arête de liaison : c'est pour cela que la détection de cycles dans le cas non orienté exclut seulement le cas $parent[x] = y$.

30.4.3 Parcours quelconque

On va considérer ici que l'on dispose d'une structure de données `sac` dans laquelle on peut placer des éléments et en sortir. On considère aussi qu'on a un moyen de marquer les sommets. Par défaut, ils sont non marqués.

On considère alors l'algorithme suivant décrit en pseudo-code et auquel on fera référence comme étant le parcours **quelconque** dans la suite :

Algorithme 30.59 — : parcours quelconque

Entrées :

- Un graphe $G = (S, A)$
- Un sommet $s \in S$ source du parcours

Déroulement :

- On crée un sac ne contenant que s

- Tant que le sac est non vide
  - On tire un sommet x du sac
  - Si x n'est pas marqué
    - On marque x
    - pour chaque $x \to y \in A$
      - On ajoute y dans le sac

On remarque que les sommets marqués sont exactement les sommets accessibles depuis s, on retrouve ainsi le théorème vu pour le parcours récursif. En effet, un sommet n'est placé dans le sac que s'il est s ou le successeur d'un sommet marqué, donc par récurrence tout sommet marqué est accessible. Réciproquement, si $s = s_0 \to \ldots \to s_p$, on montre par récurrence sur i que s_i est marqué : quand s_{i-1} est marqué, on ajoute s_i dans le sac, et l'algorithme ne termine que lorsque le sac est vide, donc après avoir tiré s_i, qui est alors marqué s'il ne l'était pas déjà.

En fait, le parcours quelconque ne permet pas de déduire plus d'information que cela, mais dans un graphe non orienté c'est déjà suffisant pour en déduire les composantes connexes.

La question qui se pose alors est celle de la stratégie d'ajout/tirage dans le sac. Selon ce que l'on considère, l'ordre de visite des sommets va changer. On va considérer trois stratégies :

- LIFO : Last In First Out, le prochain sommet tiré est le dernier ajouté
- FIFO : First In First Out, le prochain sommet tiré est le sommet le plus anciennement ajouté
- aléatoire : on tire aléatoirement et uniformément un sommet du sac

Définition 30.60

Un sac avec une stratégie LIFO est appelé une **pile**.

Un sac avec une stratégie FIFO est appelé une **file**.

Exemple 30.61

Si on considère le graphe suivant :

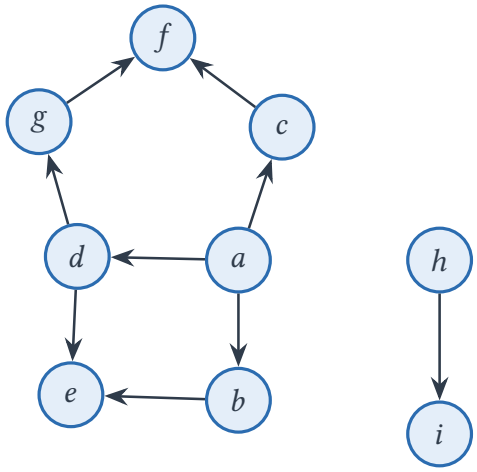


Figure 30.19 — Graphe orienté parcouru depuis a.

```
En effectuant un parcours quelconque issu de $a$  :

* en stratégie LIFO, on va voir les sommets dans l'ordre $a, b, e, c, f, d, g$
  (si on ajoute les voisins dans le sac en commençant par le dernier
  de la liste d'adjacence, sinon on obtient $a, d, g, f, e, c, b$).
* en stratégie FIFO, on va voir les sommets dans l'ordre
  $a, b, c, d, e, f, g$
```

```
* en stratégie aléatoire, on pourrait avoir
  $a, b, c, e, d, f, g$
```

Une manière de *voir* ces trois différentes stratégies est de considérer une grille et de colorer les sommets marqués avec une couleur changeant à chaque marquage. On peut voir dans l'ordre LIFO, FIFO, aléatoire l'effet de ces trois stratégies en partant du même point sur l'animation suivante :

[figure : animation disponible sur le site]

Remarque 30.62 La stratégie LIFO semble *bornée* : elle choisit un cap et elle s'y tient tant que c'est possible. La stratégie FIFO va elle s'étendre comme une onde partant du point de départ. De manière assez étonnante, la stratégie aléatoire semble être une bonne approximation de la stratégie FIFO, ce qui montre le caractère exceptionnel du LIFO.

Le parcours quelconque en stratégie FIFO s'appelle un *parcours en largeur*. Comme on va le voir, le parcours avec stratégie LIFO est exactement le *parcours en profondeur* précédent.

Implémentation en `Python`

On va supposer qu'on dispose de quatre fonctions manipulant des sacs :

- `sac_vide()` qui renvoie un nouveau sac vide
- `ajoute(sac, x)` qui ajoute `x` dans le sac `sac`
- `retire(sac)` qui retire un élément du sac `sac` et le renvoie
- `est_vide(sac)` qui renvoie un booléen indiquant si le `sac` est vide

Le programme du parcours quelconque s'écrit alors

```
def parcours_sac(ladj, source):
    a_traiter = sac_vide()
    ajoute(a_traiter, source)
    marque = [ False ] * len(ladj)

    while not est_vide(a_traiter):
        x = retire(a_traiter)
        if not marque[x]:
            marque[x] = True
            for y in ladj[x]:
                ajoute(a_traiter, y)
```

Remarque 30.63 Pour utiliser ce parcours, on peut rajouter des instructions au moment où on marque un sommet ou au moment où on traite une arête.

Pour implémenter un sac, on va d'abord utiliser le type `list` de `Python` :

- Pour créer un sac vide, on utilise la valeur `[]` :

```
def sac_vide():
    return []
```

- Pour tester si le sac est vide, on utilise donc le test à `[]` :

```
def est_vide(sac):
    return sac == []
```

- Pour ajouter un élément `x` dans le sac `s` dans tous les cas :

```
def ajoute(sac, x):
    sac.append(x)
```

- (LIFO) Pour retirer le dernier élément ajouté, on peut utiliser directement `pop` :

```
def retire(sac):
    return sac.pop()
```

- (FIFO) Pour retirer l'élément le plus anciennement ajouté, on peut utiliser la fonction de suppression `del l[0]` qui retire le premier élément

```
def retire(sac):
    x = sac[0]
    del sac[0]
    return x
```

- C'est anecdotique, mais pour retirer un élément au hasard, il faut choisir un indice `i`, renvoyer l'élément qui s'y trouve en le supprimant du sac avec `del s[i]` :

```
from random import randint
def retire(sac):
    i = randint(0, len(sac) - 1)
    x = sac[i]
    del sac[i]
    return x
```

Si on peut supposer que les quatre premières opérations sont en temps constant, les deux dernières sont linéaires en le nombre d'éléments dans le sac, ce qui est assez coûteux.

En effet, le parcours quelconque va accéder en pire cas à chaque sommet et à chaque liste d'adjacence, et ajouter un élément dans le sac pour chaque arête parcourue, soit au plus $|A|+1$ ajouts et donc au plus $|A|+1$ retraits dans un sac de taille au plus $|A|+1$. Donc, une complexité en $O(|S|+|A|+|A|\,f(|A|))$ où $f(n)$ est la complexité du retrait dans un sac de taille n.

On obtient donc $O(|S|+|A|)$ en LIFO mais $O(|S|+|A|^2)$ en FIFO avec cette implémentation.

En `Python`, il est possible d'avoir une file (`queue` en anglais) permettant de réaliser le retrait FIFO en $O(1)$:

- On importe le type `deque` : `from collections import deque`
- On crée une file vide avec `s = deque()`
- On ajoute un élément `x` avec `s.append(x)`
- Mais on dispose d'une fonction efficace `s.popleft()` pour retirer le premier élément.

En fait, on dispose également d'un `appendleft` et du `pop`. Toutes ces opérations étant en $O(1)$.

Dans l'implémentation précédente, il suffit donc de remplacer les fonctions par les suivantes :

```
from collections import deque

def sac_vide():
    return deque()

def est_vide(sac):
    return len(sac) == 0

def ajoute(sac, x):
    sac.append(x)

def retire(sac):
    return sac.popleft()
```

Remarque 30.64 La structure de données `deque` est une liste doublement chaînée (de blocs). Elle permet de remonter dans la chaîne d'un maillon vers le maillon dont il est le suivant.

En pratique, on pourra définir des fonctions de parcours utilisant directement les bonnes structures :

```python
def parcours_profondeur(ladj, source):
    a_traiter = [ source ]
    marque = [ False ] * len(ladj)

    while a_traiter != []:
        x = a_traiter.pop()
        if not marque[x]:
            marque[x] = True
            for y in ladj[x]:
                a_traiter.append(y)

### à placer en début du programme :
from collections import deque

def parcours_largeur(ladj, source):
    a_traiter = deque([ source ])
    marque = [ False ] * len(ladj)

    while len(a_traiter) != 0:
        x = a_traiter.popleft()
        if not marque[x]:
            marque[x] = True
            for y in ladj[x]:
                a_traiter.append(y)
```

30.4.4 Parcours en largeur

On a vu que le parcours quelconque en FIFO s'appelait le parcours en largeur. Il permet de parcourir les sommets d'un graphe en rayonnant à partir du sommet source. En effet on traite d'abord :

- le sommet source *src*
- les voisins de *src*
- les voisins des voisins de *src*
- ...

Pour pouvoir parler de chemins, il est nécessaire d'introduire une notion de parenté. Pour cela, on a deux possibilités.

- Soit on n'ajoute plus des sommets dans le sac mais des couples (sommet, parent) qui correspondent à une arête de découverte :

Algorithme 30.65 — : parcours quelconque avec couples

Entrées :

- Un graphe $G = (S, A)$
- Un sommet $s \in S$ source du parcours

Déroulement :

- On crée un sac ne contenant que (s, s)
- Tant que le sac est non vide
  - On tire un couple (x, p) du sac
  - Si x est non marqué
    - On marque x
    - On indique que $parent[x] = p$
    - pour chaque $x \to y \in A$
      - On ajoute (y, x) dans le sac

- Soit on modifie le parcours *quelconque* pour en déduire un parcours quelconque qui a l'air a priori d'être *optimisé*.

Algorithme 30.66 — : parcours quelconque *optimisé*

Entrées :

- Un graphe $G = (S, A)$
- Un sommet $s \in S$ source du parcours

Déroulement :

- On crée un sac ne contenant que s et on marque s
- Tant que le sac est non vide
  - On tire un sommet x du sac
  - pour chaque $x \to y \in A$
    - Si y est non marqué
      - On indique que $parent[y] = x$
      - On ajoute y dans le sac
      - On marque y

Exemple 30.67

On va reprendre l'exemple précédent

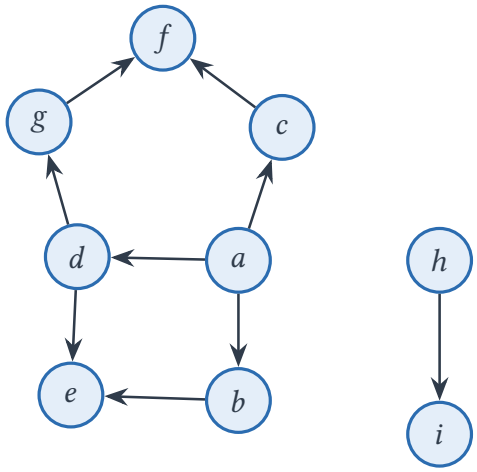


Figure 30.20 — Graphe orienté parcouru depuis a.

On obtient avec un parcours en largeur le schéma suivant :

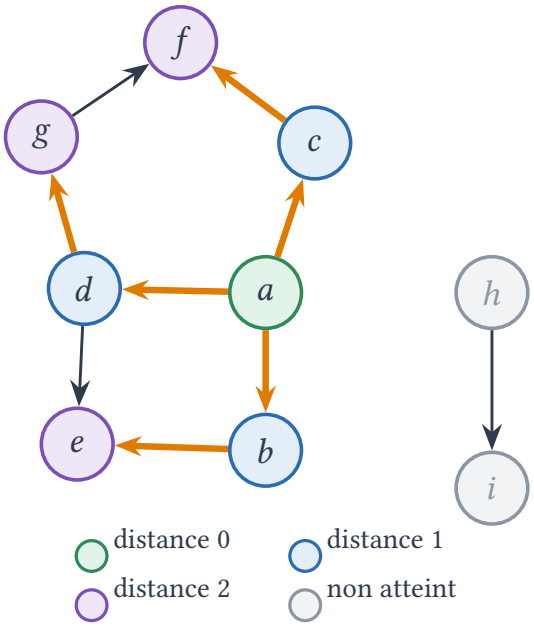


Figure 30.21 — Parcours en largeur depuis a : arêtes de parenté en orange, couleur selon la distance à a.

```
où on a noté les arêtes de parenté ainsi que d'une même couleur
les éléments à égale distance de $a$ dans le parcours en largeur.

On remarque qu'on a obtenu le chemin $a \rightarrow c \rightarrow f$
là où un parcours en profondeur aurait pu donner le chemin plus long
$a \rightarrow d \rightarrow g \rightarrow f$.
```

Comme semble le suggérer l'exemple précédent, on peut démontrer le théorème suivant :

Théorème 30.68

Les chemins donnés par un parcours en largeur depuis x sont de plus petite longueur :

Si $\varphi : x \rightsquigarrow y$ est donné par le parcours, alors $\forall \psi : x \rightsquigarrow y, |\psi| \geq |\varphi|$.

Preuve On raisonne sur le parcours *optimisé* avec une file, issu de x. Pour un sommet marqué y, on note $n(y)$ la longueur du chemin donné par le parcours : $n(x) = 0$ et $n(y) = n(\mathit{parent}[y]) + 1$. On note $\delta(y)$ la longueur minimale d'un chemin de x à y. Comme le chemin donné par le parcours est un chemin, on a toujours $\delta(y) \leq n(y)$.

Invariant. Si la file contient $y_1, \ldots, y_r$ (de la tête à la queue), alors $n(y_1) \leq n(y_2) \leq \ldots \leq n(y_r) \leq n(y_1) + 1$. C'est vrai initialement pour la file $[x]$. Si on retire y_1 et qu'on ajoute en queue des sommets z avec $n(z) = n(y_1) + 1$, on a bien $n(y_r) \leq n(y_1) + 1 = n(z) \leq n(y_2) + 1$, et l'invariant est préservé. On en déduit que les valeurs de n des sommets retirés successivement de la file forment une suite croissante.

Supposons par l'absurde qu'il existe un sommet y accessible avec $\delta(y) < n(y)$ et choisissons-le avec $\delta(y)$ minimal. On a $y \neq x$. Soit z le prédécesseur de y sur un chemin de longueur $\delta(y)$: on a $\delta(z) = \delta(y) - 1$, donc par minimalité $n(z) = \delta(z)$. Au moment où l'on retire z de la file :

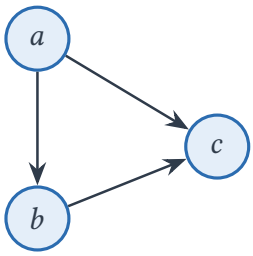
- soit y n'est pas encore marqué, et alors $\mathit{parent}[y] = z$ et $n(y) = n(z) + 1$;
- soit y a déjà été marqué lors du retrait d'un sommet w retiré avant z, donc $n(y) = n(w) + 1 \leq n(z) + 1$ par croissance.

Dans les deux cas $n(y) \leq n(z) + 1 = \delta(z) + 1 = \delta(y)$, ce qui est absurde.

□

30.4.5 Pseudo-parcours en profondeur

On peut se poser la question de la nature du parcours quelconque optimisé effectué avec une stratégie LIFO. Naïvement, on peut penser qu'il s'agit d'un parcours en profondeur. Cependant, si on considère le graphe :



On va avoir les deux arbres différents suivants :

Avec un parcours en profondeur :

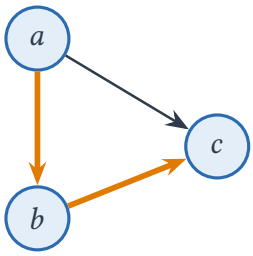


Figure 30.22 — Arbre du parcours en profondeur depuis *a* (en orange).

Avec un pseudo-parcours en profondeur :

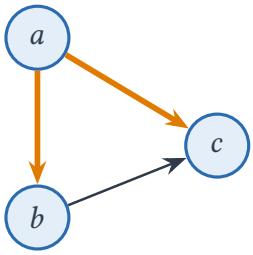


Figure 30.23 — Arbre du pseudo-parcours en profondeur depuis *a* (en orange).

On perd une propriété fondamentale du parcours en profondeur qui est qu'un parent ne peut empêcher un de ses enfants de découvrir un sommet. Ici, dans le pseudo-parcours, *a* bloque la découverte de *c* par *b*.

Exercice

On fixe un graphe, l'ordre de ses listes d'adjacence et un sommet de départ, et on compare l'arbre (le tableau `parent`) obtenu par le parcours quelconque *optimisé* à celui du parcours en profondeur récursif utilisant les mêmes listes d'adjacence.

1. Pour le graphe ci-dessus, parcouru depuis *a* avec la liste d'adjacence `[b, c]` pour *a*, montrer qu'aucune stratégie de tirage dans le sac ne permet au parcours optimisé d'obtenir l'arbre du parcours en profondeur.
2. Montrer qu'avec la liste d'adjacence `[c, b]` pour *a*, les deux arbres coïncident, quelle que soit la stratégie. L'énoncé « aucune stratégie ne convient » n'a donc de sens qu'à ordre des listes d'adjacence fixé.
3. Trouver un graphe et un sommet de départ pour lesquels, quel que soit l'ordre des listes d'adjacence, aucune stratégie ne permet au parcours optimisé d'obtenir l'arbre d'un parcours en profondeur.

30.5 Tri topologique

Un cas très important de graphes orientés est celui de graphes de dépendances :

- les sommets sont des tâches à réaliser
- une arête $x \to y$ indique que la tâche x doit être réalisée avant la tâche y.

Citons, par exemple, les dépendances entre chapitres dans un cours, entre unités de programme lors d'une compilation, entre produits dans un procédé industriel…

Ces graphes ont tous la particularité d'être orientés et acycliques. En effet, si on a un cycle, cela signifie qu'il y a une dépendance inextricable.

Remarque 30.69 On parle en anglais de **DAG** pour *directed acyclic graph.*

Définition 30.70

Soit $G = (S, A)$ un graphe orienté acyclique. Une énumération $(x_1, \ldots, x_n)$ des sommets de S est appelée un **tri topologique** lorsque pour toute arête $x_i \to x_j$ on a $i < j$.

Autrement dit, un tri topologique est un ordre linéaire de traitement des tâches respectant les dépendances.

Théorème 30.71

Soit G un graphe orienté acyclique et $x_1, \ldots, x_p$ les sommets de degré entrant nuls de G.

Si on effectue successivement des parcours en profondeur récursifs depuis chacun des x_i, en conservant les états d'un parcours à l'autre et avec la même horloge, alors les sommets triés dans l'ordre décroissant des temps de sortie constituent un tri topologique de G.

Ici, l'horloge est un compteur incrémenté à chaque fois qu'un sommet devient traité, et le temps de sortie $t_s(x)$ de x est la valeur de l'horloge à ce moment-là.

Preuve *Tous les sommets sont traités.* Soit $y \in S$. En remontant les prédécesseurs depuis y, on obtient un chemin qui ne peut pas répéter de sommet puisque G est acyclique : il est donc de longueur au plus $|S| - 1$ et se termine sur un sommet sans prédécesseur, c'est-à-dire un x_i. Ainsi y est accessible depuis un x_i, il est donc traité par le parcours issu de x_i (ou par un parcours antérieur).

Ordre. Il suffit de montrer que pour toute arête $x \to y$, on a $t_s(x) > t_s(y)$. Au moment où l'appel sur x examine l'arête $x \to y$:

- si y est inconnu, l'appel sur y est lancé et se termine avant celui sur x, donc $t_s(y) < t_s(x)$;
- si y est traité, $t_s(y)$ est déjà fixé alors que x n'est pas encore traité, donc $t_s(y) < t_s(x)$;
- si y est découvert mais non traité, l'appel sur y est en cours et x est un de ses descendants : $y \rightsquigarrow x \to y$ est un cycle, ce qui est impossible.

□

Remarque 30.72 Dans le vocabulaire de la classification des arêtes, les trois cas de cette preuve sont ceux d'une arête de liaison, d'une arête avant ou transverse, et d'une arête arrière : la preuve revient à dire qu'un graphe acyclique n'a pas d'arête arrière et que seules ces arêtes vérifient $t_s(x) < t_s(y)$. L'horloge utilisée ici ne compte que les sorties alors que celle des temps d'entrée et de sortie compte aussi les entrées, mais l'ordre des temps de sortie est le même et c'est le seul qui intervient. Enfin, on y a vu qu'on peut lancer les parcours depuis tous les sommets, dans n'importe quel ordre, sans calculer les degrés entrants.

Pour programmer le tri topologique, plutôt que de calculer les temps de sortie puis de trier, on ajoute chaque sommet à une liste au moment où il devient traité : cette liste est dans l'ordre croissant des temps de sortie et il suffit de la renverser. En OCaml, on ajoute en tête et on obtient directement le bon ordre ; en C, on remplit le tableau en partant de la fin.

Python

```python
INCONNU = 0
DECOUVERT = 1
TRAITE = 2

def parcours(ladj, x, etat, ordre):
    for y in ladj[x]:
        if etat[y] == INCONNU:
            etat[y] = DECOUVERT
            parcours(ladj, y, etat, ordre)
    etat[x] = TRAITE
    ordre.append(x) # ordre croissant des temps de sortie

def tri_topologique(ladj):
    etat = [ INCONNU ] * len(ladj)
    ordre = []
    for x in range(len(ladj)):
        if etat[x] == INCONNU:
            etat[x] = DECOUVERT
            parcours(ladj, x, etat, ordre)
    ordre.reverse() # ordre décroissant des temps de sortie
    return ordre
```

OCaml

```
type etat = Inconnu | Decouvert | Traite

let rec parcours ladj x etat ordre =
    List.iter (fun y ->
        if etat.(y) = Inconnu
        then begin
            etat.(y) <- Decouvert;
            parcours ladj y etat ordre
        end
    ) ladj.(x);
    etat.(x) <- Traite;
    (* x est traité : on le place devant les sommets déjà traités *)
    ordre := x :: !ordre

let tri_topologique ladj =
    let n = Array.length ladj in
    let etat = Array.make n Inconnu in
    let ordre = ref [] in
    for x = 0 to n - 1 do
        if etat.(x) = Inconnu
        then begin
            etat.(x) <- Decouvert;
            parcours ladj x etat ordre
        end
    done;
    !ordre
```

C

```c
#include <stdlib.h>

enum etat { INCONNU, DECOUVERT, TRAITE };

void parcours(graphe g, int x, int *etat, int *ordre, int *k) {
    for (int i = 0; i < g.degre[x]; i++) {
        int y = g.voisins[x][i];
        if (etat[y] == INCONNU) {
            etat[y] = DECOUVERT;
            parcours(g, y, etat, ordre, k);
        }
    }
    etat[x] = TRAITE;
    // x est traité : on le place juste avant les sommets déjà traités
    *k -= 1;
```

```
    ordre[*k] = x;
}

// Remplit ordre (de taille n) avec un tri topologique de g supposé acyclique
void tri_topologique(graphe g, int *ordre) {
    int *etat = malloc(g.n * sizeof(int));
    for (int i = 0; i < g.n; i++) {
        etat[i] = INCONNU;
    }
    int k = g.n;
    for (int x = 0; x < g.n; x++) {
        if (etat[x] == INCONNU) {
            etat[x] = DECOUVERT;
            parcours(g, x, etat, ordre, &k);
        }
    }
    free(etat);
}
```

30.6 Plus courts chemins

30.6.1 Graphes pondérés et définition du problème

Définition 30.73

Un graphe pondéré est la donnée d'un graphe, orienté ou non, $G = (S, A)$ et d'une fonction de pondération $\pi : A \to \mathbb{R}$ qui associe à chaque arête un nombre réel, son poids.

Remarque 30.74 En informatique pratique, $\mathbb{R}$ n'existe pas vraiment. Les poids seront donc le plus souvent des entiers ou des flottants. Cependant, les problèmes possiblement rencontrés par des poids réels sont pertinents et le cadre unifié rend les définitions plus simples.

Définition 30.75

On définit le poids d'un chemin φ comme la somme des poids de ses arêtes.

Exemple 30.76

Dans le graphe suivant :

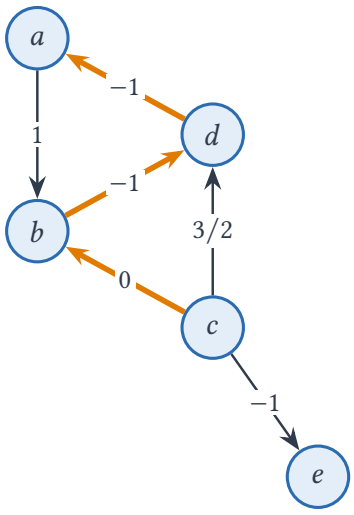


Figure 30.24 — Graphe pondéré ; en orange le chemin $c \to b \to d \to a$, de poids -2.

```
le chemin $\varphi = c \rightarrow b \rightarrow d \rightarrow a$ est de poids $\pi(\varphi) = 0  -
1 -1 = -2$.

On remarque que $b \rightarrow d \rightarrow a \rightarrow b$ est un cycle
de poids $-1$, on peut donc l'emprunter autant de fois qu'on le souhaite sur un
chemin passant par $a,b$ ou $d$ et faire diminuer son poids autant qu'on
le souhaite. On parle de cycle de poids négatif et leur présence est
problématique.
```

Théorème 30.77

Si G est un graphe pondéré **sans cycles de poids négatif**, φ est un chemin et ψ est un sous-chemin de φ obtenu en enlevant un cycle, alors $\pi(\varphi) \geq \pi(\psi)$.

Preuve Si $\varphi = \varphi_1 c \varphi_2$ où c est un cycle et $\psi = \varphi_1\varphi_2$, alors $\pi(\varphi) = \pi(\psi) + \pi(c) \geq \pi(\psi)$ car $\pi(c) \geq 0$.

□

Du théorème précédent, on en déduit que pour obtenir un chemin de poids minimum, on peut supposer que le chemin est élémentaire. En effet, si un chemin $(s_0, ..., s_p)$ d'un graphe orienté n'est pas élémentaire, on choisit $i < j$ avec $s_i = s_j$ et $j - i$ minimal (et $(i, j) \neq (0, p)$) : la portion $(s_i, ..., s_j)$ est alors un cycle (ses sommets sont distincts, sauf les extrémités, et elle est de longueur au moins 2 car il n'y a pas de boucle). On l'enlève, ce qui ne fait pas augmenter le poids, et on recommence ; la longueur diminuant strictement, ce procédé termine sur un chemin élémentaire. Or, comme il existe un nombre fini de chemins élémentaires entre deux sommets, on peut en déduire la définition suivante :

Remarque 30.78 Pour un graphe non orienté, on raisonne sur le graphe orienté symétrique associé. Une arête $\{x, y\}$ de poids négatif y donne le cycle $x \to y \to x$ de poids négatif : on peut faire des allers-retours sur cette arête pour faire diminuer le poids d'un chemin autant qu'on le souhaite. Pour un graphe non orienté, l'hypothèse « sans cycle de poids négatif » impose donc en pratique des poids positifs.

Définition 30.79

Soit $G = (S, A, \pi)$ un graphe pondéré **sans cycles de poids négatifs** et $x, y \in S$ tels que $x \rightsquigarrow y$, on appelle plus court chemin de x à y, un chemin $\varphi : x \rightsquigarrow y$ tel que

$$\pi(\varphi) = \min\{\pi(\psi) \mid \psi : x \rightsquigarrow y\}$$

Il est possible de travailler avec la complétion d'un graphe pondéré en considérant toutes les arêtes possibles mais en spécifiant que si $x \to y$ n'est pas une arête du graphe initial, alors $\pi(x \to y) = \infty$. On peut alors poser pour tout couple $(x, y) \in S^2$, $dist(x, y) = \min\{\pi(\psi) \mid \psi : x \rightsquigarrow y\}$ sachant que cette distance vaut ∞ lorsque y n'est pas accessible depuis x.

On va s'intéresser à deux problèmes :

Problème 30.80 — : PlusCourtCheminSource

Entrées :

- Un graphe pondéré $G = (S, A, \pi)$
- Un sommet source $s \in S$

Sortie :

- La donnée pour chaque $x \in S$, de la valeur $dist(s, x)$.
- Éventuellement, un chemin $\varphi : s \rightsquigarrow x$ réalisant cette distance.

Problème 30.81 — : PlusCourtCheminToutCouple

Entrées :

- Un graphe pondéré $G = (S, A, \pi)$

Sortie :

- La donnée pour chaque couple $x, y \in S$, de la valeur $dist(x, y)$.
- Éventuellement, un chemin $\varphi : x \rightsquigarrow y$ réalisant cette distance.

Le premier problème est naturellement inclus dans le second.

30.6.2 Cas des poids rationnels

Il se trouve qu'on a déjà trouvé un algorithme permettant de résoudre le problème quand $\forall e \in A, \pi(e) = 1$: le **parcours en largeur**.

Si les poids sont dans $\mathbb{Q}_+^*$, on peut se ramener à des poids dans $\mathbb{N}^*$ en multipliant chaque poids par le ppcm des dénominateurs des poids. Cela ne change pas la relation d'ordre entre les poids de chemins, et donc les plus courts chemins sont les mêmes.

Si $\pi(x \to y) = k > 1$, on peut rajouter $k - 1$ sommets artificiels $z_1, ..., z_{k-1}$ et des arêtes de poids unitaire $x \to z_1 \to z_2 \to ... \to z_{k-1} \to y$ à la place de l'arête $x \to y$. En procédant ainsi, on se ramène donc à un graphe dont les poids sont unitaires et on peut résoudre le problème avec un parcours en largeur.

Si on considère que la longueur des poids en mémoire fait partie de l'entrée, alors cette réduction est très coûteuse (le nombre de sommets ajoutés est de l'ordre de la somme des poids, donc exponentiel en la taille de leur écriture), mais si on considère que les poids sont bornés par une constante, alors toute cette réduction ne fait que grossir la constante du $O(|S| + |A|)$!

> **Remarque 30.82** On pourrait être tenté de supprimer les poids négatifs en ajoutant une constante à chaque poids, cependant, en faisant cela, on pénalise les chemins en fonction de leur longueur et on ne préserve pas la notion de plus court chemin.
>
> Considérons par exemple le graphe :

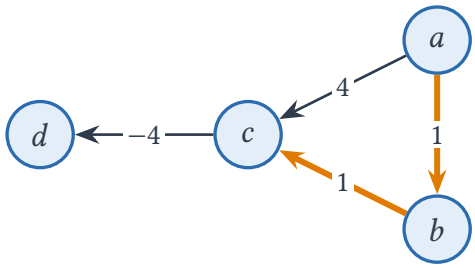


Figure 30.25 — Le plus court chemin de a à c (en orange) est $a \to b \to c$, de poids 2.

```
Le chemin le plus court entre $a$ et $c$ est de poids $2$ et est le chemin
$a \rightarrow b \rightarrow c$.

Si on ajoute 5 à chaque poids pour qu'ils soient $>0$, on obtient le graphe :
```

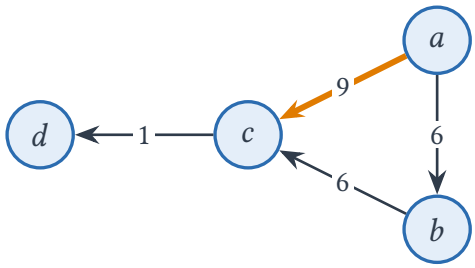


Figure 30.26 — Après ajout de 5 à chaque poids, le plus court chemin de a à c devient $a \to c$.

```
Et ici, le chemin le plus court devient $a \rightarrow c$ de poids $9$ et non
plus $a \rightarrow b \rightarrow c$ qui devient de poids $12$.

Plus généralement, si on ajoute $\eta$ à chaque poids, alors le poids de
$\varphi$ devient $\pi(\varphi) + \eta |\varphi|$ : l'ordre entre les poids de
deux chemins de longueurs différentes n'est pas préservé.
```

30.6.3 Algorithme de Dijkstra

On se concentre ici sur le problème `PlusCourtCheminSource` dans le cas de poids positifs.

Si $G = (S, A, \pi)$ est un graphe pondéré et $s \in S$, on considère un étiquetage $d : S \to \mathbb{R} \cup \{\infty\}$ qu'on va faire évoluer dans un algorithme et tel que $d(x)$ représente le plus petit poids trouvé jusqu'ici d'un chemin entre s et x. En l'absence de cycle négatif, on sait qu'un tel chemin peut être supposé élémentaire.

Au départ, on pose $d(s) = 0$ et $\forall x \in S \setminus \{s\}, d(x) = \infty$.

L'idée est d'emprunter les raccourcis, c'est-à-dire les arêtes $y \to x$ telles que $d(y) + \pi(y \to x) < d(x)$:

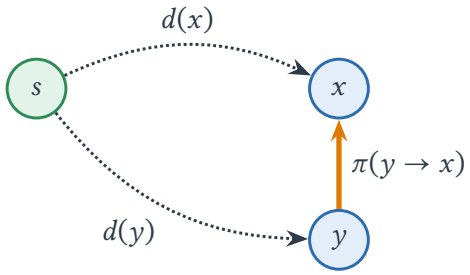


Figure 30.27 — Raccourci : on emprunte $y \to x$ si $d(y) + \pi(y \to x) < d(x)$.

Algorithme 30.83 — : Dijkstra

Entrées :

- Un graphe pondéré $G = (S, A, \pi)$ où $\pi : A \to \mathbb{R}_+$
- Un sommet $s \in S$

Déroulement :

- On crée un tableau `d` avec $d(s) = 0$ et $d(x) = \infty$ pour $x \neq s$.
- On crée un tableau `a_traiter` et on ajoute `s`
- Les sommets sont initialement non marqués
- Tant que `a_traiter` est non vide
  - On retire le sommet `x` de `a_traiter` de valeur $d(x)$ **minimale**
  - Si `x` est non marqué
    - On marque `x`
    - Pour chaque arête $x \to y$ où $d(x) + \pi(x \to y) < d(y)$
      - On pose $d(y) = d(x) + \pi(x \to y)$
      - On ajoute y dans `a_traiter`

Exemple 30.84

On considère le graphe non orienté suivant (vu comme un graphe orienté symétrique)

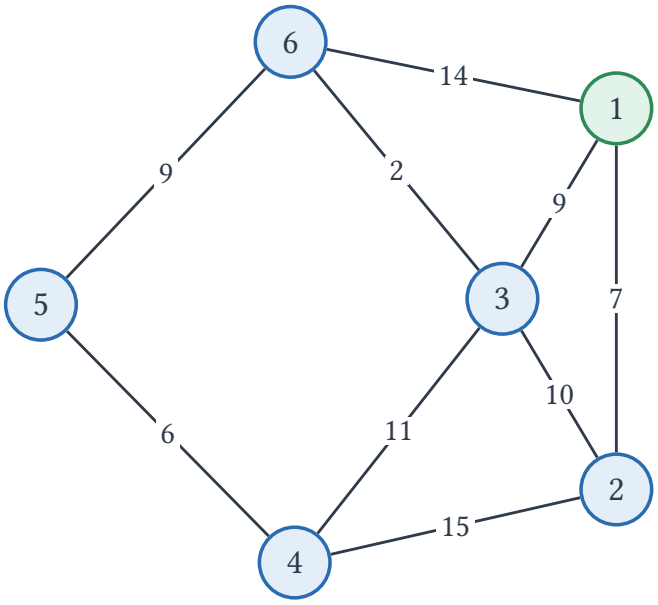


Figure 30.28 — Graphe pondéré non orienté, de source $s = 1$

```
pour $s = 1$ où les sommets dans `a_traiter` sont
indiqués en bleu, les sommets marqués sont indiqués en vert (en orange
celui qui vient d'être retiré), les sommets non atteints en gris
et où la valeur de $d$ est donnée dans le cadre à côté du
sommet :
```

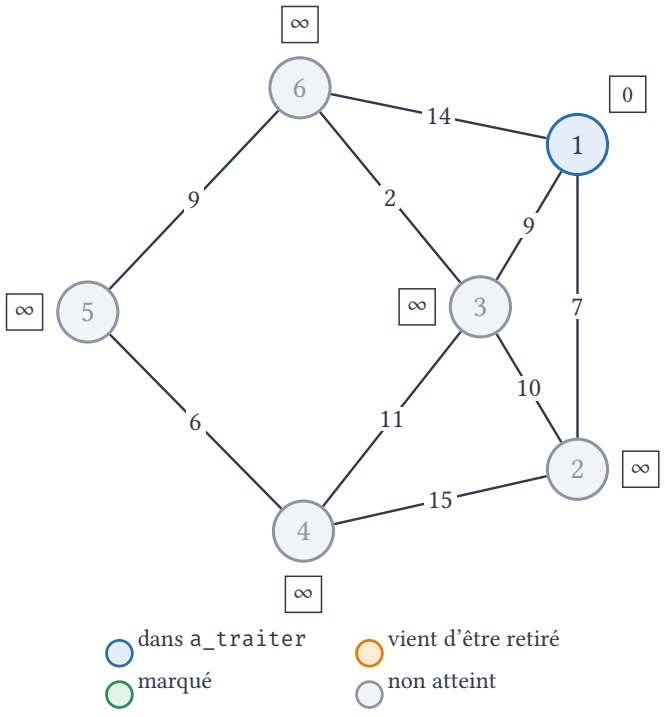
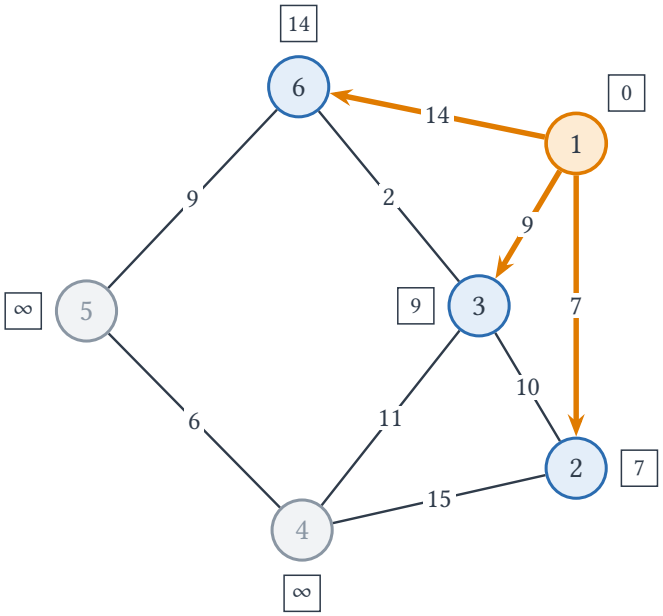
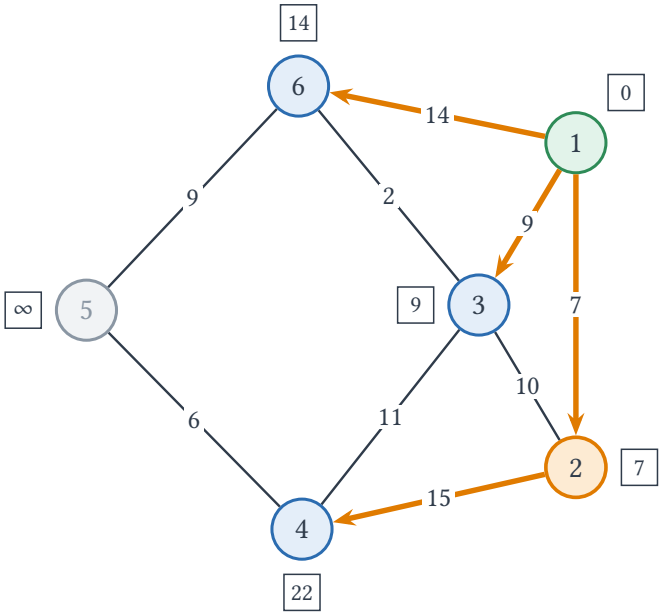


Figure 30.29 — Initialisation : $d(1) = 0$, $d(x) = \infty$ ailleurs.

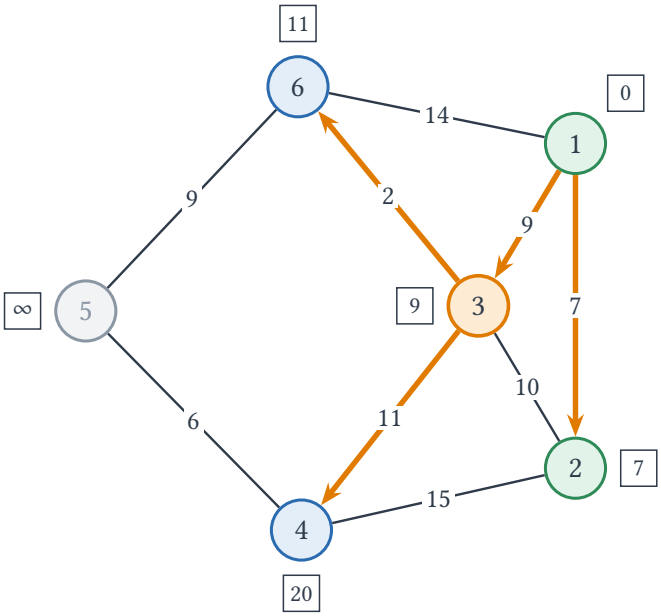
```
On commence par retirer 1 qui est seul dans `a_traiter` avec $d(1) = 0$. On peut
alors ajouter les sommets $2, 3$ et $6$ :
```



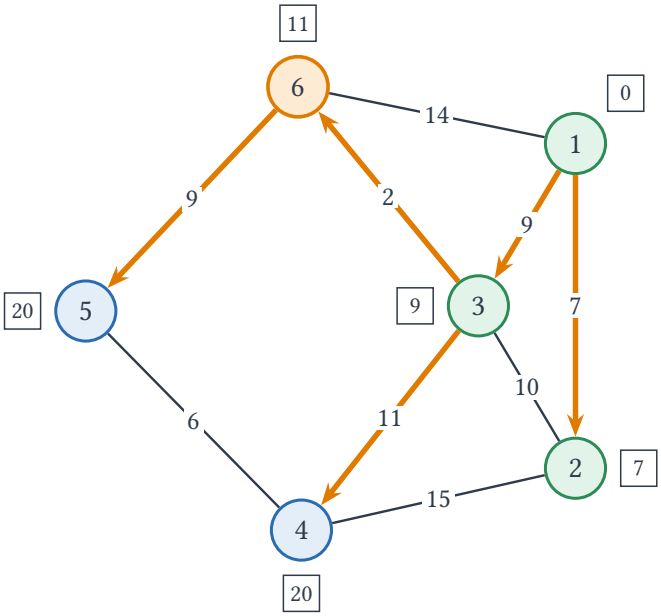
On retire alors le sommet 2 qui a la plus petite valeur de d. L'arête $2 \rightarrow 4$ vérifie la condition mais pas l'arête $2 \rightarrow 3$ (car $7 + 10 \ge 9$).
On ajoute donc 4 dans les sommets à traiter :



On retire alors 3 et on voit que les deux arêtes $3 \rightarrow 6$ et $3 \rightarrow 4$ permettent d'améliorer d ($9 + 2 < 14$ et $9 + 11 < 22$). On rajoute donc 4 et 6 dans `a_traiter` :



On retire alors le sommet 6 qui permet d'emprunter $6 \rightarrow 5$ et qui rajoute ainsi 5 à `a_traiter` :



On traite les deux derniers sommets, 4 et 5 (tous deux avec $d = 20$), en les retirant : aucune arête ne vérifie plus la condition.

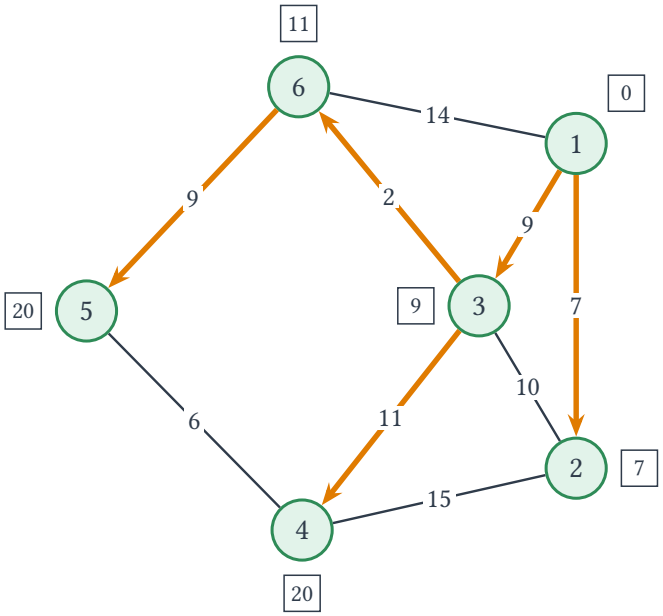


Figure 30.30 — État final : les valeurs de d sont les distances depuis 1.

Cet algorithme est essentiellement un parcours quelconque où la stratégie de sortie du sac est de retirer le sommet de plus petite valeur de d (un successeur y d'un sommet marqué qui n'est pas ajouté vérifie $d(y) < \infty$, il a donc déjà été ajouté). On sait donc déjà qu'il termine et qu'il explore $acc(s) = \{x \in S \mid s \rightsquigarrow x\}$.

Comme on ne modifie d que pour mettre des nombres $\neq \infty$, on est assuré qu'à la fin de l'algorithme les sommets dans $acc(s)$ ont une valeur dans $\mathbb{R}_+$ dans le tableau d et que les autres ont ∞. La force de l'algorithme de Dijkstra découle du contenu final de d qui répond à la question voulue :

Théorème 30.85

À la fin de l'algorithme, si $s \rightsquigarrow x$, $d(x) = dist(s, x)$.

Preuve Une récurrence immédiate montre qu'à tout moment, si $d(x) < \infty$, alors $d(x)$ est le poids d'un chemin de s à x, donc $d(x) \geq dist(s, x)$.

Montrons que lorsqu'un sommet x est marqué, on a $d(x) = dist(s, x)$. Comme $d(x)$ ne peut ensuite que diminuer et reste minoré par $dist(s, x)$, cette valeur ne changera plus, ce qui prouve le théorème puisque tous les sommets accessibles sont marqués.

Le premier sommet marqué est s avec $d(s) = 0 = dist(s, s)$ (les poids étant positifs). Supposons par l'absurde qu'un sommet x soit marqué avec $d(x) > dist(s, x)$, et considérons le premier tel sommet. Soit φ un plus court chemin de s à x. Comme s est marqué et que x ne l'est pas encore au moment où on le retire, il existe une première arête $u \to v$ de φ avec u marqué et v non marqué. Lors du marquage de u, on avait $d(u) = dist(s, u)$ par minimalité de x, et l'arête $u \to v$ a été examinée : on a donc ensuite $d(v) \leq dist(s, u) + \pi(u \to v) = dist(s, v)$, la portion de φ allant de s à v étant un plus court chemin. De plus, v est alors dans `a_traiter` : il y a été ajouté lors de la dernière diminution de $d(v)$ et n'en a pas été retiré puisqu'il n'est pas marqué.

Comme les poids sont positifs, $dist(s, v) \leq dist(s, x)$, donc $d(v) \leq dist(s, x) < d(x)$. Ceci contredit le choix de x comme sommet de `a_traiter` de valeur d minimale (si $v = x$, c'est directement $d(x) \leq dist(s, x)$ qui est contredit).

C'est la positivité des poids qui permet d'affirmer $dist(s, v) \leq dist(s, x)$: avec des poids négatifs, ce raisonnement tombe en défaut.

□

On peut implémenter directement cet algorithme :

OCaml

```
let rec extrait_plus_petit l d =
    match l with
    | [] -> failwith "impossible"
    | [x] -> (x, [])
    | x::q ->
        let (y, q') = extrait_plus_petit q d in
        if d.(y) < d.(x)
        then (y, x :: q')
        else (x, q)

let dijkstra g src =
    let n = Array.length g in
    let inf = max_int in
    let d = Array.make n inf in
    d.(src) <- 0;
    let marques = Array.make n false in
    let a_traiter = ref [ src ] in
    while !a_traiter <> [] do
        let x, q = extrait_plus_petit !a_traiter d in
        a_traiter := q;
        if not marques.(x)
        then begin
            marques.(x) <- true;
            List.iter (fun (y, poids) ->
                let n_d = d.(x) + poids in
                if n_d < d.(y)
                then begin
                    d.(y) <- n_d;
                    a_traiter := y :: !a_traiter
                end
            ) g.(x)
        end
    done;
    d
```

Python

```
def extrait_plus_petit(t, d):
    x = t[0] # recherche du d[x] min dans t
    for i in t[1:]:
        if d[i] < d[x]:
            x = i
    t.remove(x) # on le retire
    return x # on le renvoie

def dijkstra(g, src):
    n = len(g)
    inf = float('inf')
    d = [ inf ] * n
    d[src] = 0
    a_traiter = [ src ]
    marques = [ False ] * n
    while a_traiter != []:
        x = extrait_plus_petit(a_traiter, d)
        if not marques[x]:
            marques[x] = True
            for y, poids in g[x]:
                n_d = d[x] + poids
                if n_d < d[y]:
                    d[y] = n_d
                    a_traiter.append(y)
    return d # distances finales
```

C

```c
#include <stdlib.h>
#include <stdbool.h>
#include <limits.h>
#include <assert.h>

#define INF INT_MAX

struct arete {
    int but;
    int poids;
};
typedef struct arete arete;

struct graphe_pondere {
    int n;
    int *degre;
    arete **voisins; // voisins[x][i] : i-ème arête issue de x
};
typedef struct graphe_pondere graphe_pondere;

// Retire de a_traiter (de taille *taille) un sommet x
// de d[x] minimal et le renvoie
int extrait_plus_petit(int *a_traiter, int *taille, int *d) {
    int i_min = 0; // recherche du d[x] min dans a_traiter
    for (int i = 1; i < *taille; i++) {
        if (d[a_traiter[i]] < d[a_traiter[i_min]]) {
            i_min = i;
        }
    }
    int x = a_traiter[i_min];
    // on le retire en le remplaçant par le dernier élément
    *taille -= 1;
    a_traiter[i_min] = a_traiter[*taille];
    return x;
}

// Renvoie le tableau des distances (à libérer par l'appelant)
int *dijkstra(graphe_pondere g, int src) {
    int *d = malloc(g.n * sizeof(int));
    bool *marques = malloc(g.n * sizeof(bool));
    int nb_aretes = 0;
    for (int x = 0; x < g.n; x++) {
        d[x] = INF;
        marques[x] = false;
        nb_aretes += g.degre[x];
    }
    d[src] = 0;
    // au plus |A| + 1 ajouts dans a_traiter
    int capacite = nb_aretes + 1;
    int *a_traiter = malloc(capacite * sizeof(int));
    int taille = 0;
    a_traiter[taille++] = src;
    while (taille > 0) {
        int x = extrait_plus_petit(a_traiter, &taille, d);
        if (!marques[x]) {
            marques[x] = true;
            for (int i = 0; i < g.degre[x]; i++) {
                int y = g.voisins[x][i].but;
                int n_d = d[x] + g.voisins[x][i].poids;
                if (n_d < d[y]) {
                    d[y] = n_d;
                    assert(taille < capacite);
                    a_traiter[taille++] = y;
                }
            }
        }
    }
    free(a_traiter);
    free(marques);
```

```
    return d; // distances finales
}
```

Dans les trois versions, le graphe est donné par listes d'adjacence où chaque voisin est accompagné du poids de l'arête, et `a_traiter` est une simple liste (un tableau en C) dans laquelle on cherche linéairement le minimum. En C, un sommet peut figurer plusieurs fois dans `a_traiter`, comme dans les deux autres versions : on dimensionne donc le tableau avec le nombre maximal d'ajouts $|A|+1$ justifié dans la remarque suivante. Les poids étant des entiers, on représente ∞ par `INT_MAX` (et par `max_int` en OCaml) : on ne lui ajoute jamais rien car seuls des sommets marqués, donc de distance finie, propagent leur valeur.

Remarque 30.86 La complexité de la recherche du sommet de valeur d minimale est linéaire en la taille de `a_traiter`. Chaque ajout dans `a_traiter` correspond à une arête $x \to y$ examinée depuis un sommet x marqué, donc il y a au plus $|A|+1$ ajouts, et autant de retraits, dans une liste de taille au plus $|A|+1$: on a une complexité totale en $O(|S|+|A|^2)$. Si on n'ajoute y que lorsqu'il n'est pas déjà dans `a_traiter`, cette liste est de taille au plus $|S|$ et on obtient $O(|S|\times|A|)$ (on peut même obtenir $O(|S|^2)$ en cherchant le minimum directement dans le tableau `d` parmi les sommets non marqués).

Utilisation d'une file de priorité

On peut améliorer l'algorithme précédent avec une **file de priorité** où la priorité de x est $d(x)$ et où on suppose qu'on extrait la priorité minimale. Cela permet ainsi d'obtenir une complexité en $O(|S|+|A|\log|S|)$.

Cependant, il est alors nécessaire de modifier la priorité ce qui est une opération assez pénible. À la place, on peut ajouter une autre fois le sommet. Au pire, on fera grossir la file (qui reste de longueur $O(|A|)$) et quand on retirera un sommet déjà traité, celui-ci sera ignoré. Comme $\log|A| \leq \log|S|^2 = 2\log|S|$, la complexité reste en $O(|S|+|A|\log|S|)$.

On obtient ainsi l'algorithme suivant :

Algorithme 30.87 — : Dijkstra

Entrées :

- Un graphe pondéré $G=(S,A,\pi)$ où $\pi : A \to \mathbb{R}_+$
- Un sommet $s \in S$

Déroulement :

- On crée un tableau `d` avec $d(s)=0$ et $d(x)=\infty$ pour $x \neq s$.
- On crée une file de min-priorité `a_traiter` et on ajoute `s` avec priorité `0`
- Les sommets sont initialement non marqués
- Tant que `a_traiter` est non vide
  - On retire le sommet `x` de `a_traiter`
  - Si `x` est non marqué
    - On marque `x`
    - Pour chaque arête $x \to y$ où $d(x)+\pi(x\to y) < d(y)$
      - On pose $d(y) = d(x)+\pi(x\to y)$
      - On insère y avec priorité $d(y)$

30.6.4 Relaxation

Pour éclairer la correction de Dijkstra et le lien avec la positivité des poids des arêtes, on va présenter ici un cadre plus général.

Arêtes tendues

Définition 30.88

On dit qu'une arête $x \to y$ est **tendue** si

$$d(y) > d(x) + \pi(x \to y)$$

Quand on pose ensuite $d(y) = d(x) + \pi(x \to y)$ on dit qu'on **relâche l'arête**.

Ainsi, une arête est tendue quand on peut l'emprunter pour améliorer l'estimation du plus petit chemin de s à y.

Théorème 30.89

On suppose que G est sans cycle de poids négatif, que $d(s) = 0$ et que pour tout $x \in S$, $d(x)$ vaut ∞ ou le poids d'un chemin de s à x (c'est le cas initialement et cela reste vrai quand on relâche une arête). Si aucune arête n'est tendue, alors $d(x) = dist(s, x)$ pour tout $x \in S$ (avec $= \infty$ si x est inaccessible).

Preuve Par hypothèse, $d(x) \geq dist(s, x)$ pour tout x, en particulier $d(x) = \infty = dist(s, x)$ si x est inaccessible. Il reste à montrer que $d(x) \leq dist(s, x)$ pour x accessible.

Pour un tel x, on note $k(x)$ le nombre minimal d'arêtes d'un plus court chemin de s à x, et on raisonne par récurrence forte sur $k(x)$.

Si $k(x) = 0$, alors $x = s$ et $d(s) = 0 = dist(s, s)$ (il n'y a pas de cycle de poids négatif).

Si $k(x) \geq 1$, soit φ un plus court chemin de s à x ayant $k(x)$ arêtes et y le prédécesseur de x dans ce chemin : on a $\varphi = \psi\,(y \to x)$ où $\psi : s \rightsquigarrow y$. Nécessairement, le chemin ψ est un plus court chemin de s à y, sinon on obtiendrait un chemin vers x de poids strictement plus petit que celui de φ. Ainsi $k(y) \leq k(x) - 1$ et, par hypothèse de récurrence, $d(y) \leq dist(s, y) = \pi(\psi)$.

Comme l'arête $y \to x$ n'est pas tendue, on a alors $d(x) \leq d(y) + \pi(y \to x) \leq \pi(\psi) + \pi(y \to x) = \pi(\varphi) = dist(s, x)$.

□

On en déduit un pseudo-algorithme appelé *méthode de relaxation* :

Algorithme 30.90 — : Relaxation

Entrées :

- Un graphe pondéré $G = (S, A, \pi)$ sans cycle de poids négatif
- Un sommet $s \in S$

Déroulement :

- On pose d tel que $d(s) = 0$ et $\forall x \in S \setminus \{s\}, d(x) = \infty$.
- Tant qu'il existe une arête e tendue
  - On relâche l'arête e

Remarque 30.91

- On montre par récurrence sur le nombre d'itérations que si $d(x) < \infty$ alors $d(x)$ est le poids d'un chemin simple de s à x. Comme ces chemins simples sont en nombre fini, leurs poids aussi et donc d ne peut prendre qu'un nombre fini de valeurs. Comme d diminue strictement en un point à chaque itération, la boucle termine.
- En présence d'un cycle de poids négatif accessible depuis s, on pourrait continuellement relâcher les arêtes du cycle et donc l'algorithme ne termine pas.
- En sortie de l'algorithme on a donc $dist(s, x) \leq dist(s, y) + \pi(y \to x)$ pour toute arête $y \to x$. Or, on a l'égalité pour l'avant-dernier sommet y d'un plus court chemin. On en déduit donc l'égalité **très importante** suivante :

$$\forall x \in S \setminus \{s\}, dist(s, x) = \min\{dist(s, y) + \pi(y \to x) \mid y \in S\}$$

En fait, si on considère l'opération Φ définie par $\Phi(d)(s) = 0$ et :

$$\forall x \in S \setminus \{s\}, \Phi(d)(x) = \min\{d(y) + \pi(y \to x) \mid y \in S\}$$

on remarque que $x \mapsto dist(s, x)$ est un point fixe de Φ.

On pourrait ainsi le calculer comme la limite de la suite stationnaire $d_{n+1} = \Phi(d_n)$ où $d_0(s) = 0$ et $\forall x \neq s, d_0(x) = \infty$.

Si on considère le graphe

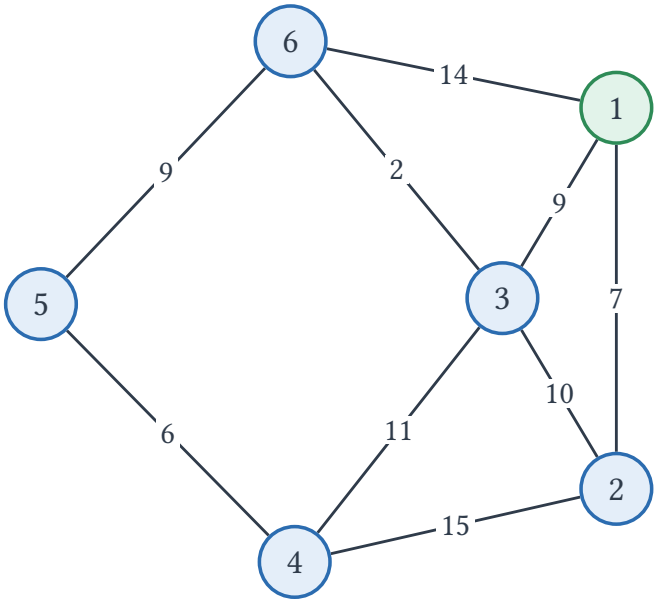


Figure 30.31 — Graphe pondéré non orienté, de source $s = 1$

```
et que l'on représente la suite $d$ sous une forme de tableau, on
obtient (pour $s = 1$) :


::: {.mtable cols="7"}
::: cellule
$n$
:::
::: cellule
$d_n(1)$
:::
::: cellule
$d_n(2)$
:::
::: cellule
$d_n(3)$
:::
::: cellule
$d_n(4)$
:::
::: cellule
$d_n(5)$
:::
::: cellule
$d_n(6)$
:::
::: cellule
$0$
:::
::: cellule
$0$
```

```
:::
::: cellule
$\infty$
:::
::: cellule
$\infty$
:::
::: cellule
$\infty$
:::
::: cellule
$\infty$
:::
::: cellule
$\infty$
:::
::: cellule
$1$
:::
::: cellule
$0$
:::
::: cellule
$7$
:::
::: cellule
$9$
:::
::: cellule
$\infty$
:::
::: cellule
$\infty$
:::
::: cellule
$14$
:::
::: cellule
$2$
:::
::: cellule
$0$
:::
::: cellule
$7$
:::
::: cellule
$9$
:::
::: cellule
$20$
:::
::: cellule
$23$
:::
::: cellule
$11$
:::
::: cellule
$3$
:::
::: cellule
$0$
:::
::: cellule
$7$
:::
::: cellule
$9$
```

```
:::
::: cellule
$20$
:::
::: cellule
$20$
:::
::: cellule
$11$
:::
::: cellule
$4$
:::
::: cellule
$0$
:::
::: cellule
$7$
:::
::: cellule
$9$
:::
::: cellule
$20$
:::
::: cellule
$20$
:::
::: cellule
$11$
:::
::: cellule
$\vdots$
:::
::: cellule
\
:::
::: cellule
\
:::
::: cellule
\
:::
::: cellule
\
:::
::: cellule
\
:::
::: cellule
\
:::
:::


On peut montrer par récurrence sur $n$ que, pour $x \neq s$, $d_n(x)$ est le
plus petit poids d'un chemin de $s$ à $x$ ayant au plus $n$ arêtes. En
l'absence de cycles négatifs, un plus court chemin peut être supposé
élémentaire, donc avoir au plus $|S|-1$ arêtes : la suite est stationnaire
à partir du rang $|S|-1$ au plus. L'algorithme obtenu s'appelle
l'algorithme de **Bellman-Ford**.
```

✍ On en déduit une implémentation directe de l'algorithme de Bellman-Ford, avec la même représentation des graphes pondérés que pour Dijkstra. On calcule d_k à partir de d_{k-1} pour k allant de 1 à $|S| - 1$, puis on vérifie qu'aucune arête n'est encore tendue.

✍ **Python**

```python
def bellman_ford(g, src):
    # renvoie None si un cycle de poids négatif est accessible depuis src
    n = len(g)
    inf = float('inf')
    d = [ inf ] * n
    d[src] = 0
    for k in range(1, n):
        n_d = d.copy() # calcul de d_k à partir de d_{k-1}
        for x in range(n):
            for y, poids in g[x]:
                if d[x] + poids < n_d[y]:
                    n_d[y] = d[x] + poids
        d = n_d
    # une arête encore tendue révèle un cycle de poids négatif
    for x in range(n):
        for y, poids in g[x]:
            if d[x] + poids < d[y]:
                return None
    return d
```

OCaml

```
(* Renvoie Some d, ou None si un cycle de poids négatif
   est accessible depuis src *)
let bellman_ford g src =
    let n = Array.length g in
    let inf = max_int in
    let d = Array.make n inf in
    d.(src) <- 0;
    for _ = 1 to n - 1 do
        (* calcul de d_k à partir de d_{k-1}, recopié ensuite dans d *)
        let n_d = Array.copy d in
        for x = 0 to n - 1 do
            if d.(x) <> inf
            then List.iter (fun (y, poids) ->
                if d.(x) + poids < n_d.(y)
                then n_d.(y) <- d.(x) + poids
            ) g.(x)
        done;
        Array.blit n_d 0 d 0 n
    done;
    (* une arête encore tendue révèle un cycle de poids négatif *)
    let tendue = ref false in
    for x = 0 to n - 1 do
        if d.(x) <> inf
        then List.iter (fun (y, poids) ->
            if d.(x) + poids < d.(y)
            then tendue := true
        ) g.(x)
    done;
    if !tendue then None else Some d
```

C

```c
#include <stdlib.h>
#include <limits.h>

#define INF INT_MAX

// Renvoie le tableau des distances (à libérer par l'appelant),
// ou NULL si un cycle de poids négatif est accessible depuis src
int *bellman_ford(graphe_pondere g, int src) {
    int *d = malloc(g.n * sizeof(int));
    int *n_d = malloc(g.n * sizeof(int));
    for (int x = 0; x < g.n; x++) {
        d[x] = INF;
    }
    d[src] = 0;
```

```
    for (int k = 1; k < g.n; k++) {
        // calcul de d_k à partir de d_{k-1} dans n_d
        for (int x = 0; x < g.n; x++) {
            n_d[x] = d[x];
        }
        for (int x = 0; x < g.n; x++) {
            if (d[x] != INF) {
                for (int i = 0; i < g.degre[x]; i++) {
                    int y = g.voisins[x][i].but;
                    if (d[x] + g.voisins[x][i].poids < n_d[y]) {
                        n_d[y] = d[x] + g.voisins[x][i].poids;
                    }
                }
            }
        }
        int *tmp = d;
        d = n_d;
        n_d = tmp;
    }
    free(n_d);
    // une arête encore tendue révèle un cycle de poids négatif
    for (int x = 0; x < g.n; x++) {
        if (d[x] != INF) {
            for (int i = 0; i < g.degre[x]; i++) {
                int y = g.voisins[x][i].but;
                if (d[x] + g.voisins[x][i].poids < d[y]) {
                    free(d);
                    return NULL;
                }
            }
        }
    }
    return d;
}
```

Remarque 30.92

- Le tableau `n_d` est initialisé avec d_{k-1} et pas avec des ∞ : en l'absence de cycle de poids négatif, cela ne change rien car $d_k(y) \leq d_{k-1}(y)$, et cela évite de traiter à part le sommet s.
- La complexité est en $O(|S| \times (|S| + |A|))$ en temps et $O(|S|)$ en espace.
- La vérification finale détecte exactement les cycles de poids négatif accessibles depuis s. S'il n'y en a pas, on peut se restreindre aux sommets accessibles depuis s : d contient les distances, qui vérifient $dist(s, y) \leq dist(s, x) + \pi(x \to y)$, et aucune arête n'est tendue. Réciproquement, si $c = v_0 \to v_1 \to \ldots \to v_p = v_0$ est un cycle accessible depuis s, ses sommets ont une valeur de d finie (ils sont accessibles par un chemin élémentaire, d'au plus $|S| - 1$ arêtes). Si aucune arête n'était tendue, on aurait $d(v_i) \leq d(v_{i-1}) + \pi(v_{i-1} \to v_i)$ pour tout $i \in [\![1, p]\!]$ et, en sommant, $0 \leq \pi(c)$: le cycle n'est pas de poids négatif.

Algorithme de Dijkstra comme relaxation

Quand on retire un sommet x de la file, si $e = x \to y$ alors soit l'arête est non tendue et alors $d(y) \leq d(x) + \pi(x \to y)$, ce qui signifie qu'on connaît déjà un chemin simple de poids $d(y)$ qui est au moins aussi bon que de passer par x. Cela signifie notamment que y a déjà été découvert à un moment par l'algorithme ($d(y) < \infty$). Soit l'arête est tendue et passer par x améliore le chemin vers y. On relâche l'arête. On a alors deux possibilités : soit y est dans la file, et on change sa priorité, soit on l'insère dans la file.

✍ Quand la file est vide, aucune arête n'est tendue : en effet, après le dernier retrait d'un sommet x, les arêtes issues de x ne sont plus tendues, et elles ne peuvent le redevenir que si $d(x)$ diminue, auquel cas x est remis dans la file. D'après le théorème précédent, d contient alors les distances.

En procédant ainsi, on obtient un algorithme qui implémente le pseudo-algorithme précédent :

Algorithme 30.93 — : ProtoDijkstra

Entrées :

- Un graphe pondéré $G = (S, A, \pi)$ sans cycle négatif
- Un sommet $s \in S$

Déroulement :

- On crée un tableau `d` avec $d(s) = 0$ et $d(x) = \infty$ pour $x \neq s$.
- On crée une file de min-priorité `a_traiter` et on ajoute `s` avec priorité `0`
- Tant que `a_traiter` est non vide
  - On retire le sommet `x` de `a_traiter`
  - Pour chaque arête $x \to y$ tendue vis-à-vis de d
    - On relâche l'arête dans d
    - Si y est dans `a_traiter`
      - Alors, on remplace sa priorité par la nouvelle valeur de $d(y)$
      - Sinon, on insère y avec priorité $d(y)$

Exemple 30.94

L'exemple suivant permet d'observer l'algorithme sur un graphe comportant des poids négatifs. On a indiqué en bleu les sommets sur la file, en orange le sommet actuellement traité, en vert les sommets déjà retirés et en gris les sommets non atteints. Les arêtes de parenté sont présentées comme dans l'exemple précédent à mesure qu'elles sont modifiées.

On va ainsi partir du graphe suivant et du sommet source a :

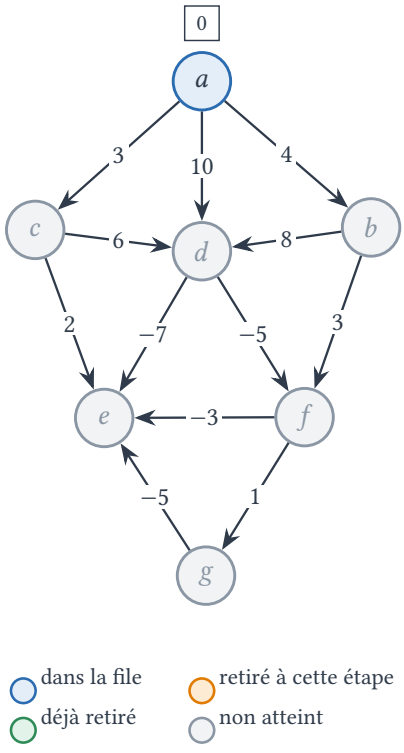
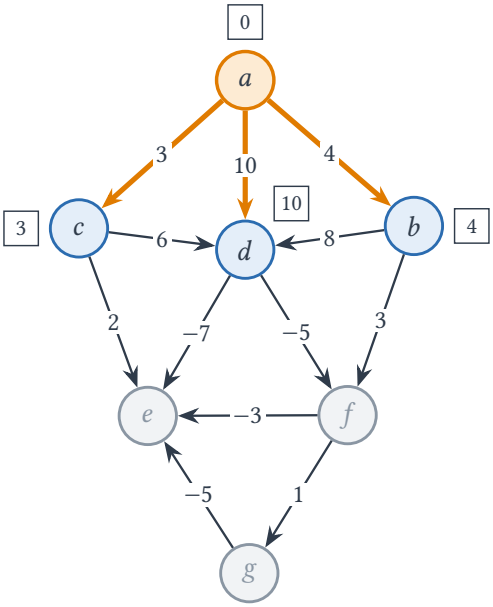
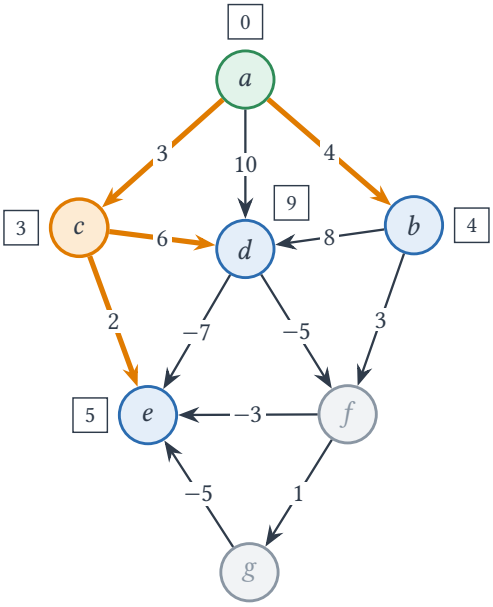
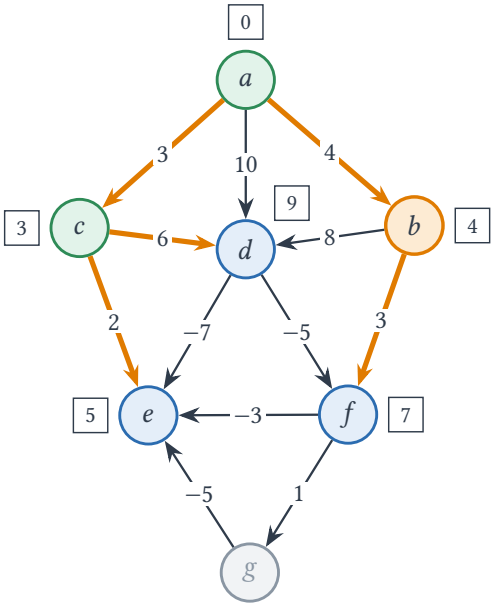
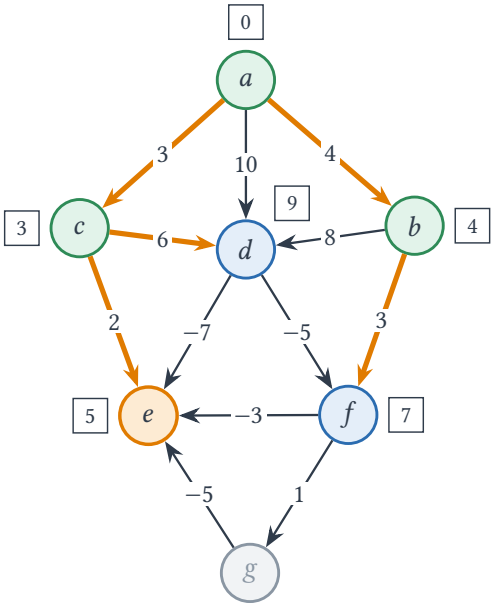


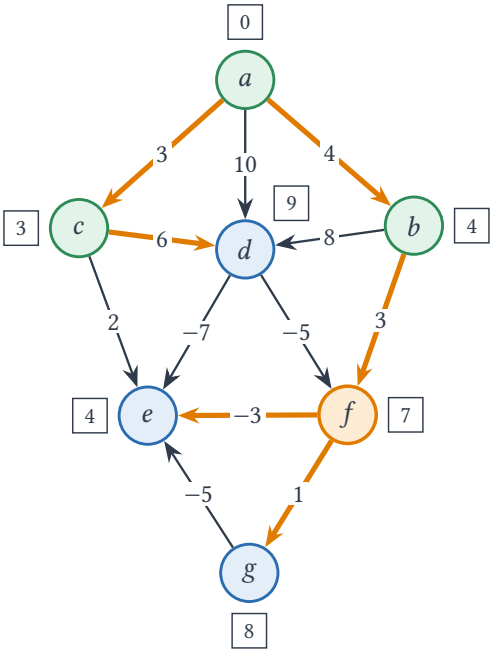
Figure 30.32 — Graphe à poids négatifs, source a.

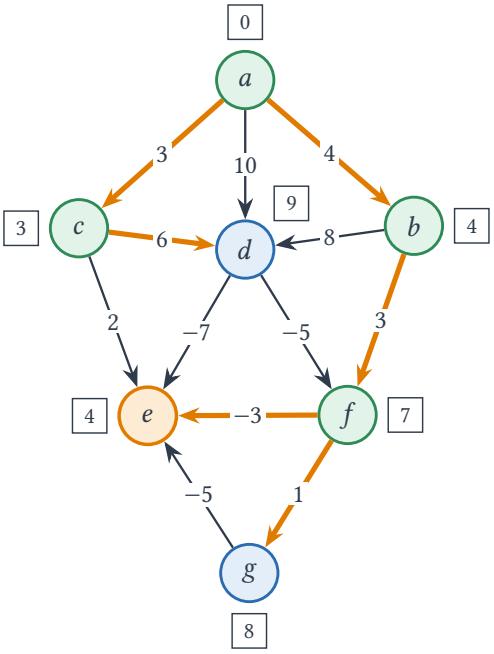

0
a
3
10
4
3
c
10
d
b
4
6
8
2
−7
−5
3
e
−3
f
−5
1
g

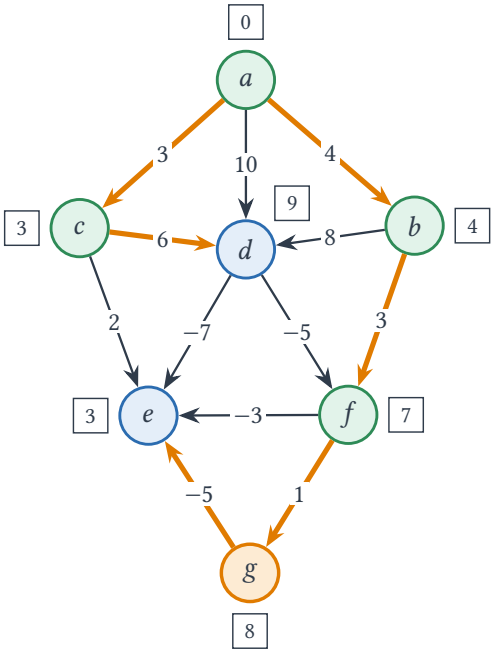

0
a
3
10
4
3
c
9
d
b
4
6
8
2
−7
−5
3
5
e
−3
f
−5
1
g

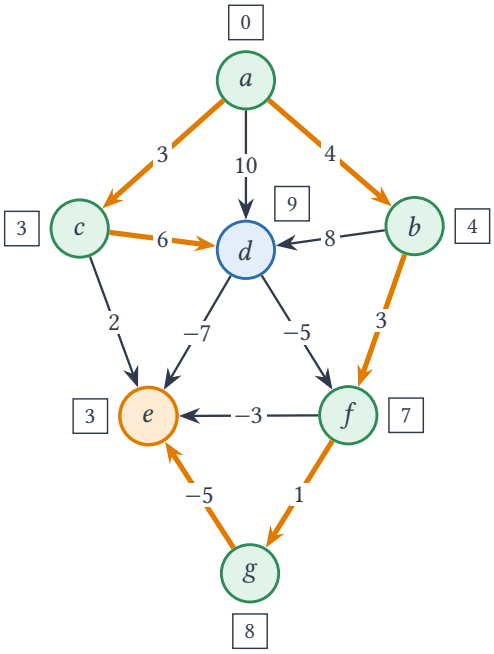

0
a
3
10
4
3
c
9
b
4
6
d
8
2
−7
−5
3
5
e
−3
f
7
−5
1
g

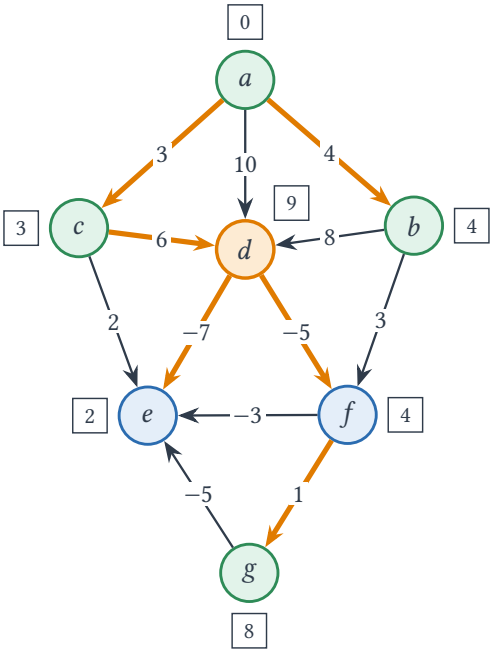

0
a
3
10
4
3
c
9
b
4
6
d
8
2
−7
−5
3
5
e
−3
f
7
−5
1
g

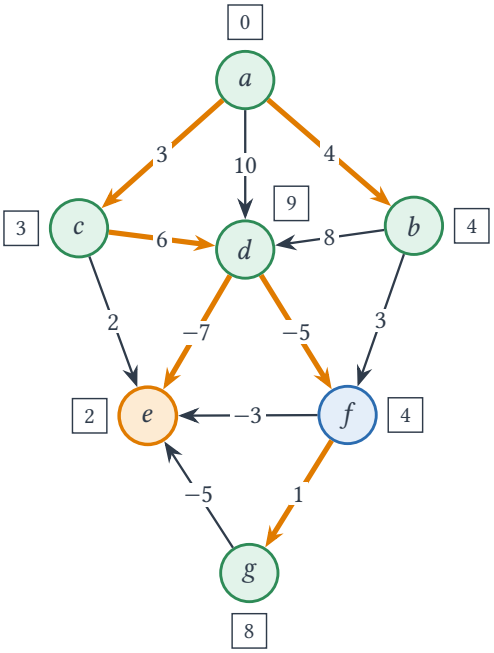

0
a
3
10
4
3
c
9
b
4
6
d
8
2
−7
−5
3
4
e
−3
f
7
−5
1
g
8

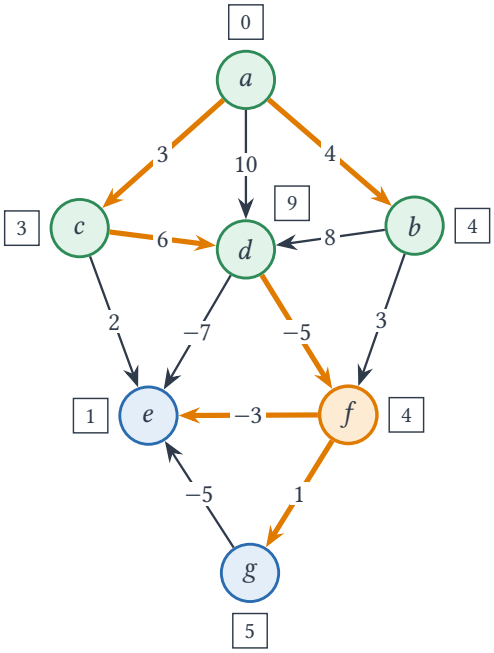

0
a
3
10
4
3
c
9
b
4
6
d
8
2
−7
−5
3
4
e
−3
f
7
−5
1
g
8

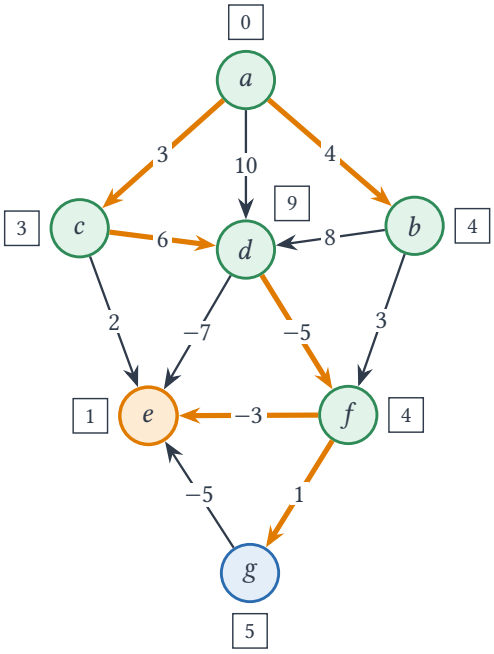

0
a
3
10
4
3
c
9
d
b
4
6
8
2
−7
−5
3
3
e
−3
f
7
−5
1
g
8

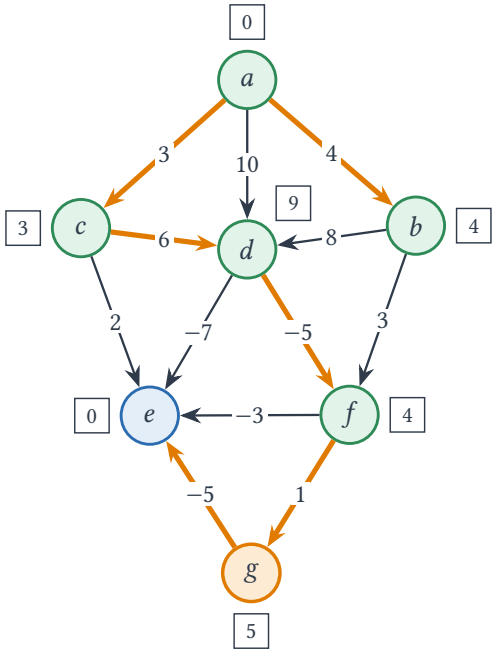

0
a
3
10
4
3
c
9
d
b
4
6
8
2
−7
−5
3
3
e
−3
f
7
−5
1
g
8

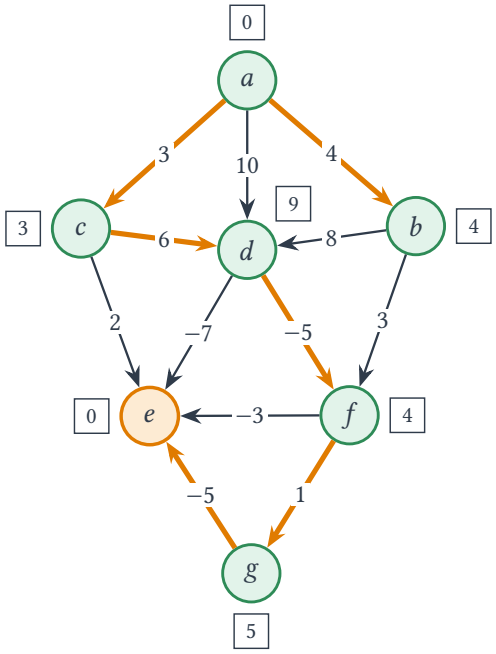

0
a
3
10
4
3
c
9
d
b
4
6
8
2
−7
−5
3
2
e
−3
f
4
−5
1
g
8


0
a
3
10
4
3
c
9
d
b
4
6
8
2
−7
−5
3
2
e
−3
f
4
−5
1
g
8


0
a
3
10
4
3
c
9
d
b
4
6
8
2
−7
−5
3
1
e
−3
f
4
−5
1
g
5


0
a
3
10
4
3
c
9
d
b
4
6
8
2
−7
−5
3
1
e
−3
f
4
−5
1
g
5





Contrairement à cet exemple, on constate, comme on l'a fait dans le paragraphe précédent, qu'avec des poids positifs l'algorithme se comporte particulièrement bien et d'une manière assez proche du parcours en largeur. Notamment, un sommet n'est placé qu'une fois sur la file de priorité : quand il en est retiré, d vaut déjà sa distance à s (c'est ce qu'on a montré pour l'algorithme de Dijkstra) et ne diminuera plus. On constate donc une vague de traitement qui progresse linéairement sur le graphe.

On peut donc se permettre de ne pas avoir à implémenter l'opération assez pénible de mise à jour de la priorité.

Remarque 30.95 Cette opération n'est pas vraiment compliquée dans un tas car il suffit de changer la valeur et de faire remonter l'élément le long de sa branche en direction de la racine en cas d'inversion de priorité. Le problème est qu'on a besoin de préserver un lien entre les sommets et leur emplacement dans le tas afin de savoir où est le nœud qui le contient. Le plus simple est de rajouter un dictionnaire, par exemple avec une table de hachage, mais cela alourdit conséquemment l'implémentation du tas car il faut maintenir l'intégrité du dictionnaire.

À la place, on va ajouter un sommet avec une autre priorité, meilleure, et il sera ainsi retiré avant. Pour ne pas traiter deux fois un sommet, on adopte alors une notion de marquage comme dans les parcours précédents. On retrouve ainsi l'algorithme de Dijkstra avec file de priorité présenté plus haut.

30.6.5 Floyd-Warshall

Pour résoudre le problème PlusCourtCheminToutCouple dans le cas d'un graphe orienté, l'algorithme de Floyd-Warshall va considérer une énumération $x_1, ..., x_n$ des sommets du graphe et construire une famille de matrices $dist^{(k)}$ où $dist^{(k)}(i,j)$ indique le poids d'un plus court chemin de x_i à x_j **dont les sommets intermédiaires** sont dans $x_1, ..., x_k$ lorsqu'un tel chemin existe, et ∞ sinon.

$$dist^{(0)}(i,j) = \{0 \text{ si } i = j$$
$$\pi(x_i \to x_j) \text{ si } x_i \to x_j \in A$$

On a donc ∞ sinon .

On remarque rapidement, **en l'absence de cycle négatif**, que ces matrices vérifient l'égalité, pour $k > 0$:

$$dist^{(k)}(i,j) = \min\Big(dist^{(k-1)}(i,j), dist^{(k-1)}(i,k) + dist^{(k-1)}(k,j)\Big)$$

En effet, un plus court chemin pouvant être supposé élémentaire, soit il ne passe pas par x_k, et son poids est $dist^{(k-1)}(i,j)$, soit il passe exactement une fois par x_k, et il se décompose en un chemin de x_i à x_k puis un chemin de x_k à x_j dont les sommets intermédiaires sont dans $x_1, ..., x_{k-1}$.

On en déduit un algorithme consistant à calculer successivement ces $n+1$ matrices avec trois boucles for imbriquées.

Exemple 30.96

Si on considère le graphe

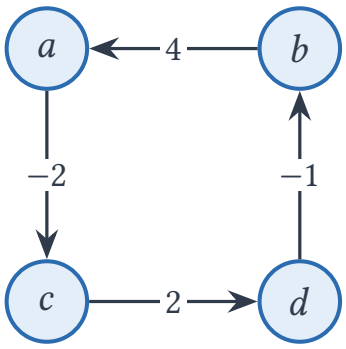


Figure 30.33 — Graphe à poids négatifs sans cycle négatif.

```
on va avoir la suite de matrices suivantes, dans l'ordre d'énumération
alphabétique des sommets :

$$
dist^{(0)} = \begin{pmatrix}
0&\infty&-2&\infty\\
4&0&\infty&\infty\\
\infty&\infty&0&2\\
\infty&-1&\infty&0\\
\end{pmatrix}
$$

$$
dist^{(1)} = \begin{pmatrix}
0&\infty&-2&\infty\\
4&0&2&\infty\\
\infty&\infty&0&2\\
\infty&-1&\infty&0\\
```

```
\end{pmatrix}
$$

$$
dist^{(2)} = \begin{pmatrix}
0&\infty&-2&\infty\\
4&0&2&\infty\\
\infty&\infty&0&2\\
3&-1&1&0\\
\end{pmatrix}
$$

$$
dist^{(3)} = \begin{pmatrix}
0&\infty&-2&0\\
4&0&2&4\\
\infty&\infty&0&2\\
3&-1&1&0\\
\end{pmatrix}
$$

$$
dist^{(4)} = \begin{pmatrix}
0&-1&-2&0\\
4&0&2&4\\
5&1&0&2\\
3&-1&1&0\\
\end{pmatrix}
$$
```

Une première version consisterait à calculer toutes les matrices, ce qui prend un espace en $O(|S|^3)$. On peut optimiser cela avec une paire de matrices qui changent de rôle, mais on peut faire mieux. En effet, on a $dist^{(k)}(i,k) = dist^{(k-1)}(i,k)$ car $dist^{(k-1)}(k,k) = 0$ en l'absence de cycle négatif, et de même $dist^{(k)}(k,j) = dist^{(k-1)}(k,j)$: la ligne et la colonne k ne sont pas modifiées à l'étape k. On en déduit donc qu'on peut effectuer le calcul *en place* dans une seule matrice. On obtient alors l'algorithme suivant :

Algorithme 30.97 — : Floyd-Warshall

Entrées :

- Un graphe pondéré $G = (S, A, \pi)$ et une énumération $x_1, ..., x_n$ de S

Déroulement :

- On initialise une matrice d telle que $d(i,i) = 0$, $d(i,j) = \pi(x_i \to x_j)$ si $x_i \to x_j \in A$ et $d(i,j) = \infty$ sinon
- Pour k allant de 1 à n
  - Pour i allant de 1 à n
    - Pour j allant de 1 à n
      - $d(i,j) = \min(d(i,j), d(i,k) + d(k,j))$

On en déduit le programme suivant, où le graphe est donné, comme pour Dijkstra, par listes d'adjacence pondérées et où les sommets sont numérotés à partir de 0. En OCaml et en C, on teste explicitement ∞ pour ne pas dépasser la capacité des entiers.

Python

```python
def floyd_warshall(g):
    n = len(g)
    inf = float('inf')
    d = [ [ inf ] * n for _ in range(n) ]
    for i in range(n):
        d[i][i] = 0
        for j, poids in g[i]:
            d[i][j] = poids
    for k in range(n):
```

```
        for i in range(n):
            for j in range(n):
                if d[i][k] + d[k][j] < d[i][j]:
                    d[i][j] = d[i][k] + d[k][j]
    return d
```

OCaml

```
let floyd_warshall g =
    let n = Array.length g in
    let inf = max_int in
    let d = Array.make_matrix n n inf in
    for i = 0 to n - 1 do
        d.(i).(i) <- 0;
        List.iter (fun (j, poids) -> d.(i).(j) <- poids) g.(i)
    done;
    for k = 0 to n - 1 do
        for i = 0 to n - 1 do
            for j = 0 to n - 1 do
                if d.(i).(k) <> inf && d.(k).(j) <> inf
                   && d.(i).(k) + d.(k).(j) < d.(i).(j)
                then d.(i).(j) <- d.(i).(k) + d.(k).(j)
            done
        done
    done;
    d
```

C

```
#include <stdlib.h>
#include <limits.h>

#define INF INT_MAX

// Renvoie la matrice des distances (à libérer avec libere_matrice)
int **floyd_warshall(graphe_pondere g) {
    int n = g.n;
    int **d = malloc(n * sizeof(int *));
    for (int i = 0; i < n; i++) {
        d[i] = malloc(n * sizeof(int));
        for (int j = 0; j < n; j++) {
            d[i][j] = INF;
        }
        d[i][i] = 0;
        for (int a = 0; a < g.degre[i]; a++) {
            d[i][g.voisins[i][a].but] = g.voisins[i][a].poids;
        }
    }
    for (int k = 0; k < n; k++) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (d[i][k] != INF && d[k][j] != INF
                    && d[i][k] + d[k][j] < d[i][j]) {
                    d[i][j] = d[i][k] + d[k][j];
                }
            }
        }
    }
    return d;
}

void libere_matrice(int **d, int n) {
    for (int i = 0; i < n; i++) {
        free(d[i]);
    }
    free(d);
}
```

En l'absence de cycle de poids négatif, la matrice obtenue vérifie ainsi $d(i,j) = dist(x_i, x_j)$. La complexité est en $O(|S|^3)$ en temps et $O(|S|^2)$ en espace.

Remarque 30.98 Il est aussi possible d'imposer que les plus courts chemins soient de longueur au moins 1, ce qui va mettre des ∞ sur la diagonale dans $dist^{(0)}$ et qui permettra, en fin d'algorithme, d'en déduire la présence de cycles ainsi que le poids d'un plus court cycle passant par x_i dans $d(i,i)$.

Il y a de nombreux contextes dans lesquels la détermination d'une matrice des plus petits poids est suffisante. Cependant, si on a également besoin des chemins eux-mêmes, il est possible de construire une matrice de parenté P où P_{ij} indique le prédécesseur de x_j dans un plus court chemin de x_i à x_j. On utilisera le symbole $\perp$ quand un tel chemin n'existe pas (ou quand $i = j$).

Pour calculer P il suffit d'adapter l'algorithme précédent en considérant les $P^{(k)}$ et en posant :

$$P_{ij}^{(0)} = \begin{cases} i & \text{si } x_i \to x_j \in A \\ \perp & \text{sinon} \end{cases}$$

Puis,

$$\text{Pour } k \geq 1, P_{ij}^{(k)} = \begin{cases} P_{ij}^{(k-1)} & \text{si } d_{ij}^{(k)} = d_{ij}^{(k-1)} \\ P_{kj}^{(k-1)} & \text{sinon} \end{cases}$$

Ce qui revient à changer le prédécesseur selon la résolution du min.

Remarque 30.99

- En appliquant Floyd-Warshall à un graphe quelconque, on peut détecter la présence de cycles de poids négatifs. En effet, si un tel cycle existe, on aura nécessairement $d(i,i) < 0$ à la fin pour un sommet x_i de ce cycle.
- Floyd-Warshall a une application importante dans le cas du calcul de la fermeture transitive d'un graphe. En effet, il suffit de considérer la matrice d'adjacence au départ et d'appliquer Floyd-Warshall pour en déduire la matrice d'adjacence de la fermeture réflexive et transitive (en remplaçant les 0 hors de la diagonale par des ∞) où les arêtes sont en plus pondérées par la longueur d'un plus court chemin qui la réalise.

EXERCICES 31

Graphes

DANS CE CHAPITRE

31.1 **Arbres**
31.2 **Parcours en largeur**
31.3 **Un exemple de parcours**

Ces exercices accompagnent le chapitre Graphes.

31.1 Arbres

Exercice

Soit $T = (S, A)$ un graphe non orienté avec $S \neq \emptyset$.

Montrer que les propriétés suivantes sont équivalentes :

1. T est acyclique et connexe ;
2. T est acyclique maximal (si on ajoute une arête il devient cyclique) ;
3. T est acyclique et $|A| \geq |S| - 1$;
4. T est connexe minimal (si on retire une arête il devient non connexe) ;
5. T est connexe et $|A| \leq |S| - 1$;
6. il existe un unique chemin élémentaire entre deux sommets de T.

On dit que T est un **arbre**.

Corrigé

$6 \Rightarrow 1$: il existe un chemin entre deux sommets quelconques donc T est connexe, et si T a un cycle, on en déduit deux chemins élémentaires distincts entre deux sommets de ce cycle en le coupant en deux.

$1 \Rightarrow 6$: soit $x, y \in S$ deux sommets distincts. Si, par l'absurde, il existe deux chemins élémentaires différents entre x et y, alors on peut considérer le dernier sommet z commun aux deux chemins avant qu'ils ne se séparent et le premier sommet t après z sur le premier chemin qui appartient aussi au second (il existe car les chemins se rejoignent forcément en y). On peut alors construire un cycle avec les deux sous-chemins de z à t qui n'ont, par construction, que z et t en commun. Ce cycle est de longueur au moins 3 : les deux sous-chemins ne peuvent pas être tous les deux réduits à l'arête $z \smile t$ puisque les chemins se séparent après z.

$1 \Rightarrow 2$: soit $x \smile y$ l'arête qu'on rajoute. Comme T est connexe, il y a un chemin élémentaire de x à y. En rajoutant la nouvelle arête à ce chemin, on obtient un cycle.

$2 \Rightarrow 1$: soit x, y deux sommets distincts. Si $x \smile y \notin A$, alors comme rajouter cette arête rendrait T cyclique, c'est que le cycle créé passe par cette arête et qu'il existe un chemin de x à y dans T. Ainsi, dans tous les cas, x et y sont reliés. T est bien connexe.

$1 \Rightarrow 4$: soit $x \smile y$ l'arête qu'on enlève. Comme T est acyclique, il ne peut exister de chemin élémentaire entre x et y sans cette arête, sinon ce chemin prolongé de l'arête omise serait un cycle dans T.

$4 \Rightarrow 1$: si T avait un cycle, il suffirait de supprimer n'importe quelle arête du cycle pour qu'il reste connexe. Ainsi, T est acyclique.

Pour les deux équivalences restantes, c'est un peu plus compliqué. On commence par montrer un lemme.

Lemme. Si T est un arbre (au sens de 1) alors $|A| = |S| - 1$.

Preuve du lemme. Par récurrence forte sur $|S|$. C'est direct si $|S| = 1$: il n'y a aucune arête.

Supposons le résultat vrai pour les arbres à moins de $n \geq 2$ sommets et soit T un arbre ayant n sommets. Comme T est connexe et a au moins deux sommets, il possède une arête $e = x \smile y$. Le graphe $T' = T - e$ est non connexe car T est connexe minimal ($1 \Rightarrow 4$). Il a exactement deux composantes connexes, celle de x et celle de y : tout sommet z est relié à x dans T et, si le chemin élémentaire correspondant passe par e, sa partie avant e relie z à x ou à y dans T'. Ces deux composantes $C_1 = (S_1, A_1)$ et $C_2 = (S_2, A_2)$ sont connexes et acycliques, donc des arbres ayant moins de n sommets, et l'hypothèse de récurrence s'applique : $|S| = |S_1| + |S_2| = |A_1| + |A_2| + 2 = |A| + 1$ car $A = A_1 \cup A_2 \cup \{e\}$, réunion disjointe. On a bien validé la récurrence.

On en déduit donc $1 \Rightarrow 3$ et $1 \Rightarrow 5$, ainsi que le corollaire suivant, en appliquant le lemme à chaque composante connexe : si $G = (S, A)$ est acyclique et a k composantes connexes, alors $|A| = |S| - k$.

$3 \Rightarrow 1$: soit $k \geq 1$ le nombre de composantes connexes de T. On a $|A| = |S| - k \geq |S| - 1$, donc $k \leq 1$ et ainsi $k = 1$: T est connexe.

$5 \Rightarrow 1$: il existe $A' \subset A$ tel que $T' = (S, A')$ soit connexe et acyclique. En effet, on commence avec $A' = A$ et tant que T' contient un cycle, on retire une arête du cycle de A', ce qui conserve la connexité. Ce processus termine car A' est fini. L'arbre T' obtenu vérifie $|A'| = |S| - 1 \geq |A|$ et donc $A = A'$. Ainsi T était déjà acyclique.

31.2 Parcours en largeur

Note dans cette partie on va considérer un graphe orienté, mais la plupart des notions se transmettent directement au cas des graphes non orientés.

On considère le parcours en largeur écrit ainsi :

Algorithme 31.1 — : parcours en largeur

Entrées :

- Un graphe $G = (S, A)$
- Un sommet $s \in S$ source du parcours

Déroulement :

- On crée une file ne contenant que s
- Tous les sommets sont non marqués.
- On pose $d(s) = 0$ et $d(x) = \infty$ pour $x \neq s$.
- Tant que la file est non vide
  - On retire un sommet x de la file
  - Si x n'est pas marqué
    - On marque x

- Pour chaque $x \to y \in A$ où $d(y) = \infty$
  - On ajoute y dans la file
  - On pose $d(y) = 1 + d(x)$
- On renvoie d

Dans la suite un sommet non marqué et dont $d(x) = \infty$ est dit **inconnu** alors qu'un sommet non marqué et dont $d(x) < \infty$ est dit **découvert**.

Définition 31.2

Soit $G = (S, A)$ un graphe et $x, y \in S$, on pose

$$\delta(x,y) = \begin{cases} \infty & \text{s'il n'existe aucun chemin } x \rightsquigarrow y \\ \min\{|\varphi| \mid \varphi : x \rightsquigarrow y\} & \text{sinon.} \end{cases}$$

Il s'agit de la plus petite longueur d'un chemin de x à y lorsqu'il en existe.

Question 1 Montrer que si $x \in S$ et $y \to z \in A$, alors $\delta(x,z) \leq \delta(x,y) + 1$.

Question 2 Soit $G = (S, A)$ et $s \in S$. On considère un parcours en largeur de source s. À la fin du BFS, montrer qu'on a $\forall x \in S, d(x) \geq \delta(s,x)$.

Question 3 On suppose que durant l'exécution du BFS, la file contient les sommets $(v_1, v_2, \ldots, v_r)$ où v_1 est la tête de la file (le sommet qui sera retiré en premier).

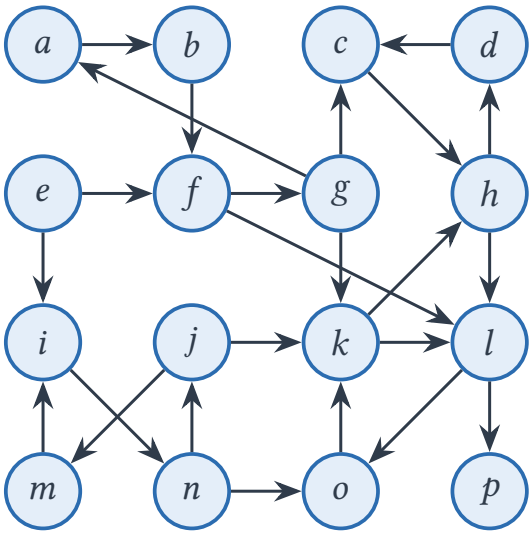
Montrer qu'on a $d(v_r) \leq d(v_1) + 1$ et que $d(v_1) \leq d(v_2) \leq \ldots \leq d(v_r)$.

Question 4 En déduire que si v_i et v_j sont insérés dans la file lors de l'exécution et que v_i a été inséré avant v_j, alors on a $d(v_i) \leq d(v_j)$ **au moment où on insère** v_j.

Question 5 Soit $G = (S, A)$ un graphe orienté et $s \in S$ source à partir duquel on a effectué un BFS. Montrer que pour tout $x \in S$, x est marqué si et seulement s'il est accessible depuis s, et que dans ce cas on a $d(x) = \delta(s,x)$ et le chemin emprunté par le parcours est un plus court chemin.

31.3 Un exemple de parcours

On considère le graphe orienté :



On effectue un parcours en profondeur récursif partant de a puis un parcours récursif partant de e.

Question 6 Déterminer les temps d'entrée et de sortie sur une frise chronologique et classifier les arêtes (voir la partie Classification des arêtes du cours).

Corrigé

On suppose que les voisins sont examinés dans l'ordre alphabétique. Le parcours depuis a découvre tous les sommets sauf e, i, j, m, n, que découvre ensuite le parcours depuis e. On obtient le mot bien parenthésé

$$(_a(_b(_f(_g(_c(_h(_d)_d(_l(_o(_k)_k)_o(_p)_p)_l)_h)_c)_g)_f)_b)_a(_e(_i(_n(_j(_m)_m)_j)_n)_i)_e$$

c'est-à-dire les temps suivants :

sommet	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p
t_e	1	2	5	7	23	3	4	6	24	26	11	9	27	25	10	14
t_s	22	21	18	8	32	20	19	17	31	29	12	16	28	30	13	15

Classification des arêtes :

- arêtes de liaison : $a \to b$, $b \to f$, $f \to g$, $g \to c$, $c \to h$, $h \to d$, $h \to l$, $l \to o$, $o \to k$, $l \to p$, $e \to i$, $i \to n$, $n \to j$, $j \to m$;
- arêtes arrière : $g \to a$, $d \to c$, $k \to h$, $k \to l$, $m \to i$;
- arêtes avant : $f \to l$, $g \to k$;
- arêtes transverses : $e \to f$, $j \to k$, $n \to o$.

Le graphe a donc des cycles, puisqu'il y a des arêtes arrière. Ce même graphe sert d'exemple pour l'algorithme de Kosaraju dans le chapitre Algorithmique avancée des graphes.

32

TRAVAUX PRATIQUES 32

Graphes

DANS CE CHAPITRE

32.1 Parcours de graphes en C
32.2 Étude d'un graphe issu d'un réseau social
32.3 Plus courts chemins en OCaml

Ces TP mettent en œuvre les notions du chapitre sur les graphes.

32.1 Parcours de graphes en C

32.1.1 Représentation

On va considérer le type suivant pour les graphes qui suppose qu'on n'aura jamais plus que `MAXV` sommets. On utilise ici une constante avec l'alias `#define` : partout où on écrit `MAXV`, il sera remplacé par la valeur 100.

```c
#define MAXV 100 /* nombre maximum de sommets */

struct edgenode {
    int y; // le voisin
    struct edgenode *next; // la suite de la liste
};
typedef struct edgenode edgenode;

struct graph {
    edgenode *edges[MAXV]; // tableau de listes d'adjacence
    int degree[MAXV]; // le degré de chaque sommet
    int nvertices;
    int nedges;
    bool directed; // indique si le graphe est orienté
};
typedef struct graph graph;
```

Dans un graphe non orienté, `nedges` compte chaque arête une seule fois, même si elle apparaît dans deux listes d'adjacence.

Question 1 Écrire une fonction

```c
void initialize_graph(graph *g, bool directed);
```

qui initialise le graphe `g` passé par pointeur comme étant le graphe vide, orienté ou non selon la valeur de `directed`.

Attention vous êtes libres d'initialiser les listes d'adjacence à la liste vide ou de considérer que cela sera fait dans la fonction `read_graph` ci-dessous.

Corrigé

```
void initialize_graph(graph *g, bool directed)
{
    g->directed = directed;
    g->nvertices = 0;
    g->nedges = 0;
    for (int i = 0; i < MAXV; i++)
    {
        g->edges[i] = NULL;
        g->degree[i] = 0;
    }
}
```

Question 2 Écrire une fonction

```
void insert_edge(graph *g, int x, int y, bool directed);
```

qui insère une arête de `x` à `y` en considérant `g` comme orienté ou non selon `directed` **et donc en ignorant** `g->directed`.

Attention il faudra prendre garde au fait que `g` soit orienté ou non. Dans le second cas, on représente les arêtes comme dans un graphe orienté symétrique, donc il faut en ajouter deux. C'est la raison pour laquelle on utilise le paramètre `directed` plutôt que de consulter `g->directed` : on pourra ainsi appeler récursivement `insert_edge(g, y, x, true)`.

Corrigé

```
void insert_edge(graph *g, int x, int y, bool directed)
{
    edgenode *edge = malloc(sizeof(edgenode));
    edge->y = y;
    edge->next = g->edges[x];
    g->edges[x] = edge;
    g->degree[x] = 1 + g->degree[x];
    if (!directed) {
        insert_edge(g, y, x, true);
    } else {
        g->nedges = 1 + g->nedges;
    }
}
```

Pour travailler sur des graphes, on va écrire une fonction permettant de lire un fichier contenant le graphe sous le format suivant :

- une première ligne contenant trois entiers, le nombre de sommets n, le nombre d'arêtes p et 0 ou 1 selon que le graphe soit non orienté ou orienté ;
- ensuite p lignes contenant deux entiers i et j et indiquant qu'il y a une arête de i vers j.

Par exemple :

```
4 5 1
0 1
0 2
1 2
2 3
3 1
```

sera représenté par le graphe :

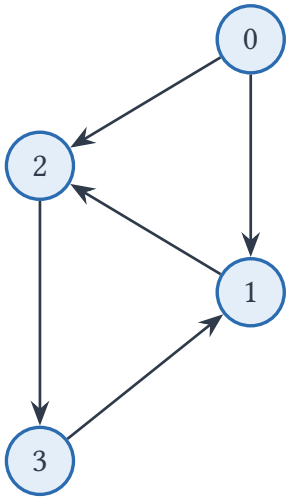


Figure 32.1 — Graphe orienté décrit par le fichier d'exemple.

Remarque 32.1 On va utiliser ici l'entrée standard, c'est-à-dire l'entrée de l'utilisateur depuis le terminal. Cependant, il est possible de rediriger cette entrée depuis un fichier. En effet, si on appelle

```
./monprogramme < monfichier
```

alors l'entrée standard sera le contenu de `monfichier`. Cela permet de ne pas avoir à se préoccuper d'ouvrir de fichier et d'utiliser directement `scanf`.

Rappel `scanf("%d %d", &x, &y);` va lire deux entiers et placer la valeur du premier dans `x` et la valeur du second dans `y`.

Question 3 Écrire une fonction

```
void read_graph(graph *g);
```

qui lit un graphe depuis l'entrée standard et le place dans `g` **après l'avoir initialisé**.

Corrigé

Attention à ne pas lire directement le nombre d'arêtes dans `g->nedges` : ce champ est déjà incrémenté par `insert_edge`.

```
void read_graph(graph *g)
{
    int directed, nedges;
    initialize_graph(g, false);
    scanf("%d %d %d", &g->nvertices, &nedges, &directed);
    assert(0 <= g->nvertices && g->nvertices <= MAXV);
    g->directed = directed == 1;
    // g->nedges est mis à jour par insert_edge
    for (int i = 0; i < nedges; i++)
    {
        int x, y;
        scanf("%d %d", &x, &y);
        insert_edge(g, x, y, g->directed);
    }
}
```

Question 4 Écrire une fonction

```
void free_edges(graph *g);
```

qui libère les listes d'adjacence.

Corrigé

```
void free_edges(graph *g)
{
    for (int i = 0; i < g->nvertices; i++)
    {
        edgenode *n = g->edges[i];
        while(n != NULL)
        {
            edgenode *temp = n->next;
            free(n);
            n = temp;
        }
        g->edges[i] = NULL;
    }
}
```

32.1.2 Parcours en profondeur récursif

On va modifier la structure de graphe et rajouter trois nouveaux champs :

```
bool discovered[MAXV]; // Quels sommets sont connus
bool processed[MAXV]; // Quels sommets sont traités
int parent[MAXV]; // parent[x] est le père de x dans le parcours
                  // s'il n'y en a pas, c'est -1
```

Remarque 32.2 Avec tous les champs ajoutés dans la suite, une valeur de type `graph` occupe plusieurs kilo-octets. On évitera de la passer par valeur et on pourra l'allouer avec `malloc` dans le `main`.

Question 5 Écrire une fonction

```
void initialize_search(graph *g);
```

qui initialise ces tableaux pour commencer une nouvelle recherche.

Corrigé

```
void initialize_search(graph *g)
{
    for (int i = 0; i < g->nvertices; i++)
    {
        g->discovered[i] = false;
        g->processed[i] = false;
        g->parent[i] = -1;
    }
}
```

On va définir trois fonctions qui seront appelées lors d'un parcours et qu'on pourra redéfinir.

```
void process_vertex_early(graph *g, int v)
{
    printf("processing vertex %d\n", v);
}

void process_vertex_late(graph *g, int v)
{
}

void process_edge(graph *g, int x, int y)
{
    printf("processed edge %d --> %d\n", x, y);
}
```

Question 6 Écrire une fonction récursive

```
void dfs(graph *g, int x);
```

qui effectue un parcours en profondeur en partant du sommet `x`.

Cette fonction va appeler `process_vertex_early` au début du traitement de `x`, `process_vertex_late` à la fin et `process_edge` pour chaque arête rencontrée.

Attention si le graphe est non orienté et qu'on a rencontré `x -> y` dans cet ordre, on ne fera pas de traitement dans le sens `y -> x`.

Corrigé

On traite les arêtes de l'arbre `x -> y` *avant* de découvrir `y`, ce qui permettra de les reconnaître dans `process_edge` (`y` n'est pas encore découvert).

```
void dfs(graph *g, int x)
{
    g->discovered[x] = true;
    process_vertex_early(g, x);

    edgenode *n = g->edges[x];
    while (n != NULL)
    {
        int y = n->y;
        if (!g->discovered[y])
        {
            // arête de l'arbre : on la traite avant de découvrir y
            process_edge(g, x, y);
            g->parent[y] = x;
            dfs(g, y);
        }
        else if (g->directed || (!g->processed[y] && g->parent[x] != y))
        {
            // dans un graphe non orienté, on ne traite pas y -> x si
            // x -> y a déjà été traitée (y est le père de x ou y est traité)
            process_edge(g, x, y);
        }
        n = n->next;
    }

    process_vertex_late(g, x);
    g->processed[x] = true;
}
```

Question 7 Comme on l'a démontré, le parcours dans un graphe non orienté permet de traiter exactement la composante connexe du sommet de départ.

Écrire une fonction

```
void connected_components(graph *g);
```

qui affiche les composantes connexes sous la forme (l'ordre des sommets dépendant de celui des listes d'adjacence) :

```
Component 1: 0 1 2 3 4
Component 2: 5 6 7
```

pour le graphe

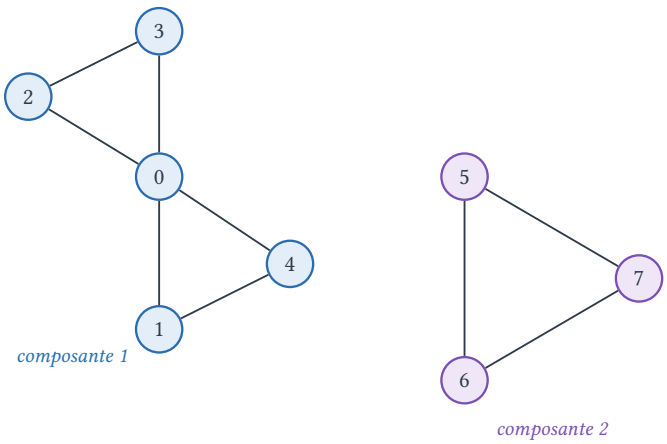


Figure 32.2 — Graphe à deux composantes connexes : $\{0, 1, 2, 3, 4\}$ et $\{5, 6, 7\}$.

Il sera nécessaire de modifier les fonctions `process_*`.

Corrigé

On remplace les fonctions par :

```
void process_vertex_early(graph *g, int v)
{
    printf(" %d", v);
}

void process_vertex_late(graph *g, int v)
{
}

void process_edge(graph *g, int x, int y)
{
}
```

et on définit :

```
void connected_components(graph *g)
{
    initialize_search(g);
    int comp = 0;
    for (int i = 0; i < g->nvertices; i++)
    {
        if (!g->discovered[i])
        {
            comp = comp + 1;
            printf("Component %d:", comp);
            dfs(g, i);
            printf("\n");
        }
    }
}
```

On relance ainsi le parcours en repartant d'un sommet non découvert. Avec le fichier donné en exemple ci-dessous (où les arêtes sont insérées en tête des listes d'adjacence), on obtient `Component 1: 0 4 1 3 2` et `Component 2: 5 7 6`.

```
8 9 0
0 1
0 2
0 3
1 4
```

```
4 0
3 2
5 6
5 7
7 6
```

Question 8 Adaptez le parcours pour détecter des cycles. Quand vous détectez un cycle, affichez les sommets qui le composent. Dans le graphe précédent, avec le fichier ci-dessus, il faut afficher :

```
Cycle : 0 1 4
Cycle : 0 2 3
Cycle : 5 6 7
```

Corrigé

Il suffit de changer `process_edge` dans le parcours de toutes les composantes pour détecter les arêtes arrière : on affiche alors le cycle en remontant les pères depuis `x` jusqu'à `y`.

```
void process_edge(graph *g, int x, int y)
{
    // Teste si on a une arête arrière
    if (g->discovered[y]
            && !g->processed[y]
            && (g->directed || g->parent[x] != y))
    {
        printf("Cycle : %d", y);
        int cur = x;
        while (cur != y && cur > -1)
        {
            printf(" %d", cur);
            cur = g->parent[cur];
        }
        printf("\n");
    }
}
```

Question 9 À l'aide d'un parcours, déterminez si un graphe non orienté est biparti.

Corrigé

On rajoute des champs

```
bool color[MAXV];
bool bipartite;
```

dans `graph` et on commence en donnant la couleur `true` au sommet de départ de chaque parcours. Ensuite, il suffit de colorer `y` avec la couleur différente de celle de x quand on découvre y par une arête $x \to y$ de l'arbre, puis de vérifier à chaque autre arête qu'elle relie des sommets de couleurs différentes.

```
void process_edge(graph *g, int x, int y)
{
    if (!g->discovered[y])
        // arête de l'arbre : y prend la couleur opposée à celle de x
        g->color[y] = !g->color[x];
    else
        g->bipartite = g->bipartite && (g->color[x] != g->color[y]);
}

bool is_bipartite(graph *g)
{
    initialize_search(g);
```

```
    g->bipartite = true;
    for (int i = 0; i < g->nvertices; i++)
    {
        if (!g->discovered[i])
        {
            g->color[i] = true;
            dfs(g, i);
        }
    }
    return g->bipartite;
}
```

En effet, si le graphe est biparti, la couleur imposée à chaque sommet par l'arbre de parcours est celle de sa classe, et aucune arête ne relie deux sommets de même couleur ; réciproquement, si aucune arête ne relie deux sommets de même couleur, les deux couleurs forment une bipartition.

32.1.3 Temps et classification des arêtes

On rajoute à la structure `graph` deux tableaux et un entier :

```
int time; // l'horloge
int entry_time[MAXV];
int exit_time[MAXV];
```

Question 10 Rajouter à `dfs` le maintien de l'horloge et des temps d'entrée et de sortie.

Corrigé

Il suffit de modifier les traitements de sommets (en ayant initialisé `g->time` à 0) :

```
void process_vertex_early(graph *g, int v)
{
    g->entry_time[v] = g->time;
    g->time = g->time + 1;
}

void process_vertex_late(graph *g, int v)
{
    g->exit_time[v] = g->time;
    g->time = g->time + 1;
}
```

On définit des constantes :

```
#define TREE 0
#define BACK 1
#define FORWARD 2
#define CROSS 3
```

Question 11 Écrire une fonction :

```
int edge_classification(graph *g, int x, int y);
```

qui pourra être appelée dans `process_edge` pour déterminer la classe d'une arête selon les constantes précédentes (voir la classification des arêtes dans le cours). On renverra −1 si l'arête ne peut être déterminée (est-ce possible ?).

Corrigé

Avec notre `dfs`, une arête de l'arbre est traitée avant que `y` soit découvert. Sinon, `y` est découvert : il est soit en cours de traitement (c'est un ancêtre de `x`, arête arrière), soit déjà traité, et on compare alors les

temps d'entrée. Chaque arête tombe donc dans l'un des quatre cas et on ne renvoie jamais −1. Dans un graphe non orienté, on n'obtient que des arêtes de l'arbre et des arêtes arrière.

```
int edge_classification(graph *g, int x, int y)
{
    if (!g->discovered[y])
        return TREE;
    if (!g->processed[y])
        return BACK;
    if (g->entry_time[x] < g->entry_time[y])
        return FORWARD;
    return CROSS;
}
```

32.1.4 Parcours avec une structure

On va réutiliser ici des implémentations de files et de piles dans un tableau de taille fixe.

Question 12 Compléter les implémentations en se basant sur ce qui a déjà été fait dans les TP précédents. On choisit une capacité `MAXQ` suffisante pour la suite (voir la question suivante).

```
#define MAXQ (MAXV * MAXV + 1)

struct queue {
    int elts[MAXQ];
    int front;
    int back;
};
typedef struct queue queue;

struct stack {
    int elts[MAXQ];
    int back;
};
typedef struct stack stack;

void init_stack(stack *s)
{
    // initialise la pile
}

bool empty_stack(stack *s)
{
    // indique si la pile est vide
}

int pop(stack *s)
{
    // dépile un élément
}

void push(stack *s, int x)
{
    // empile x sur la pile s
}

void init_queue(queue *s)
{
    // initialise la file
}

bool empty_queue(queue *s)
{
    // indique si la file est vide
}
```

```c
int dequeue(queue *s)
{
    // défile un élément
}

void enqueue(queue *s, int x)
{
    // enfile x sur la file s
}
```

Corrigé

```c
void init_stack(stack *s)
{
    s->back = 0;
}

bool empty_stack(stack *s)
{
    return s->back == 0;
}

int pop(stack *s)
{
    assert(s->back > 0);
    s->back = s->back - 1;
    return s->elts[s->back];
}

void push(stack *s, int x)
{
    assert(s->back < MAXQ);
    s->elts[s->back] = x;
    s->back = s->back + 1;
}

// file circulaire : on laisse toujours une case vide pour
// distinguer la file vide (front == back) de la file pleine
void init_queue(queue *s)
{
    s->front = 0;
    s->back = 0;
}

bool empty_queue(queue *s)
{
    return s->front == s->back;
}

int dequeue(queue *s)
{
    assert(!empty_queue(s));
    int x = s->elts[s->front];
    s->front = (s->front + 1) % MAXQ;
    return x;
}

void enqueue(queue *s, int x)
{
    assert((s->back + 1) % MAXQ != s->front);
    s->elts[s->back] = x;
    s->back = (s->back + 1) % MAXQ;
}
```

Question 13 Écrire des fonctions

```c
void dfs_stack(graph *g, int src);
void bfs(graph *g, int src);
```

effectuant un parcours en utilisant pour les sommets à visiter une pile ou une file (c'est le parcours *quelconque* du cours). On ne pourra pas maintenir les temps de sortie ici car cela n'a plus vraiment de sens.

Corrigé

Chaque sommet traité ajoute tous ses voisins dans le sac : on peut donc ajouter jusqu'à $|A| + 1 \leq$ `MAXV * MAXV + 1` sommets, ce qui justifie la capacité choisie. Une structure de taille fixe sur la pile d'appel serait trop grosse : on l'alloue avec `malloc`.

```c
void dfs_stack(graph *g, int src)
{
    stack *s = malloc(sizeof(stack));
    init_stack(s);
    push(s, src);
    while (!empty_stack(s))
    {
        int x = pop(s);
        if (!g->processed[x])
        {
            g->processed[x] = true;
            process_vertex_early(g, x);

            edgenode *n = g->edges[x];
            while (n != NULL)
            {
                push(s, n->y);
                n = n->next;
            }
        }
    }
    free(s);
}

void bfs(graph *g, int src)
{
    queue *s = malloc(sizeof(queue));
    init_queue(s);
    enqueue(s, src);
    while (!empty_queue(s))
    {
        int x = dequeue(s);
        if (!g->processed[x])
        {
            g->processed[x] = true;
            process_vertex_early(g, x);

            edgenode *n = g->edges[x];
            while (n != NULL)
            {
                enqueue(s, n->y);
                n = n->next;
            }
        }
    }
    free(s);
}
```

Question 14 À l'aide d'un parcours en largeur, déterminez, étant donnés un sommet `x` et un sommet `y` de sa composante connexe, un plus court chemin de `x` vers `y`. On l'affichera sous la forme `0 -- 3 -- 1 -- 5`.

Corrigé

On rajoute un champ comme vu dans le cours :

```
int d[MAXV];
```

on marque `d[x] = 0` et on utilise un parcours en largeur où l'on marque les sommets dès qu'on les ajoute dans la file, en retenant leur père. On n'ajoute ainsi chaque sommet qu'une fois dans la file.

```
void bfs_dist(graph *g, int src)
{
    queue *s = malloc(sizeof(queue));
    init_queue(s);
    enqueue(s, src);
    g->discovered[src] = true;
    g->d[src] = 0;

    while (!empty_queue(s))
    {
        int x = dequeue(s);
        g->processed[x] = true;

        edgenode *n = g->edges[x];
        while (n != NULL)
        {
            if (!g->discovered[n->y])
            {
                g->discovered[n->y] = true;
                g->d[n->y] = 1 + g->d[x];
                g->parent[n->y] = x;
                enqueue(s, n->y);
            }
            n = n->next;
        }
    }
    free(s);
}

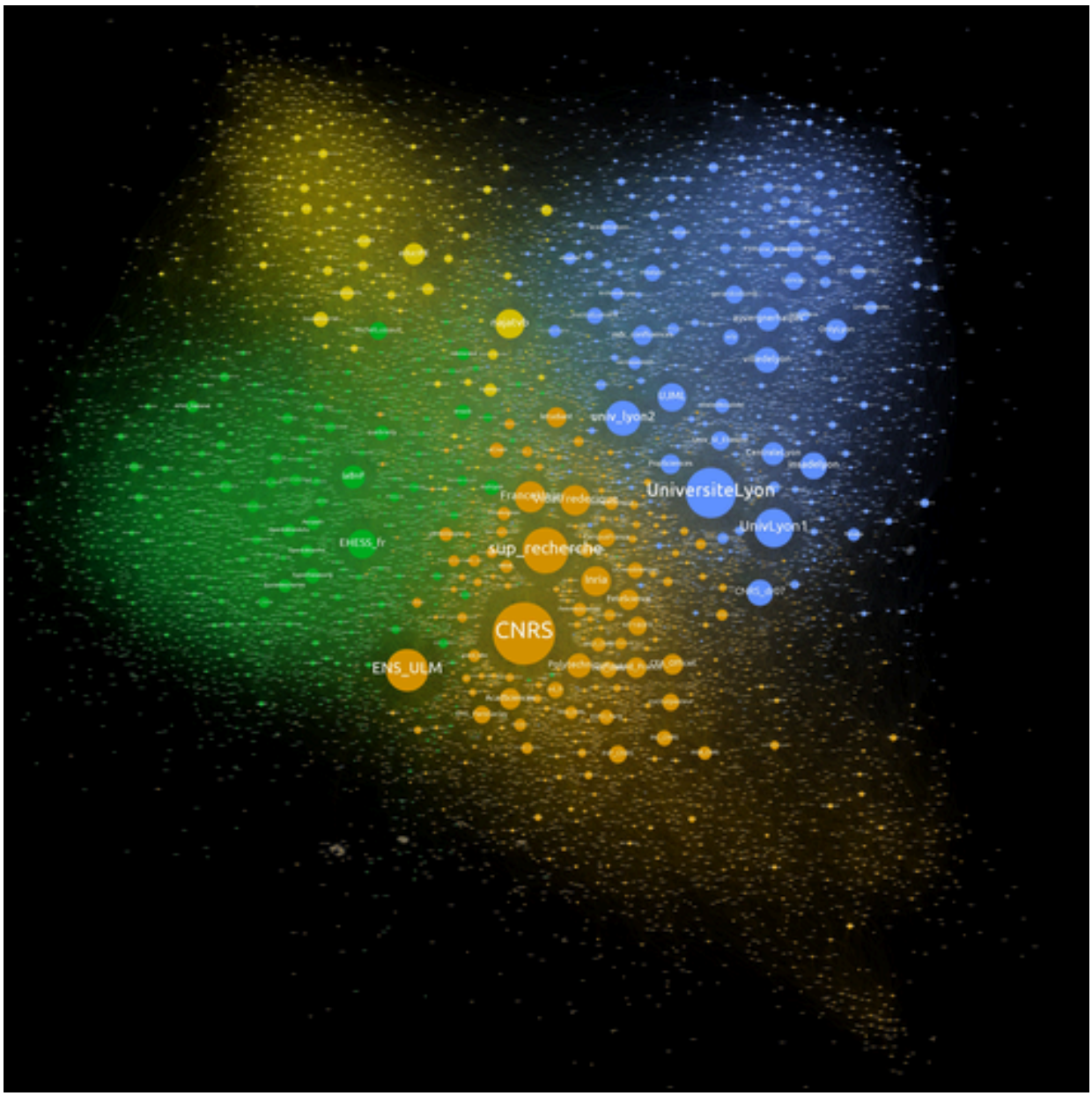
void print_path(graph *g, int x, int y)
{
    // affiche le chemin de x à y en remontant les pères depuis y
    if (y == x)
        printf("%d", x);
    else
    {
        print_path(g, x, g->parent[y]);
        printf(" -- %d", y);
    }
}

void shortest_path(graph *g, int x, int y)
{
    initialize_search(g);
    for (int i = 0; i < g->nvertices; i++)
        g->d[i] = -1;
    bfs_dist(g, x);
    assert(g->d[y] != -1); // y est dans la composante de x
    print_path(g, x, y);
    printf("\n");
}
```

32.2 Étude d'un graphe issu d'un réseau social

Dans ce TP on va étudier le graphe **orienté** des abonnés (*followers*) du compte Twitter `@ENSdeLyon`.

Une représentation graphique de ce graphe est donnée dans l'image suivante :



Les sommets sont les comptes donnés par leur identifiant (le `@identifiant` de Twitter) et une arête $x \to y$ indique que le compte x est abonné au compte y.

Ce graphe est assez conséquent : il comporte 8418 sommets et 305288 arêtes. Il nous permettra ainsi d'étudier en pratique la complexité des différents algorithmes étudiés. On va commencer par lire ce graphe depuis un fichier, ensuite, on en déduira différents graphes associés (sous-graphes, symétrisés par excès ou par défaut…) sur lesquels on pourra appliquer les algorithmes demandés. Une table de résultats est fournie en fin de TP pour vérifier vos résultats.

32.2.1 Définition et lecture du graphe

Le graphe est donné dans le fichier ENSdeLyon.graph. Il s'agit d'un fichier texte ayant la structure suivante :

- un entier `n_sommets` sur une ligne ;
- un entier `n_aretes` sur une ligne ;
- `n_sommets` lignes contenant une chaîne de caractères représentant l'identifiant d'un sommet ;
- `n_aretes` couples de lignes comportant sur la première un entier `src` et sur la seconde un entier `tgt` indiquant une arête `src -> tgt`.

Question 15 Expliquer pourquoi cela ne semble pas être une bonne idée de représenter ce graphe par une matrice d'adjacence.

Corrigé

La matrice aurait $8418^2 \approx 7 \times 10^7$ cases, alors qu'il n'y a que 305 288 arêtes : moins d'une case sur 200 serait utile. Les listes d'adjacence occupent un espace en $O(|S| + |A|)$ et permettent de parcourir les voisins d'un sommet en temps proportionnel à son degré.

On va utiliser le type suivant permettant de représenter le graphe par listes d'adjacence :

```
type graphe = {
    sommets : string array;
    aretes : int list array
}
```

Pour lire le graphe depuis le fichier, le plus simple est de le rediriger sur l'entrée standard (*Rappel* `./monprogramme < monfichier`) et d'utiliser les deux fonctions suivantes :

- `read_int : unit -> int` lit une ligne composée d'un entier et renvoie sa valeur ;
- `read_line : unit -> string` lit une ligne et la renvoie sans le caractère de saut de ligne, c'est-à-dire, sans le `'\n'`.

Alternativement, on peut lire le graphe depuis un fichier avec :

- `open_in : string -> in_channel` qui crée un descripteur de fichier en lecture pour le nom de fichier passé en paramètre ;
- `input_line : in_channel -> string` qui lit une ligne dans le descripteur et la renvoie sans le saut de ligne ;
- `int_of_string : string -> int` qui convertit une chaîne contenant un entier en entier.

Question 16 Écrire une fonction `read_graphe : unit -> graphe` qui lit un graphe dans le format précédent sur l'entrée standard et tester que vous arrivez à lire le fichier.

Un graphe minimaliste de 3 sommets et 4 arêtes est donné dans le fichier test.graph afin de vous permettre de tester votre fonction.

Corrigé

```
let read_graphe () =
    let nb_sommets = read_int () in
    let nb_aretes = read_int () in

    let sommets = Array.init nb_sommets
        (fun _ -> read_line ()) in

    let aretes = Array.make nb_sommets [] in

    for _ = 0 to nb_aretes - 1 do
        let src = read_int () in
        let tgt = read_int () in

        aretes.(src) <- tgt :: aretes.(src)
    done;
    { sommets=sommets; aretes=aretes }
```

Si $G = (S, A)$ est un graphe dont les sommets sont énumérés $S = \{s_0, s_1, ..., s_{n-1}\}$, on note, pour $p \leq n$, G_p le sous-graphe induit par $\{s_0, ..., s_{p-1}\}$.

Question 17 Écrire une fonction `restriction : graphe -> int -> graphe` telle que `restriction g p`, où `g` est la représentation d'un graphe G, renvoie la représentation du graphe G_p.

Corrigé

On adopte une approche proche de la fonction précédente : on itère sur les arêtes du graphe initial et on ne sélectionne que celles qui sont compatibles avec la restriction.

```
let restriction g p =
    let sommets = Array.sub g.sommets 0 p in
    let aretes = Array.make p [] in
    for i = 0 to p - 1 do
        List.iter (fun j ->
            if j < p then aretes.(i) <- j :: aretes.(i))
            g.aretes.(i)
    done;
    { sommets = sommets; aretes = aretes }
```

Notons ici qu'on aurait pu avoir une approche plus fonctionnelle pour sélectionner les bonnes arêtes :

```
let restriction g p =
    let sommets = Array.sub g.sommets 0 p in
    let aretes = Array.map (List.filter ((>)p))
        (Array.sub g.aretes 0 p) in
    { sommets = sommets; aretes = aretes }
```

Si $G = (S, A)$ est un graphe orienté, on a vu au paragraphe Graphes non orientés les graphes non orientés par défaut et par excès, G^- et G^+, qui lui sont associés.

Question 18 Écrire des fonctions `par_defaut : graphe -> graphe` et `par_exces : graphe -> graphe` qui, étant donné un graphe G, renvoient les graphes G^- et G^+ représentés en tant que graphes orientés symétriques.

Corrigé

On adopte encore l'approche de création par remplissage.

```
let par_defaut g =
    let n = Array.length g.sommets in
    let sommets = Array.copy g.sommets in
    let aretes = Array.make n [] in
    for i = 0 to n - 1 do
        List.iter (fun j ->
            if List.mem i g.aretes.(j)
            then aretes.(i) <- j :: aretes.(i))
            g.aretes.(i)
    done;
    { sommets = sommets; aretes = aretes }

let par_exces g =
    let n = Array.length g.sommets in
    let sommets = Array.copy g.sommets in
    (* on recopie les arêtes existantes *)
    let aretes = Array.copy g.aretes in
    for i = 0 to n - 1 do
        List.iter (fun j ->
            (* on rajoute les retours absents *)
            if not (List.mem i g.aretes.(j))
            then aretes.(j) <- i :: aretes.(j))
            g.aretes.(i)
    done;
    { sommets = sommets; aretes = aretes }
```

Si $G = (S, A)$ est un graphe orienté, on note $rev(G) = (S, A')$ son miroir qui vérifie $(i, j) \in A \Leftrightarrow (j, i) \in A'$, c'est-à-dire qui renverse toutes les arêtes.

Question 19 Écrire une fonction `miroir : graphe -> graphe` qui renvoie le miroir d'un graphe.

Corrigé

Cette fonction ressemble à la construction du graphe par excès, sans recopier les arêtes initiales :

```
let miroir g =
    let n = Array.length g.sommets in
    let sommets = Array.copy g.sommets in
    let aretes = Array.make n [] in
    for i = 0 to n - 1 do
        List.iter (fun j -> aretes.(j) <- i :: aretes.(j))
            g.aretes.(i)
    done;
    { sommets = sommets; aretes = aretes }
```

Dans la suite du sujet, on note $\mathcal{G}$ le graphe des abonnés contenu dans le fichier. On va considérer dans la suite les graphes : $\mathcal{G}, rev(\mathcal{G}), \mathcal{G}^-, \mathcal{G}^+, \mathcal{G}_{500}, rev(\mathcal{G}_{500}), \mathcal{G}_{500}^-$ et $\mathcal{G}_{500}^+$.

32.2.2 Statistiques sur les degrés

Question 20 Écrire une fonction `stat_degre : graphe -> int * float` qui renvoie un couple (d_{max}, d_{moy}) où d_{max} est le plus grand des degrés (sortants) du graphe et d_{moy} est le degré moyen donné par un nombre flottant.

Corrigé

On itère directement sur les sommets :

```
let stat_degre g =
    let n = Array.length g.sommets in
    let sum_d, max_d = ref 0, ref 0 in
    for i = 0 to n-1 do
        let d = List.length g.aretes.(i) in
        max_d := max !max_d d;
        sum_d := !sum_d + d
    done;
    !max_d, float_of_int !sum_d /. float_of_int n
```

Notons qu'on peut, ici aussi, écrire une fonction utilisant la bibliothèque standard efficacement :

```
let stat_degre g =
    let a = Array.map List.length g.aretes in
    let fold = Array.fold_left in
    let foi = float_of_int in
    let n = Array.length g.aretes in
    fold max 0 a, foi (fold (+) 0 a) /. foi n
```

32.2.3 Parcours en largeur

On va réaliser ici un parcours en largeur qui sera amené à être modifié et enrichi dans les questions suivantes. On vous laisse libre d'enrichir ce parcours en utilisant des fonctionnelles pour les traitements ou de modifier le code du parcours directement.

Pour utiliser une file, on va utiliser le module Queue. Dans le parcours on va calculer la fonction de distance `d` et pour gérer les cas où $d(x) = \infty$, on va la représenter par un `int option array`. Si `d.(x) = None` c'est que `x` est inconnu, on peut donc se servir de ce tableau pour avoir l'état d'un sommet.

Question 21 Écrire une fonction `bfs : graphe -> int -> int option array * int option array` telle que `bfs g x` renvoie un couple `(d,parent)` où `d.(y)` est la distance minimale de x à y et `parent.(y)` est l'indice du prédécesseur de `y` dans un tel chemin de x à y.

Corrigé

On implémente directement le parcours en largeur vu dans le cours.

```
let bfs g x =
    let n = Array.length g.sommets in
    let p = Array.make n None in
    let d = Array.make n None in
    let a_traiter = Queue.create () in
    Queue.add x a_traiter;
    d.(x) <- Some 0;
    while not (Queue.is_empty a_traiter) do
        let x = Queue.take a_traiter in
        List.iter (fun y ->
                Queue.add y a_traiter;
                p.(y) <- Some x;
                d.(y) <- Some (Option.get d.(x) + 1))
            (List.filter (fun y -> d.(y) = None)
                g.aretes.(x))
    done;
    d, p
```

Question 22 Écrire une fonction `chemin : graphe -> int option array -> int -> int list` telle que `chemin g parent y` renvoie les sommets présents dans un chemin de `x` à `y`, où `parent` a été obtenu par `bfs g x`.

Corrigé

On utilise ici la récursivité terminale pour remettre le chemin dans l'ordre.

```
let chemin g p y =
    let rec aux y l =
        match p.(y) with
        | None -> y :: l
        | Some x -> aux x (y :: l)
    in aux y []
```

Question 23 Écrire une fonction `affiche_chemin : graphe -> int list -> unit` qui prend un graphe et un chemin donné par la fonction précédente et l'affiche avec le format :

```
compte1 -> compte2 -> ... -> compten
```

Corrigé

```
let affiche_chemin g p =
    Printf.printf "%s\n"
        (String.concat " -> "
            (List.map (fun i -> g.sommets.(i)) p))
```

Si $x \in S$, on note $\underline{x} = \{y \in S \mid x \rightsquigarrow y\}$ l'ensemble des sommets accessibles depuis x (x compris).

Question 24 Écrire une fonction `accessibles : graphe -> int -> int` qui calcule le cardinal de $\underline{x}$ étant donnés un graphe G et un sommet x donné par son indice.

Corrigé

```
let accessibles g x =
    let d, p = bfs g x in
    Array.fold_left (+) 0
        (Array.map
            (fun v -> if v = None then 0 else 1) d)
```

Question 25 Écrire une fonction `max_accessibles : graphe -> int * int` qui renvoie un couple $(x, |\underline{x}|)$ où x est un sommet pour lequel $|\underline{x}|$ est maximal.

Corrigé

```
let max_accessibles g =
    let m, v = ref 0, ref (accessibles g 0) in
    for i = 1 to Array.length g.sommets - 1 do
        let v' = accessibles g i in
        if v' > !v
        then ( v := v'; m := i )
    done;
    !m, !v
```

32.2.4 Plus long chemin et diamètre

Question 26 Écrire une fonction `plus_loin : graphe -> int -> int * int * int list` telle que `plus_loin g x` renvoie un triplet `(y, d, c)` où `y` est un sommet accessible depuis `x` le plus éloigné de `x`, `d` sa distance à `x` et `c` un plus court chemin de `x` à `y`.

Corrigé

```
let plus_loin g x =
    let n = Array.length g.sommets in
    let d, p = bfs g x in
    let i = ref 0 in
    while d.(!i) = None do
        incr i
    done;
    let m = ref (!i) in
    for j = !i+1 to n - 1 do
        match d.(j) with
        | Some v -> if v > Option.get d.(!m) then m := j
        | None -> ()
    done;
    let v = Option.get d.(!m) in
    !m, v, chemin g p !m
```

On appelle ici diamètre d'un graphe la plus grande distance $d(x, y)$ entre deux sommets tels que y est accessible depuis x.

Question 27 Écrire une fonction `diametre : graphe -> int list` qui renvoie un chemin réalisant le diamètre d'un graphe.

Corrigé

On lance un parcours en largeur depuis chaque sommet, ce qui donne une complexité en $O(|S|(|S| + |A|))$.

```
let diametre g =
    let n = Array.length g.sommets in
    let v, p = ref (-1), ref [] in
    for i = 0 to n-1 do
        let j, v', chemin = plus_loin g i in
        if !v < v'
        then begin
            v := v';
            p := chemin
        end
    done;
    !p
```

32.2.5 Table de résultats

Attention : s'il faut peu de temps pour obtenir les résultats pour le sous-graphe de 500 sommets, c'est beaucoup plus long sur le graphe en entier (de l'ordre de la minute pour le calcul des diamètres, en compilant avec `ocamlopt`). En cas d'égalité, le sommet ou le chemin obtenu peut dépendre de l'ordre des listes d'adjacence.

- $\mathcal{G}_{500}$:

```
degré max 10
degré moyen 0.430000
max_accessibles Mishkalashnikov avec 16 sommets
Diamètre 7 réalisé par :
Isaac__K -> naxonlabs -> faezeh_db -> MooreInst ->
    fath_gabrielle -> hypothesesorg -> ScienceFactor -> savantures
```

- $rev(\mathcal{G}_{500})$:

```
degré max 31
degré moyen 0.430000
max_accessibles savantures avec 76 sommets
Diamètre 7 réalisé par :
savantures -> ScienceFactor -> hypothesesorg ->
    fath_gabrielle -> MooreInst -> faezeh_db -> naxonlabs -> Isaac__K
```

- $\mathcal{G}_{500}^{-}$:

```
degré max 4
degré moyen 0.176000
max_accessibles SeverineWozniak avec 8 sommets
Diamètre 4 réalisé par :
QLMB8mars -> giu_sapio -> louise_tbr -> GroupeImpec -> halfbloodqueenx
```

- $\mathcal{G}_{500}^{+}$:

```
degré max 32
degré moyen 0.684000
max_accessibles helloselyn avec 98 sommets
Diamètre 12 réalisé par :
TsamiyahL -> FES_AFNEUS -> FlorestanAFNEUS -> FedeAddiction ->
    LS46151053 -> hypothesesorg -> Osec2022 -> ardakaniz ->
    ValRobert974 -> DialloAIbrahim2 -> Defense137 -> KArthemis ->
    SGF_GEOSOC
```

- $\mathcal{G}$:

```
degré max 950
degré moyen 36.266096
max_accessibles Boris_Brana avec 6049 sommets
Diamètre 9 réalisé par :
MonaEmara10 -> SambitPhD -> MIT_CSAIL -> MehdiKaytoue -> gromuald ->
    ECHARDE_ENSL -> cerseilia_ -> dadoyeldado -> Deccefunjoogu -> stoicsalik
```

- $rev(\mathcal{G})$:

```
degré max 3655
degré moyen 36.266096
max_accessibles JustVonBraun avec 7532 sommets
Diamètre 9 réalisé par :
Bonusbasci -> TCebere -> Miruna_Rosca -> h2020prometheus ->
    barENdSonLyon -> INP_CNRS -> ThierryCoulhon -> Phil_Baty ->
    HigherEdFutures -> HEMobilities
```

- $\mathcal{G}^{-}$:

```
degré max 610
degré moyen 12.069850
max_accessibles augabcoh avec 5352 sommets
Diamètre 10 réalisé par :
LeaLescouzeres -> Gauthier_tls -> MorganeBoulch -> CSNB14 ->
    leo_chapuis -> CCILYONMETRO -> IsabelleHuault -> Phil_Baty ->
    UNIKEhighered -> HigherEdFutures -> HEMobilities
```

- $\mathcal{G}^+$:

```
degré max 3655
degré moyen 60.462343
max_accessibles augabcoh avec 7854 sommets
Diamètre 7 réalisé par :
GabrielMarseres -> caroched -> MarieMoroso -> L3vironaute -> najatvb ->
    LeankonCarotte -> JustVonBraun -> Sardine49160063
```

32.2.6 Aller plus loin

On propose ici plusieurs pistes de réflexion pour prolonger le TP :

- On a vu des algorithmes de dessin de graphes adaptés à des petits graphes. La présence de l'interaction sommet-sommet semble leur donner une complexité en $O(n^2)$ qui est rédhibitoire ici. Cependant, des sommets éloignés ont peu de chance d'interagir, comment pourrait-on modifier l'algorithme pour ignorer les interactions de répulsion entre sommets éloignés ? On remarque que la distance n'est pas un critère valide car les sommets peuvent tous être superposés. Une manière de traiter cela efficacement est de découper le plan en régions par des droites successives. Allez voir la page Binary Space Partitioning et en déduire un algorithme effectif de dessin de graphe adapté.
- Pour estimer l'importance d'un compte, on ne peut pas se fier à son degré. En effet, celui-ci peut être augmenté artificiellement. Une manière fiable de mesurer l'importance est d'imaginer quelqu'un naviguant aléatoirement sur des comptes en suivant des liens d'abonnement et de mesurer la probabilité qu'il se retrouve sur un compte donné. C'est le principe qui est à la base de l'algorithme PageRank utilisé par Google. Implémenter cet algorithme et en déduire les comptes les plus importants dans cet exemple.

32.3 Plus courts chemins en OCaml

Ce TP vous demande plus d'autonomie que les TP précédents. Il s'agit de mobiliser des connaissances plus anciennes et d'implémenter effectivement des algorithmes que l'on comprend bien en pseudo-code.

L'objectif est de pouvoir résoudre des problèmes comme :

- HIGHWAYS
- MICEMAZE
- TRAFFICN
- SAMER08A

Si vous êtes courageux, foncez sans lire la suite ;-)

32.3.1 Écriture naïve de Dijkstra

Écrire une implémentation de Dijkstra reposant sur une référence sur une liste de sommets pour effectuer la recherche du sommet de plus petite priorité. On pourra s'inspirer du programme `Python` donné dans le cours, à la section Algorithme de Dijkstra.

On considérera, comme pour les autres parcours, que le graphe est donné sous la forme de listes d'adjacence mais cette fois, les listes devront comporter les poids. En supposant que $S = \{0, 1, ..., n-1\}$ on utilisera directement le type suivant en supposant des poids entiers positifs :

```
type graph = (int * int) list array
```

Pour un sommet x on a donc une liste de couples (y, w), un pour chaque arête $x \to y$, avec $\pi(x \to y) = w$.

Ainsi, le graphe suivant :

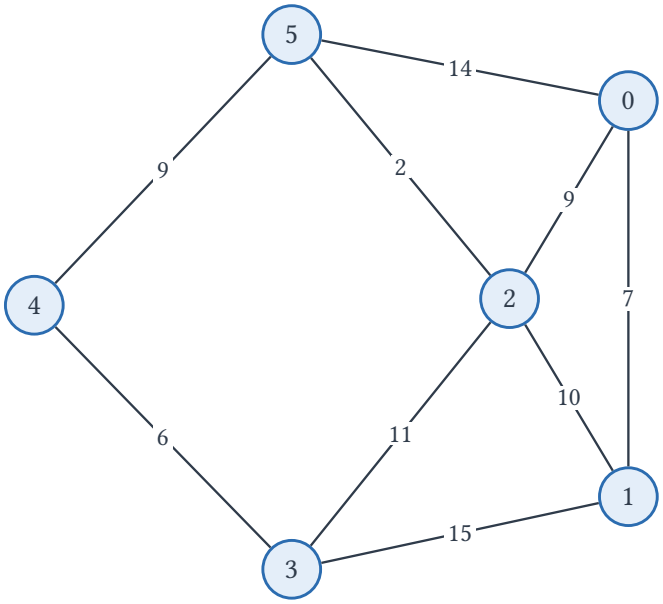


Figure 32.3 — Graphe non orienté pondéré à six sommets.

est donné par la valeur :

```
let g = [|
    [ (1, 7); (2, 9); (5, 14) ];
    [ (0, 7); (2, 10); (3, 15) ];
    [ (0, 9); (1, 10); (3, 11); (5, 2) ];
    [ (1, 15); (2, 11); (4, 6) ];
    [ (3, 6); (5, 9) ];
    [ (0, 14); (2, 2); (4, 9) ]
|]
```

Pour le tableau des distances, on pourra définir un `type poids = I | N of int` ou considérer pour l'infini un entier plus grand que tous les poids de chemins, comme la somme des poids de toutes les arêtes plus 1, ou encore `max_int` en faisant attention aux additions.

Corrigé

```
let rec extrait_plus_petit l d =
    match l with
    | [] -> failwith "impossible"
    | [x] -> (x, [])
    | x::q ->
        let (y, q') = extrait_plus_petit q d in
        if d.(y) < d.(x)
        then (y, x :: q')
        else (x, q)

let dijkstra g src =
    let n = Array.length g in
    let inf = max_int in
    let d = Array.make n inf in
    d.(src) <- 0;
    let marques = Array.make n false in
    let a_traiter = ref [ src ] in
    while !a_traiter <> [] do
        let x, q = extrait_plus_petit !a_traiter d in
        a_traiter := q;
        if not marques.(x)
        then begin
```

```
            marques.(x) <- true;
            List.iter (fun (y, poids) ->
                let n_d = d.(x) + poids in
                if n_d < d.(y)
                then begin
                    d.(y) <- n_d;
                    a_traiter := y :: !a_traiter
                end
            ) g.(x)
        end
    done;
    d
```

Sur le graphe précédent, `dijkstra g 0` renvoie `[|0; 7; 9; 20; 20; 11|]`.

32.3.2 Réalisation d'une file de min-priorité

Reprendre l'implémentation des files de priorité du TP sur les tas et files de priorité en faisant en sorte de gérer des couples (*sommet*, *distance*) et en faisant attention au fait que *distance* peut être ∞.

Attention il s'agit ici de file de priorité minimale !

Remarque 32.3 Ici, on peut aussi considérer directement la file de priorité maximale et ajouter des opposés.

Corrigé

On utilise ici un tas max et des priorités opposées. Les sommets à distance infinie ne sont jamais ajoutés dans la file.

```
module Heap = struct
    type 'a t = {
        elts : 'a array;
        mutable length : int
    }

    let make size_max default =
        {
            elts = Array.make size_max default;
            length = 0
        }

    let length heap = heap.length

    let father a =
        if a = 0
        then failwith "root"
        else (a+1)/2 - 1

    let left a = 2*(a+1) - 1
    let right a = 2*(a+1)

    let nth heap i = heap.elts.(i)

    let swap v x y =
        let t = v.(x) in
        v.(x) <- v.(y);
        v.(y) <- t

    exception Overflow
    let add heap x =
        if heap.length = Array.length heap.elts
        then raise Overflow;
        let pos = ref heap.length in
        heap.elts.(heap.length) <- x;
        heap.length <- heap.length + 1;
```

```
        while !pos > 0 && nth heap (father !pos) < nth heap !pos do
            let p = father !pos in
            swap heap.elts p !pos;
            pos := p
        done

    let max_with_sons heap a =
        let current_max = ref a in
        if heap.length > left a
            && nth heap (left a) > nth heap !current_max
        then current_max := left a;
        if heap.length > right a
            && nth heap (right a) > nth heap !current_max
        then current_max := right a;
        !current_max

    exception Empty
    let take heap =
        if heap.length = 0
        then raise Empty;
        heap.length <- heap.length - 1;
        let x = heap.elts.(0) in
        if heap.length > 0
        then begin
            let pos = ref 0 in
            heap.elts.(0) <- heap.elts.(heap.length);
            while max_with_sons heap !pos <> !pos do
                let max_sons = max_with_sons heap !pos in
                swap heap.elts !pos max_sons;
                pos := max_sons
            done
        end;
        x

    let peek heap =
        if heap.length = 0
        then raise Empty;
        heap.elts.(0)
end

module PriorityQueue = struct
    type 'a t = (int * 'a) Heap.t

    let make size_max default =
        Heap.make size_max (0, default)

    let length q = Heap.length q
    let add q x p = Heap.add q (p, x)
    let take_full q = Heap.take q
    let take q = snd (take_full q)
    let peek q = snd (Heap.peek q)
    let empty q = length q = 0
end
```

32.3.3 Écriture de Dijkstra efficace

Utiliser la file de priorité pour en déduire une implémentation efficace.

Corrigé

Un sommet peut être ajouté plusieurs fois dans la file (une fois par amélioration de sa distance) : les anciennes copies sont ignorées grâce au tableau `marques`. Il y a au plus $|A|+1$ ajouts, ce qui donne la capacité nécessaire et une complexité en $O((|S|+|A|)\log|A|)$.

```
let dijkstra g src =
    let n = Array.length g in
    let d = Array.make n max_int in
    d.(src) <- 0;
    let marques = Array.make n false in
    (* chaque arête peut provoquer un ajout : il faut une
       capacité de |A| + 1 *)
    let m = Array.fold_left (fun s l -> s + List.length l) 0 g in
    let a_traiter = PriorityQueue.make (m + 1) min_int in
    PriorityQueue.add a_traiter src 0;
    while not (PriorityQueue.empty a_traiter) do
        let x = PriorityQueue.take a_traiter in
        if not marques.(x)
        then begin
            marques.(x) <- true;
            List.iter (fun (y, poids) ->
                let n_d = d.(x) + poids in
                if n_d < d.(y)
                then begin
                    d.(y) <- n_d;
                    PriorityQueue.add a_traiter y (-n_d)
                end
            ) g.(x)
        end
    done;
    d
```

32.3.4 Floyd-Warshall

Lire et implémenter l'algorithme de Floyd-Warshall vu dans le cours. On représentera ici le graphe par sa matrice des poids, où `max_int` code l'absence d'arête.

Corrigé

```
let (+$) a b =
    (* une addition pour avoir max_int + * = max_int *)
    if a = max_int || b = max_int
    then max_int
    else a+b

let floyd_warshall poids =
    let n = Array.length poids in
    let m = Array.make_matrix n n max_int in
    for i = 0 to n-1 do
        for j = 0 to n-1 do
            if i = j
            then m.(i).(i) <- 0
            else m.(i).(j) <- poids.(i).(j)
        done
    done;
    for k = 0 to n-1 do
        for i = 0 to n-1 do
            for j = 0 to n-1 do
                m.(i).(j) <-
                    min m.(i).(j) (m.(i).(k) +$ m.(k).(j))
            done
        done
    done;
    m
```

qu'on peut raffiner pour avoir les liens de parenté : `parent.(i).(j)` est le prédécesseur de `j` sur un plus court chemin de `i` à `j`.

```
let floyd_warshall poids =
    let n = Array.length poids in
```

```
    let m = Array.make_matrix n n max_int in
    let parent = Array.make_matrix n n None in
    for i = 0 to n-1 do
        for j = 0 to n-1 do
            if i = j
            then m.(i).(i) <- 0
            else begin
                m.(i).(j) <- poids.(i).(j);
                if poids.(i).(j) <> max_int
                then parent.(i).(j) <- Some i
            end
        done
    done;
    for k = 0 to n-1 do
        for i = 0 to n-1 do
            for j = 0 to n-1 do
                let par_k = m.(i).(k) +$ m.(k).(j) in
                if par_k < m.(i).(j)
                then begin
                    m.(i).(j) <- par_k;
                    parent.(i).(j) <- parent.(k).(j)
                end
            done
        done
    done;
    m, parent
```

32.3.5 Problèmes

Résoudre les problèmes donnés au-dessus. On pourra utiliser la fonction suivante en OCaml pour lire des entiers séparés par des espaces sur une ligne :

```
let read_int_list () =
    List.map int_of_string (String.split_on_char ' ' (read_line ()))
```

TRAVAUX PRATIQUES 33

Parcours récursif de graphes

DANS CE CHAPITRE

33.1 Introduction et graphes de tests
33.2 Principe du parcours récursif
33.3 Extraction des composantes connexes
33.4 Chemins
33.5 Autonomie : Kosaraju

33.1 Introduction et graphes de tests

Dans tout ce sujet de TP, on considère des graphes sur $\{0, 1, \ldots, n-1\}$ donnés par leurs listes d'adjacence (voir le chapitre sur les graphes). Un graphe non orienté est représenté par son graphe orienté symétrique : chaque arête $x - y$ apparaît dans la liste de x et dans celle de y.

Ainsi, on a :

```
type graphe = int list array
```

Question 1 Écrire des fonctions

```
val cycle : int -> graphe
val complet : int -> graphe
```

qui renvoient les graphes non orientés C_n (pour $n \geq 3$) et K_n.

Corrigé

```
let cycle n =
    (* on suppose n >= 3 *)
    let v = Array.make n [] in
    for i = 0 to n-1 do
        v.(i) <- [ (i+1) mod n;
            (i+n-1) mod n ]
    done;
    v

let complet n =
    let v = Array.make n [] in
    for i = 0 to n-1 do
        v.(i) <-
            List.init n Fun.id
            |> List.filter ((<>)i)
```

```
    done;
    v
```

Question 2 Écrire des fonctions

```
val vers_bin : int -> int -> bool list
val depuis_bin : bool list -> int
val compte_diff : 'a list -> 'a list -> int
val hypercube : int -> graphe
```

telles que

- `vers_bin n p` renvoie la représentation binaire de `p` sur `n` bits sous forme de liste avec en tête le bit de poids le plus faible (Exemple : `vers_bin 4 14 -> [false;true;true;true]`)
- `depuis_bin l` prend une telle représentation et renvoie l'entier correspondant
- `compte_diff l1 l2` renvoie le nombre d'éléments différents au même rang dans deux listes de **même longueur**
- `hypercube n` renvoie H_n.

Corrigé

```
let rec vers_bin n p =
    if n = 0
    then []
    else (p mod 2 = 1)
        :: vers_bin (n-1) (p / 2)

let rec depuis_bin l =
    match l with
    | [] -> 0
    | t :: q ->
        2 * depuis_bin q
        + (if t then 1 else 0)

let rec pow2 n =
    if n = 0 then 1 else 2 * pow2 (n-1)

let compte_diff l1 l2 =
    List.combine l1 l2
    |> List.filter
        (fun (a,b) -> a <> b)
    |> List.length

let hypercube n =
    let p2 = pow2 n in
    let v = Array.make p2 [] in
    for i = 0 to p2 - 1 do
        for j = i+1 to p2 - 1 do
            let bi, bj =
                vers_bin n i, vers_bin n j in
            if compte_diff bi bj = 1
            then begin
                v.(i) <- j :: v.(i);
                v.(j) <- i :: v.(j)
            end
        done
    done;
    v
```

Question 3 Écrire une fonction

```
val aleatoire : int -> int -> graphe
```

telle que `aleatoire n p` crée un graphe non orienté aléatoire à `n` sommets et `p` arêtes (avec $p \leq \frac{n(n-1)}{2}$), où l'ensemble des `p` arêtes est choisi uniformément parmi les ensembles de `p` arêtes possibles.

Corrigé

On tire une arête uniformément parmi les $\frac{n(n-1)}{2}$ arêtes possibles, et on recommence si elle est déjà présente. Pour tirer `b` uniforme et distinct de `a`, on tire `c` dans $[\![0, n-2]\!]$ et on décale de 1 les valeurs $\geq a$.

```
let aleatoire n p =
    (* on suppose p <= n(n-1)/2 *)
    assert (2 * p <= n * (n-1));
    let v = Array.make n [] in
    let nb = ref 0 in
    while !nb < p do
        (* a et b uniformes et distincts *)
        let a = Random.int n in
        let b =
            let c = Random.int (n-1) in
            if c >= a then c+1 else c in
        (* on recommence si l'arête existe déjà *)
        if not (List.mem b v.(a))
        then begin
            v.(a) <- b :: v.(a);
            v.(b) <- a :: v.(b);
            incr nb
        end
    done;
    v
```

Quand `p` est proche de $\frac{n(n-1)}{2}$, les tirages rejetés deviennent nombreux, mais cela reste raisonnable pour les tailles de ce TP.

33.2 Principe du parcours récursif

Pour explorer un graphe, on va reprendre le principe du parcours récursif vu pour les arbres.

`Parcours(x)` :

- Si `x` n'a pas déjà été traité
  - On traite `x` (appel d'une fonction sur `x`, changement de valeurs...)
  - Pour chaque voisin `y` de `x`
    - On appelle `Parcours(y)`

Ainsi, la fonction `parcours` devra prendre en entrée :

- le graphe sous forme de listes d'adjacence ;
- le sommet de départ `x` ;
- une structure permettant de gérer le parcours pour s'assurer de ne pas traiter deux fois un sommet (un `visites : bool array` suffit) ou pour rajouter des informations supplémentaires.

Question 4 Écrire une fonction

```
val parcours : graphe -> bool array -> int -> unit
```

telle que `parcours g visites x` effectue ce parcours récursif en se contentant de placer `true` dans `visites.(x)` pour le traiter.

Corrigé

```
let rec parcours g visites x =
    if not visites.(x)
    then begin
```

```
        visites.(x) <- true;
        List.iter (parcours g visites) g.(x)
    end
```

Question 5 En déduire une fonction

```
val taille_composante : graphe -> int -> int
```

telle que `taille_composante g x` renvoie le nombre de sommets de la composante connexe de `x` dans un graphe non orienté.

Corrigé

```
let taille_composante g x =
    let n = Array.length g in
    let visites = Array.make n false in
    parcours g visites x;
    Array.map
        (fun x -> if x then 1 else 0)
        visites
        |> Array.fold_left (+) 0
```

Question 6 En déduire une fonction

```
val connexe : graphe -> bool
```

qui indique si un graphe non orienté (ayant au moins un sommet) est connexe.

Corrigé

```
let connexe g =
    taille_composante g 0
        = Array.length g
```

Question 7 On va considérer ici n sommets et, pour chaque p entre 0 et p_{max}, on va compter la proportion C_p de graphes non orientés connexes obtenus pour 100 générations de graphes aléatoires à p arêtes.

Écrire la fonction

```
val proportions : int -> int -> float list
```

telle que `proportions n pmax` renvoie la liste des C_p pour p croissant.

Tracer cela pour différentes valeurs, avec `matplotlib` par exemple.

Corrigé

On s'arrête à $\frac{n(n-1)}{2}$ arêtes si p_{max} est plus grand.

```
let proportions n pmax =
    List.init
        (min pmax (n * (n-1) / 2) + 1)
        (fun p ->
            let r = ref 0 in
            for i = 1 to 100 do
                if connexe (aleatoire n p)
                then incr r
            done;
            float_of_int !r /. 100.)
```

33.3 Extraction des composantes connexes

On peut adapter le parcours précédent pour obtenir l'ensemble des composantes connexes d'un graphe non orienté ainsi :

- On commence à $k = 0$
- Tant que G a un sommet non visité x :
  - On incrémente k
  - On lance un parcours depuis x
  - On attribue à chacun des sommets visités lors de ce parcours la composante connexe d'indice k

On peut se contenter d'un unique tableau pour indiquer la composante connexe d'un sommet tout en permettant de définir la notion de visite : il suffit de mettre 0 à tous en numéro de composante puis de lancer un parcours tant qu'il y a un sommet associé à 0.

Question 8 Écrire une fonction

```
val composantes : graphe -> int list list
```

qui renvoie la liste des composantes connexes écrites sous forme de listes d'entiers (l'ordre importe peu).

Note comme souvent dans ces questions, il va falloir refaire un parcours adapté en fonction auxiliaire.

Corrigé

```
let indices v x =
    let ind = ref [] in
    for i = 0 to Array.length v - 1 do
        if v.(i) = x
        then ind := i :: !ind
    done;
    !ind

let composantes g =
    let n = Array.length g in
    let k = ref 0 in
    let comp = Array.make n 0 in
    let rec dfs x =
        if comp.(x) = 0
        then begin
            comp.(x) <- !k;
            List.iter dfs g.(x)
        end
    in
    for x = 0 to n-1 do
        if comp.(x) = 0
        then begin
            incr k;
            dfs x
        end
    done;
    List.init !k
        (fun k -> indices comp (k+1))
```

33.4 Chemins

Dans un parcours, on dit que le sommet x qui effectue un appel récursif sur un de ses voisins y est le père de y.

Les sommets pour lesquels cela ne se produit pas n'ont pas de père. Parmi ceux-ci le cas du sommet de départ est à part.

Pour obtenir le chemin depuis un sommet de départ vers un sommet z de père défini, il suffit de remonter de père en père jusqu'à retomber sur le sommet de départ.

Question 9 Écrire une fonction

```
val chemin : int option array -> int -> int -> int list option
```

telle que `chemin peres x y` renvoie un chemin de `x` à `y` s'il en existe un, donné par le tableau `peres` des pères.

Corrigé

```
let chemin peres x y =
    let rec aux pre y =
        match peres.(y) with
        | None -> None
        | Some z ->
            if z = x
            then Some (z :: pre)
            else aux (z :: pre) z
    in aux [y] y
```

Question 10 Modifier la fonction de parcours pour obtenir

```
val parcours_ex : graphe -> bool array -> int option array -> int -> unit
```

telle que `parcours_ex g visites peres x` effectue un parcours récursif en remplissant le tableau des pères.

Corrigé

```
let rec parcours_ex g visites peres x =
    if not visites.(x)
    then begin
        visites.(x) <- true;
        List.iter (fun y ->
            if peres.(y) = None
            then peres.(y) <- Some x;
            parcours_ex g visites peres y) g.(x)
    end

(* Qu'on peut utiliser ainsi par exemple : *)
let test_parcours_ex =
    let n = 8 in
    let g = hypercube 3 in
    let peres = Array.make n None in
    let visites = Array.make n false in
    parcours_ex g visites peres 0;
    List.map (chemin peres 0) (List.init n Fun.id)
```

Le sommet de départ x n'a pas de père au début du parcours. S'il possède un prédécesseur z accessible depuis x, la ligne `peres.(y) <- Some x` lui donne le père z quand on examine l'arête $z \to x$: `chemin peres x x` renvoie alors un cycle passant par x. C'est ce qu'on exploite à la question suivante.

Question 11 À l'aide de la fonction précédente, écrire une fonction

```
val cyclique_oriente : graphe -> bool
```

qui indique si un graphe orienté est cyclique.

Écrire une version non orientée en faisant attention aux possibles pièges dans ce cas.

Corrigé

```
(* Une version simple revient à lancer un parcours depuis
   chaque sommet et à étudier l'existence d'un chemin depuis
   ce sommet vers lui même.

   On verra bientôt une version directe depuis le parcours.
*)
let cyclique_oriente g =
    let n = Array.length g in
    let cyclique = ref false in
    for x = 0 to n-1 do
        let peres = Array.make n None in
        let visites = Array.make n false in
        parcours_ex g visites peres x;
        match chemin peres x x with
        | None -> ()
        | Some p -> cyclique := true
    done;
    !cyclique
```

Dans le cas non orienté, la fonction précédente répond toujours `true` dès qu'il y a une arête, puisque $x \to y \to x$ est un cycle du graphe orienté symétrique. Il faut ignorer l'arête qui ramène au père :

```
(* Dans un graphe non orienté, chaque arête x - y donne x -> y et y -> x :
   le chemin x -> y -> x n'est pas un cycle. On détecte un cycle quand on
   rencontre un sommet déjà visité qui n'est pas le père du sommet courant.
   On suppose le graphe sans boucle ni arête multiple. *)
let cyclique_non_oriente g =
    let n = Array.length g in
    let visites = Array.make n false in
    let cyclique = ref false in
    let rec dfs pere x =
        visites.(x) <- true;
        List.iter (fun y ->
            if not visites.(y)
            then dfs x y
            else if y <> pere
            then cyclique := true) g.(x)
    in
    for x = 0 to n-1 do
        if not visites.(x)
        then dfs (-1) x
    done;
    !cyclique
```

33.5 Autonomie : Kosaraju

Question 12 Documentez-vous et implémentez l'algorithme de Kosaraju pour obtenir les composantes **fortement connexes** d'un graphe orienté.

Corrigé

L'algorithme est présenté dans le chapitre sur les graphes avancés et mis en œuvre dans le TP Kosaraju et 2-SAT.

PARTIE IV

Algorithmique

34	Introduction à l'analyse des algorithmes	447
35	Complexité amortie	477
36	Algorithmique exacte	487
37	Recherche par force brute	495
38	Algorithmes gloutons	515
39	Division en sous-problèmes	535
40	Programmation dynamique	551
41	TP Programmation dynamique - TP 2 (Python)	585
42	Algorithmique des textes	593
43	TP Compression de Huffman	651

CHAPITRE 34

Introduction à l'analyse des algorithmes

DANS CE CHAPITRE

34.1 État d'un programme
34.2 Terminaison
34.3 Correction
34.4 Complexité
34.5 Exercices

Remarque 34.1 Ce chapitre présente les trois grands principes qui nous serviront de guide pour analyser les algorithmes et les programmes :

- La **terminaison** : l'algorithme termine-t-il au bout d'un nombre fini d'étapes quelle que soit l'entrée ?
- La **correction** : le résultat rendu est-il celui qui était attendu ?
- La **complexité** : combien de temps prend le programme selon la taille de l'entrée ? Combien d'espace mémoire occupe-t-il ?

Savoir répondre à ces questions, c'est pouvoir prédire, avant d'avoir écrit la moindre ligne de code, ce qui va se passer.

Les notions de variant et d'invariant de boucle ont déjà été rencontrées dans le chapitre Introduction à la programmation, et la terminaison et la correction des fonctions récursives dans le chapitre Récursivité. On les reprend ici de manière plus systématique.

34.1 État d'un programme

Avant de commencer à raisonner sur les algorithmes, il est nécessaire de préciser la notion d'état qui correspond à un instantané de l'environnement d'exécution d'un programme lorsque son exécution est interrompue. Bien entendu, la description complète d'un tel état ne serait pas forcément pertinente, car cela prendrait en compte l'ensemble de la mémoire. Le plus souvent, on considère uniquement ce qui est important pour ce qu'on étudie.

Ainsi, si on considère le programme suivant :

```
int a = 3;
int b = 2;

a = b;
```

L'état du programme juste avant l'instruction `a = b;` (ligne 4) est

variable	valeur
a	3
b	2

et l'état juste après cette instruction est

variable	valeur
a	2
b	2

Parfois, on aura besoin de plus d'information dans l'état comme la position en mémoire de certaines données. Mais assez souvent, pour les algorithmes qui nous intéressent, on pourra adopter un point de vue assez abstrait de l'état d'un algorithme comme étant une fonction partielle des noms vers les valeurs.

Pour la terminaison et la correction, on va considérer des propriétés logiques dépendant de l'état. Par exemple $a \geq 0 \wedge b \geq a$.

34.2 Terminaison

Définition 34.2

On dit qu'un algorithme **termine** quand il n'exécute qu'un nombre fini d'étapes sur toute entrée.

Remarque 34.3 Cela n'empêche pas que ce nombre d'étapes puisse être arbitrairement grand en fonction des entrées.

Un algorithme qui n'utilise ni boucles conditionnelles (boucles `while`) ni récursivité termine toujours. Ainsi, la question de la terminaison n'est à considérer que dans ces deux cas.

Considérons par exemple l'algorithme suivant qui, étant donné un entier naturel n strictement positif, inférieur à 2^{30}, détermine le plus petit entier k tel que $n \leq 2^k$:

OCaml

```
let plus_petite_puissance2 n =
    let k = ref 0 in
    let p = ref 1 in
    while !p < n do
        k := !k + 1;
        p := !p * 2
    done;
    !k
```

C

```
int plus_petite_puissance2(int n)
{
    int k = 0;
    int p = 1;

    while (p < n)
    {
        k = k+1;
        p = p*2;
    }

    return k;
}
```

Python

```
def plus_petite_puissance2(n):
    k = 0
    p = 1

    while p < n:
        k = k+1
        p = p*2

    return k
```

On remarque que `p` prend successivement pour valeurs toutes les puissances de 2 jusqu'à une éventuelle sortie de boucle. Or, il existe une puissance de 2 supérieure ou égale à `n`, donc, une fois atteinte, la condition de la boucle `while` n'est plus remplie et l'algorithme termine.

Remarque 34.4 Prouver la terminaison n'est pas une question facile, elle peut même être insoluble. Par exemple, si on considère le programme suivant :

```
def temps_de_vol(a):
    u = a
    n = 0
    while u != 1:
        if u % 2 == 0:
            u = u // 2
        else:
            u = 3*u + 1
        n = n+1
    return n
```

être capable de prouver sa terminaison pour tout entier $a \geq 1$ revient à prouver la conjecture de Collatz (encore dite de Syracuse).

On peut aussi considérer le tri suivant :

```
def bogo(t):
    while not est_trie(t):
        melange(t)
```

Il semble très improbable que cet algorithme ne termine pas, d'ailleurs on peut prouver qu'il termine avec probabilité 1, mais on ne peut pas exclure le cas où il mélange indéfiniment.

34.2.1 Variant de boucle

Pour prouver la terminaison d'une boucle conditionnelle, on utilise un **variant** de boucle.

Définition 34.5

Un **variant de boucle** est une quantité entière, dépendant de l'état, qui est **positive** à l'entrée de chaque itération de la boucle et qui diminue **strictement** à chaque itération.

Dans l'exemple précédent, la quantité $n - p$ est un variant de boucle :

- Comme il s'agit d'une différence de deux entiers, c'est un entier. Et tant que la condition de boucle est vérifiée, $p < n$ donc $n - p > 0$.
- Lorsqu'on passe d'une itération à la suivante, la quantité passe de $n - p$ à $n - 2p$ or $(n - p) - (n - 2p) = p > 0$ car $p \geq 1$. Il y a bien une stricte diminution.

Remarque 34.6 Ici, en sortie de boucle, $n - p \leq 0$. On fait donc bien attention de préciser que la quantité est positive tant que la condition de boucle est vérifiée.

Si on a un variant de boucle qui vaut initialement N avant d'entrer dans la boucle, celle-ci effectue au plus N itérations, car le variant diminue au moins de 1 à chaque itération tout en restant positif à l'entrée de chacune.

Théorème 34.7

Si une boucle admet un variant de boucle, elle termine.

Preuve Si la boucle effectuait une infinité d'itérations, les valeurs successives du variant à l'entrée de chaque itération formeraient une suite infinie strictement décroissante d'entiers naturels, ce qui est impossible.

□

34.2.2 Exemple de la recherche dichotomique

On considère ici la recherche dichotomique dans un tableau trié d'entiers. Étant donné un tableau t de taille n et un entier x dont on cherche à déterminer la présence dans le tableau entre les indices i et j, on considère l'algorithme suivant :

- Si $i > j$, alors il n'y a pas de sous-tableau et on renvoie `false`.
- Sinon, soit m l'élément d'indice $p = \left\lfloor \frac{i+j}{2} \right\rfloor$.
  - Si $x = m$, on renvoie `true`.
  - Si $x < m$, on continue la recherche dans le sous-tableau des indices i à $p - 1$.
  - Si $x > m$, on continue la recherche dans le sous-tableau des indices $p + 1$ à j.

Les programmes suivants présentent une implémentation de cet algorithme qui renvoie un indice de x plutôt qu'un booléen.

OCaml

```
exception Trouve of int

let rech_dicho t x =
    let i = ref 0 in
    let j = ref (Array.length t - 1) in
    try
        while !i <= !j do
            let p = (!i + !j)/2 in
            let m = t.(p) in
            if m = x
            then raise (Trouve p)
            else if x < m
            then j := p-1
            else i := p+1
        done;
        None
    with Trouve p -> Some p
```

C

```
int rech_dicho(int *t, size_t n, int x)
{
    /* renvoie un indice de x
      si x est dans le tableau et -1 sinon */
    int i = 0;
    int j = (int) n - 1;

    while (i <= j)
    {
        int p = (i+j)/2;
        int m = t[p];

        if (x == m)
        {
            return p;
        }
        else if (x < m)
        {
            j = p-1;
        }
```

```
        else
        {
            i = p+1;
        }
    }

    return -1;
}
```

Python

```
def rech_dicho(t, x):
    i = 0
    j = len(t) - 1

    while i <= j:
        p = (i+j)//2
        m = t[p]
        if x == m:
            return p
        elif x < m:
            j = p-1
        else:
            i = p+1

    return None
```

Remarque 34.8 On aurait pu écrire `i+(j-i)/2` et non `(i+j)/2` afin d'éviter des erreurs de dépassement dans le calcul de `i+j`.

Ici, la terminaison n'est pas immédiate, on va la prouver à l'aide d'un variant de boucle. On considère ainsi la quantité $d(i,j) = j - i$.

- C'est un entier, et la condition de boucle est équivalente à $d(i,j) \geq 0$, donc cette quantité est bien entière et positive à l'entrée de chaque itération.
- Quand on passe à l'itération suivante, on passe
  - soit de $d(i,j)$ à $d(i,p-1)$. Or $d(i,p-1) = p - 1 - i = \left\lfloor \frac{i+j}{2} \right\rfloor - 1 - i < \frac{i+j}{2} - i = \frac{j-i}{2} \leq j - i = d(i,j)$.
  - soit de $d(i,j)$ à $d(p+1,j)$. Or $d(p+1,j) = j - p - 1 = j - \left\lfloor \frac{i+j}{2} \right\rfloor - 1 < j - \frac{i+j}{2} = \frac{j-i}{2} \leq j - i = d(i,j)$, car $\lfloor y \rfloor > y - 1$ pour tout réel y.

  Dans tous les cas, $d(i,j)$ diminue strictement.

Ainsi, il s'agit d'un variant de boucle et l'algorithme termine.

Remarque 34.9 On remarque que le programme récursif suivant réalise également cet algorithme.

OCaml

```
(* rech_dicho : int array -> int -> int -> int
    -> int option *)
let rec rech_dicho t i j x =
    if i <= j
    then let p = (i+j)/2 in
         let m = t.(p) in
         if x = m
         then Some p
         else if x < m
         then rech_dicho t i (p-1) x
         else rech_dicho t (p+1) j x
    else None
```

C

```
int rech_dicho(int *t, int i, int j, int x)
{
```

```
    if (i <= j)
    {
        int p = (i+j)/2;
        int m = t[p];
        if (x == m)
            return p;
        else if (x < m)
            return rech_dicho(t, i, p-1, x);
        else
            return rech_dicho(t, p+1, j, x);
    }

    return -1;
}
```

Python

```
def rech_dicho(t, i, j, x):
    if i <= j:
        p = (i+j)//2
        m = t[p]
        if x == m:
            return p
        elif x < m:
            return rech_dicho(t, i, p-1, x)
        else:
            return rech_dicho(t, p+1, j, x)
    else:
        return None
```

Il suffit alors d'appeler `rech_dicho` sur la plage maximale d'indices (de 0 à $n-1$) pour faire une recherche sur tout le tableau.

Ici, il n'y a pas explicitement de boucle, mais le même principe peut être mis en place pour prouver que le nombre d'appels récursifs est majoré, et donc que toute exécution termine. En effet, la quantité $d(i,j) = j - i$ diminue pour les mêmes raisons à chaque appel récursif et reste entière positive tant qu'on effectue un nouvel appel (voir aussi la partie Terminaison du chapitre sur la récursivité).

34.2.3 Exemple du tri à bulles

On considère le tri à bulles dans une implémentation naïve. On effectue ainsi une série de balayages d'un tableau : on parcourt le tableau de gauche à droite et si deux éléments consécutifs sont dans le désordre, on les permute. À l'issue d'un tel balayage, si on a effectué au moins un échange, on recommence, sinon on s'arrête, car c'est le signe que le tableau est trié, comme on le prouvera à la fin de ce paragraphe.

On obtient alors le code suivant :

OCaml

```
let swap t i j =
    let temp = t.(i) in
    t.(i) <- t.(j);
    t.(j) <- temp

let tri_bulles t =
    let echange = ref true in
    while !echange do
        echange := false;
        for i = 0 to Array.length t - 2 do
            if t.(i) > t.(i+1)
            then begin
                swap t i (i+1);
                echange := true
            end
        done
    done
```

C

```c
void swap(int t[], int i, int j)
{
    int temp = t[i];
    t[i] = t[j];
    t[j] = temp;
}

void tri_bulles(int t[], int nb)
{
    bool echange = true;
    while (echange)
    {
        echange = false;
        for (int i = 0; i < nb-1; i++)
        {
            if (t[i] > t[i+1])
            {
                echange = true;
                swap(t, i, i+1);
            }
        }
    }
}
```

Python

```python
def swap(t, i, j):
    temp = t[i]
    t[i] = t[j]
    t[j] = temp

def tri_bulles(t):
    echange = True
    while echange:
        echange = False
        for i in range(len(t)-1):
            if t[i] > t[i+1]:
                echange = True
                swap(t, i, i+1)
```

Si la question de la terminaison de la boucle `for` ne se pose pas, celle de la boucle `while` mérite qu'on s'y attarde pour deux raisons :

- premièrement pour la terminaison elle-même ;
- deuxièmement pour savoir si on peut majorer le nombre d'itérations, une question qu'on reverra avec la question de la complexité.

On va prouver la terminaison pour un type d'algorithme de tri qui généralise le tri à bulles. Pour cela, on commence par définir la notion d'inversion pour un tableau t de $|\,t\,|$ éléments :

Définition 34.10

On dit qu'un couple $(i,j) \in [\![0, |\,t\,| - 1]\!]^2$ est une **inversion** pour t si $i < j$ et $t[i] > t[j]$.

On note $Inv(t)$ l'ensemble des inversions de t et $inv(t) = |\,Inv(t)\,|$ le nombre d'inversions.

On dira qu'une inversion de la forme $(i, i+1)$ est une **inversion directe**.

On considère donc un algorithme qui résout une inversion directe tant qu'il y en a au moins une.

```python
def tri(t):
    while existe_inversion_directe(t):
        i = debut_d_une_inversion_directe(t)
        echange(t, i, i+1)
```

Le tri à bulles est alors une manière de réaliser l'algorithme précédent en résolvant les inversions directes rencontrées alors qu'on avance dans le tableau.

Rappelons que t est **trié dans l'ordre croissant** lorsque pour tous indices $i < j$, on a $t[i] \leq t[j]$.

Théorème 34.11

Pour un tableau t les propositions suivantes sont équivalentes :

1. t est trié dans l'ordre croissant ;
2. t n'a pas d'inversions ;
3. t n'a pas d'inversions directes.

Preuve

- 1) $\Rightarrow$ 3) naturellement, si t est trié dans l'ordre croissant, $i < i+1$ entraîne $t[i] \leq t[i+1]$, donc il ne peut pas y avoir d'inversions directes.
- 3) $\Rightarrow$ 2) on a $t[i] \leq t[i+1]$ pour tout $i < |\, t \,| - 1$. Soit $i < j$, on va raisonner par récurrence sur $j - i \geq 1$ pour montrer que $t[i] \leq t[j]$.
  - si $j - i = 1$, comme $(i, i+1)$ n'est pas une inversion directe, $t[i] \leq t[j]$.
  - si $j - i > 1$, l'hypothèse de récurrence appliquée au couple $(i, j-1)$, qui vérifie $1 \leq (j-1) - i < j - i$, donne $t[i] \leq t[j-1]$, et $t[j-1] \leq t[j]$ car $(j-1, j)$ n'est pas une inversion directe. Donc $t[i] \leq t[j]$.

  Ainsi, dans tous les cas on a bien $t[i] \leq t[j]$, et (i, j) n'est pas une inversion.
- 2) $\Rightarrow$ 1) si $i < j$, comme t n'a pas d'inversions, on a $t[i] \not> t[j]$, c'est-à-dire $t[i] \leq t[j]$. Ainsi, le tableau est bien trié dans l'ordre croissant.

□

On peut alors démontrer le lemme suivant :

Lemme 34.12

Si t est un tableau présentant une inversion directe $(i, i+1)$ et t' est le tableau qui s'en déduit en échangant les éléments d'indices i et $i+1$, alors $inv(t') = inv(t) - 1$.

Preuve Notons σ la permutation des indices qui échange i et $i+1$ et laisse les autres indices fixes, de sorte que $t'[k] = t[\sigma(k)]$ pour tout k.

- $(i, i+1)$ est une inversion de t mais pas de t', car $t'[i] = t[i+1] < t[i] = t'[i+1]$.
- Soit (a, b) un couple avec $a < b$ et $(a, b) \neq (i, i+1)$. Comme a et b ne sont pas égaux à i et $i+1$ simultanément, on a encore $\sigma(a) < \sigma(b)$; de plus $t'[a] > t'[b] \Leftrightarrow t[\sigma(a)] > t[\sigma(b)]$. Ainsi, $(a, b) \mapsto (\sigma(a), \sigma(b))$ est une bijection de $Inv(t') \setminus \{(i, i+1)\} = Inv(t')$ sur $Inv(t) \setminus \{(i, i+1)\}$ (c'est une involution).

On a donc $inv(t') = inv(t) - 1$.

□

Remarque 34.13 Le nombre d'inversions **directes**, lui, ne diminue pas forcément : en échangeant les deux derniers éléments de $[2, 3, 1]$, qui a une seule inversion directe, on obtient $[2, 1, 3]$ qui en a encore une. C'est pour cela qu'on raisonne sur l'ensemble de toutes les inversions.

On en déduit alors :

Théorème 34.14

L'algorithme `tri` et le tri à bulles terminent et le tableau obtenu est trié. De plus, `tri` effectue exactement $inv(t)$ échanges et la boucle `while` du tri à bulles effectue au plus $inv(t) + 1 \leq \frac{n(n-1)}{2} + 1$ itérations, où t est le tableau initial et $n = |\, t \,|$.

✍ **Preuve** Dans les deux cas, le tableau n'est modifié que par des échanges d'éléments formant une inversion directe : d'après le lemme, chacun de ces échanges fait diminuer *inv* de exactement 1.

- Pour `tri`, la quantité $inv(t)$ est un entier positif qui diminue de 1 à chaque itération : c'est un variant, l'algorithme termine après exactement $inv(t)$ itérations (initialement), et en sortie de boucle il n'y a plus d'inversion directe, donc le tableau est trié d'après le théorème précédent.
- Pour le tri à bulles, chaque itération de la boucle `while` sauf la dernière effectue au moins un échange. La quantité $inv(t) + [\texttt{echange}]$, où $[\texttt{echange}]$ vaut 1 si `echange` est vrai et 0 sinon, est donc un variant : à l'entrée d'une itération elle vaut $inv(t) + 1 \geq 1$, et à la fin de l'itération soit au moins un échange a eu lieu et $inv(t)$ a diminué d'au moins 1 alors que `echange` est encore vrai, soit aucun échange n'a eu lieu et `echange` est passé à faux. Comme elle vaut au plus $inv(t_0) + 1$ initialement, il y a au plus $inv(t_0) + 1$ itérations.

  En sortie de boucle, `echange` est faux : le dernier balayage n'a effectué aucun échange, donc n'a pas modifié le tableau, et il a vérifié $t[i] \leq t[i+1]$ pour tout i. Le tableau n'a pas d'inversion directe, il est donc trié.

Enfin, $inv(t)$ est majoré par le nombre de couples (i, j) avec $i < j$, c'est-à-dire $\binom{n}{2} = \frac{n(n-1)}{2}$.

□

34.3 Correction

Pour parler de correction d'un algorithme, il est nécessaire d'identifier précisément ce qui doit être calculé par l'algorithme. Pour cela, on considère ici informellement des spécifications dépendant des entrées et du résultat de l'algorithme. On verra dans le chapitre sur la logique qu'il s'agit ici de prédicats logiques.

Voici des exemples de spécifications :

- le tableau t en sortie est trié dans l'ordre croissant et contient les mêmes éléments qu'en entrée ;
- la valeur renvoyée est le plus petit indice de x dans le tableau ou -1 s'il ne le contient pas.

Définition 34.15

Un algorithme est **correct** vis-à-vis d'une spécification lorsque, quelle que soit l'entrée,

- il **termine** ;
- le résultat renvoyé vérifie la spécification.

On considère également la correction **partielle**, qui ne demande rien quand l'algorithme ne termine pas :

Définition 34.16

Un algorithme est **partiellement correct** vis-à-vis d'une spécification lorsque, pour toute entrée sur laquelle il termine, le résultat renvoyé vérifie la spécification.

Ainsi, un algorithme est correct si et seulement s'il termine et est partiellement correct.

34.3.1 Invariant de boucle

Définition 34.17

On considère ici une boucle (conditionnelle ou non).

Un prédicat est appelé un **invariant de boucle** lorsque

- il est vérifié avant d'entrer dans la boucle ;
- s'il est vérifié en entrée d'une itération, il est vérifié en sortie de celle-ci.

Quand la boucle termine, on déduit alors, par récurrence sur le nombre d'itérations, que l'invariant est vérifié en sortie de boucle. On cherche donc un invariant qui permette de garantir la spécification en sortie de boucle.

Remarque 34.18 Pour les boucles inconditionnelles, il y a une gestion implicite de l'indice de boucle qui va se retrouver dans l'invariant. On peut alors considérer que la sortie de boucle s'effectue après être passé à l'indice suivant.

Dans le cas d'une boucle conditionnelle portant sur la condition P et ayant un invariant de boucle I, en sortie le prédicat $\neg P \wedge I$ (non P et I) sera vérifié.

On peut illustrer cela en reprenant la fonction `plus_petite_puissance2` vue à la partie Terminaison. On considère ici le prédicat $I(k,p) \coloneqq 2^{k-1} < n$ et $p = 2^k$.

- En entrée de boucle, $k = 0$ et $p = 1 = 2^0$, et on a bien $2^{-1} < n$ car $n \geq 1$.
- Si le prédicat est vérifié en entrée d'itération, on a $p = 2^k$ et, comme on est entré dans cette itération, $p = 2^k < n$. Donc en sortie d'itération on aura bien $I(k+1, 2p)$, car $2p = 2^{k+1}$ et $2^{(k+1)-1} < n$.

Ainsi, ce prédicat est bien un invariant et en sortie de boucle (ce qui arrive nécessairement, car l'algorithme termine), on a le prédicat $I(k,p)$, qui signifie que $2^{k-1} < n$ et $p = 2^k$, **et la condition de sortie de boucle** $p \geq n$, c'est-à-dire $n \leq 2^k$.

La valeur renvoyée est bien l'entier k tel que $2^{k-1} < n \leq 2^k$, c'est-à-dire le plus petit entier k tel que $n \leq 2^k$, ce qui était la spécification annoncée du programme.

✍ **Exemple 34.19**

Prouvons la correction de la recherche dichotomique (version itérative) vue dans la partie Exemple de la recherche dichotomique : si t est trié dans l'ordre croissant, la fonction renvoie un indice p tel que $t[p] = x$ si x apparaît dans t, et `-1` (ou `None`) sinon.

On considère l'invariant $I(i,j) \coloneqq$ « si x apparaît dans t, alors il apparaît dans $t[i..j]$ », où $t[i..j]$ désigne le sous-tableau des indices i à j (vide si $i > j$).

- *Initialisation* : avec $i = 0$ et $j = n-1$, $t[i..j]$ est le tableau entier.
- *Hérédité* : si l'itération ne renvoie pas de valeur, alors $x \neq m = t[p]$. Si $x < m$, comme t est trié, pour tout $k \geq p$ on a $t[k] \geq m > x$; donc si x apparaît dans $t[i..j]$, il apparaît dans $t[i..p-1]$, et on a $I(i, p-1)$. Le cas $x > m$ est symétrique et donne $I(p+1, j)$.

Si la fonction renvoie p au cours d'une itération, c'est que $t[p] = x$. Sinon, en sortie de boucle on a $i > j$: le sous-tableau $t[i..j]$ est vide et l'invariant assure que x n'apparaît pas dans t, donc renvoyer `-1` est correct. Avec la terminaison déjà prouvée, la fonction est correcte.

34.3.2 Exemple du tri par sélection

Le programme suivant présente un algorithme de tri, appelé le *tri par sélection*, dont on va analyser la correction puis la complexité. Il s'agit d'un tri qui repose sur un principe simple : on va chercher le plus petit élément du tableau restant à trier et le placer à la position courante. On définit ainsi trois fonctions :

- `echange` (ou `swap`) réalise l'échange de valeurs entre deux cases du tableau ;
- `indice_minimum` renvoie l'indice de la plus petite valeur entre deux indices donnés ;
- `tri_par_selection` réalise le tri en parcourant le tableau du premier au dernier indice et en plaçant à la position courante le minimum restant.

OCaml

```
let swap t i j =
    let temp = t.(i) in
    t.(i) <- t.(j);
    t.(j) <- temp

let indice_minimum t min_indice max_indice =
    let i = ref min_indice in
    for j = min_indice + 1 to max_indice do
        if t.(j) < t.(!i)
        then i := j
```

```
    done;
    !i

let tri_par_selection t =
    let n = Array.length t in
    for i = 0 to n - 1 do
        swap t i (indice_minimum t i (n-1))
    done
```

C

```
void echange(int *tableau, int i, int j)
{
    int temp = tableau[i];
    tableau[i] = tableau[j];
    tableau[j] = temp;
}

int indice_minimum(int *tableau,
      int min_indice, int max_indice)
{
    int i = min_indice;

    for (int j = min_indice + 1;
         j <= max_indice; j++)
    {
        if (tableau[j] < tableau[i])
            i = j;
    }

    return i;
}

void tri_par_selection(int *tableau, int taille)
{
    for (int i = 0; i < taille; i++)
    {
        echange(tableau, i,
          indice_minimum(tableau, i, taille-1));
    }
}
```

Python

```
def swap(t, i, j):
    temp = t[i]
    t[i] = t[j]
    t[j] = temp

def indice_minimum(t, min_indice, max_indice):
    i = min_indice
    for j in range(min_indice+1, max_indice+1):
        if t[j] < t[i]:
            i = j
    return i

def tri_par_selection(t):
    for i in range(0, len(t)-1):
        j = indice_minimum(t, i, len(t)-1)
        swap(t, i, j)
```

Il n'y a pas de problèmes de terminaison ici, car toutes les boucles sont inconditionnelles. Pour prouver sa correction, on va considérer séparément les deux boucles.

- Boucle dans `indice_minimum` : on va valider l'invariant $I(i,j) \coloneqq \text{min\_indice} \leq i < j$ et $\forall k \in [\![\text{min\_indice}, j-1]\!]$, tableau $[i] \leq$ tableau $[k]$, où j est l'indice de l'itération qui commence.
  - En entrée de boucle, on a $I(\text{min\_indice}, \text{min\_indice}+1)$ vérifié directement.

- Si en entrée d'itération, $I(i,j)$ est vérifié, ce qui signifie que tableau $[i]$ est plus petit que les valeurs comprises entre les indices min_indice et $j-1$, alors on distingue deux cas :
  - soit tableau $[j]$ < tableau $[i]$ et alors en sortie i devient $i'=j$. On a alors tableau $[i']$ = tableau $[j]$ < tableau $[i]$ ≤ tableau $[k]$ pour $k \in [\![\text{min\_indice}, j-1]\!]$. Donc $I(i', j+1)$ est vérifié.
  - soit tableau $[i]$ ≤ tableau $[j]$ et ainsi on a pu prolonger le prédicat à $I(i, j+1)$.

  Ce prédicat est bien un invariant. Ainsi, en sortie de boucle, et donc avant de renvoyer sa valeur, on a bien $I(i, \text{max\_indice}+1)$, donc tableau $[i]$ est la plus petite valeur du sous-tableau des indices min_indice à max_indice. C'est encore vrai si min_indice = max_indice, la boucle n'effectuant alors aucune itération.
- Boucle dans `tri_par_selection` : on va valider l'invariant $T(i)$:= le sous-tableau `tableau[0..i-1]` des indices 0 à $i-1$ est trié et ne contient que des valeurs plus petites que celles du sous-tableau `tableau[i..taille-1]`.
  - En entrée de boucle, le sous-tableau est vide donc $T(0)$ est vérifié.
  - Si en entrée d'itération, le prédicat est vérifié, on récupère l'indice j du minimum du sous-tableau `tableau[i..taille-1]` à l'aide de la fonction `indice_minimum`. Par hypothèse, `tableau[j]` est alors supérieur ou égal à chaque élément de `tableau[0..i-1]` ; en le plaçant à l'indice i, on a bien `tableau[0..i]` qui est trié et, par construction, la valeur de `tableau[i]` est inférieure à toutes celles de `tableau[i+1..taille-1]`. On a ainsi $T(i+1)$ vérifié en sortie d'itération.

  Ce prédicat est bien un invariant. Ainsi, en sortie de boucle, $T(\text{taille})$ est vérifié : le tableau est trié. Comme il n'a été modifié que par des échanges, il contient les mêmes éléments qu'au départ.

Remarque 34.20 La version Python s'arrête à l'indice $n-2$: en sortie de boucle on a alors $T(n-1)$, ce qui suffit puisque le dernier élément est supérieur ou égal à tous ceux de `t[0..n-2]`, qui est trié.

34.4 Complexité

34.4.1 Complexité dans le pire des cas

Considérons un algorithme pour lequel on peut associer à chaque entrée une notion de taille (par exemple le nombre d'éléments d'un tableau). Pour $n \in \mathbb{N}$, on note ainsi I_n l'ensemble des entrées de taille n pour cet algorithme. Pour une entrée e, on note $t(e)$ le temps pris, par exemple en secondes, par l'algorithme sur l'entrée e. De même, on note $s(e)$ l'espace mémoire maximal, par exemple en octets, occupé par l'algorithme au cours de cette exécution **sans compter la taille des entrées**.

Définition 34.21

On appelle :

- **complexité temporelle dans le pire des cas**, la suite $(C_n^t)_{n\in\mathbb{N}}$ telle que pour tout $n \in \mathbb{N}$, $C_n^t = \max_{e\in I_n} t(e)$;
- **complexité spatiale dans le pire des cas**, la suite $(C_n^s)_{n\in\mathbb{N}}$ telle que pour tout $n \in \mathbb{N}$, $C_n^s = \max_{e\in I_n} s(e)$.

Si I_n est infini, on remplace le maximum par une borne supérieure.

Comme on va le voir, calculer explicitement ces suites n'a pas beaucoup d'intérêt tant elles sont dépendantes de la manière dont on mesure le temps et l'espace. Ce qui compte ici, c'est de connaître l'*ordre de grandeur* de ces complexités en fonction de n.

Considérons par exemple la fonction suivante qui renvoie le maximum d'un tableau non vide.

OCaml

```
let maximum t =
    let m = ref t.(0) in
    for i = 1 to Array.length t - 1 do
        if !m < t.(i)
        then m := t.(i)
```

```
    done;
    !m
```

C

```c
int maximum(int *tableau, int taille)
{
    int M = INT_MIN; /* nécessite limits.h */

    for (int i = 0; i < taille; i++)
    {
        if (M < tableau[i])
            M = tableau[i];
    }

    return M;
}
```

Python

```python
def maximum(t):
    m = t[0]
    for i in range(1, len(t)):
        if m < t[i]:
            m = t[i]
    return m
```

Pour un tableau de taille n, ce programme va effectuer n itérations (ou $n-1$ selon la version) et sa complexité est ainsi de l'ordre de n. Il est possible d'être très précis en considérant les temps pris

- pour mettre en place l'appel de fonction et le passage des arguments ;
- par la gestion de l'indice de la boucle `for` ;
- pour la comparaison, puis pour l'affectation éventuelle ;
- pour mettre en place la valeur de retour afin que le résultat soit lu.

On peut remarquer que la notion de pire cas dépend de la précision à laquelle on se place. Ici, si on ne s'intéresse qu'à l'ordre de grandeur, tous les tableaux de taille n sont équivalents. Par contre, si on cherche avec précision le pire cas, il est atteint avec un tableau trié par ordre strictement croissant, car c'est le cas qui effectue une affectation à chaque itération.

34.4.2 Comparer des complexités

Avant de pouvoir comparer les complexités des algorithmes ou des programmes, il est nécessaire de mettre en place des outils pour en parler à la fois avec précision mais également sans rentrer dans des détails d'implémentation non pertinents.

En effet, comparons les deux fonctions suivantes permettant de chercher un élément dans un tableau.

A - Sortie prématurée

OCaml

```
let recherche t elem =
    let ind = ref None in
    let i = ref 0 in
    let n = Array.length t in
    while !ind = None && !i < n do
        if t.(!i) = elem
        then ind := Some !i;
        incr i
    done;
    !ind
```

C

```c
int recherche(int *tab, size_t nb, int elem)
{
    for (size_t i = 0; i < nb; i++)
    {
        if (elem == tab[i])
            return i;
    }

    return -1;
}
```

Python

```python
def recherche(t, elem):
    for i in range(len(t)):
        if t[i] == elem:
            return i
    return None
```

B - Parcours complet

OCaml

```ocaml
let recherche t elem =
    let ind = ref None in
    for i = 0 to Array.length t - 1 do
        if !ind = None && t.(i) = elem
        then ind := Some i
    done;
    !ind
```

C

```c
int recherche(int *tab, size_t nb, int elem)
{
    int ind = -1;

    for (size_t i = 0; i < nb; i++)
    {
        if (ind == -1 && elem == tab[i])
            ind = i;
    }

    return ind;
}
```

Python

```python
def recherche(t, elem):
    ind = None
    for i in range(len(t)):
        if ind is None and t[i] == elem:
            ind = i
    return ind
```

La fonction **B** semble moins efficace que la fonction **A**, car la fonction **A** sort tout de suite dès qu'on a trouvé l'élément alors que la seconde continue à parcourir `t`.

Mais on doit se poser la question de la pertinence de cette optimisation selon le pire des cas. Ici, le pire des cas correspond à ne pas avoir `elem` dans le tableau, et à ce moment-là les deux fonctions fonctionnent de la même manière.

> **Remarque 34.22** En fait, la fonction **A** sera souvent à privilégier, car il est souvent plus facile d'interrompre le flot que de faire en sorte de préserver l'état à la place du `return` pour pouvoir renvoyer la bonne valeur à la fin de la fonction.

De la même manière, il faut déterminer ce que l'on souhaite compter précisément :

- si on s'intéresse au temps mis, certaines opérations prennent moins de temps que d'autres (par exemple une addition par rapport à une multiplication) mais est-ce vraiment important à l'échelle considérée ?
- si on s'intéresse à l'espace mémoire, doit-on considérer la taille précise en octets ou se contenter d'une estimation plus grossière ?

Mis à part dans certains cadres assez spécifiques, on se contente le plus souvent d'un ordre de grandeur pour ces complexités. Pour cela, on utilise des relations de comparaison de suites et une échelle de grandeur usuelle pour les comparer.

La notation grand O

Définition 34.23

Soit $(u_n)_{n\in\mathbb{N}}$ et $(v_n)_{n\in\mathbb{N}}$ deux suites de nombres réels, la suite $(v_n)_n$ étant à termes non nuls. On dit que la suite $(u_n)_n$ est dominée par $(v_n)_n$ lorsque la suite quotient $\left(\frac{u_n}{v_n}\right)_n$ est bornée.

On note alors $u_n = O(v_n)$.

Cette dernière notation se lit *u_n est un grand O de v_n*.

Remarque 34.24 C'est bien cette locution qu'il faut avoir en tête quand on pense aux grands O et il faut faire attention de ne pas considérer l'égalité en tant que telle sans s'assurer que ce que l'on fait est licite. Quand on écrira par la suite $O(v_n)$ on signifiera *n'importe quelle suite qui soit un $O(v_n)$*.

Si $u_n = O(v_n)$, cela signifie qu'il existe un facteur $M > 0$ tel que pour tout entier n, on ait $-M \mid v_n \mid \leq u_n \leq M \mid v_n \mid$. Les variations de la suite $(u_n)_n$ sont ainsi entièrement contrôlées par les variations de $(v_n)_n$.

En informatique, on ne considère pour la complexité que des suites positives, ce qui permet de simplifier la relation : si $(u_n)_n$ est une suite de réels positifs et $(v_n)_n$ une suite de réels strictement positifs, alors $u_n = O(v_n) \Leftrightarrow \exists M > 0, \forall n \in \mathbb{N}, u_n \leq Mv_n$. C'est le cadre dans lequel on se place implicitement dans la suite de ce document.

Remarque 34.25 Des suites comme $(n \log_2 n)_n$ s'annulent ou ne sont pas définies pour les petites valeurs de n (ici $n = 0$ et $n = 1$). On ne les considère alors qu'à partir d'un certain rang, ce qui ne change rien puisque, comme on va le voir, la notion est asymptotique.

On peut visualiser graphiquement cette relation :

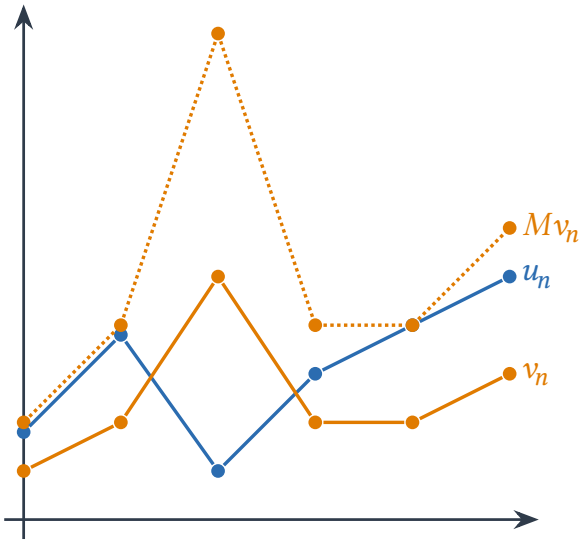


Figure 34.1 — $u_n = O(v_n)$: les points Mv_n sont tous au-dessus des points u_n

On a $u_n = O(v_n)$ si et seulement s'il est possible de multiplier les ordonnées de chaque point (n, v_n) par une même constante afin que ces nouveaux points soient tous au-dessus des points (n, u_n). On peut voir que la courbe déduite des v_n enveloppe, à un facteur près, celle des u_n.

On a relié ici les valeurs des suites pour mieux mettre en valeur cette notion d'enveloppe.

Cette relation est une notion **asymptotique** : elle n'a d'intérêt que lorsqu'on considère des rangs au voisinage de l'infini. En effet, pour un nombre fini de termes, il est toujours possible de trouver un tel M, mais pour un nombre infini, ce n'est pas le cas.

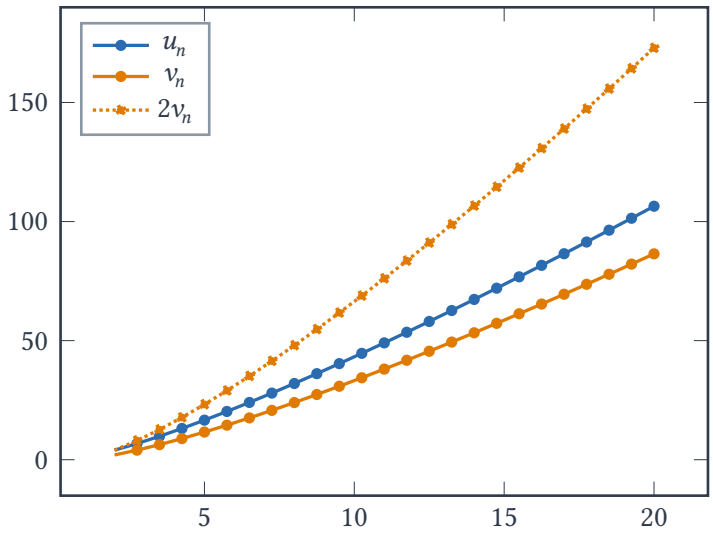


Figure 34.2 — $u_n = n + n\log_2 n$ reste sous $2v_n$ avec $v_n = n\log_2 n$

Ici, on compare asymptotiquement les suites $(u_n)_n$ et $(v_n)_n$ où pour $n \geq 2$, $u_n = n + n\log_2 n$ et $v_n = n\log_2 n$. Pour simplifier la visualisation, on a tracé les fonctions correspondantes. On remarque qu'on a bien $n + n\log_2 n = O(n\log_2 n)$: pour $n \geq 2$, $\log_2 n \geq 1$ donc $u_n \leq 2v_n$.

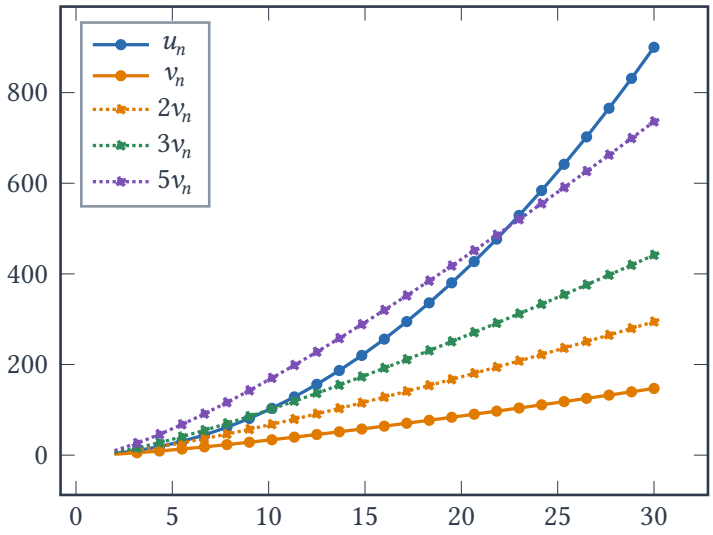


Figure 34.3 — $u_n = n^2$ finit par dépasser tout multiple de $v_n = n\log_2 n$

Par contre, si on compare les suites $(u_n)_n$ et $(v_n)_n$ où pour $n \geq 2$, $u_n = n^2$ et $v_n = n\log_2 n$, on remarque que quelle que soit la valeur choisie pour M, il y aura un rang à partir duquel $u_n > Mv_n$: en effet $\frac{u_n}{v_n} = \frac{n}{\log_2 n} \to +\infty$.

Ici, $n^2 \neq O(n\log_2 n)$.

Remarque 34.26 On a ici utilisé le logarithme en base 2, noté $\log_2$, qui est essentiel en informatique : si $x = \log_2(n)$ alors $n = 2^x$ où x est un réel. On considère aussi $p = \lceil \log_2(n) \rceil$ qui est le plus petit entier égal ou supérieur à $\log_2(n)$. On parle de **partie entière supérieure** et on a alors $2^{p-1} < n \leq 2^p$. Le nombre de chiffres nécessaires pour écrire $n \geq 1$ en binaire est $\lfloor \log_2(n) \rfloor + 1$, qui vaut p ou $p+1$. On a $\lceil \log_2(n) \rceil = O(\log_2(n))$ et ainsi, le plus souvent, on ne considère pas la partie entière explicitement. De la même manière, $\log_2(n) = \frac{\ln n}{\ln 2} = O(\ln n)$.

Un cas important de grand O est celui des $O(1)$. Si $u_n = O(1)$, cela signifie que $(u_n)_{n\in\mathbb{N}}$ est une suite bornée.

Échelle de comparaison

On rappelle les limites obtenues en mathématiques que l'on nomme **croissances comparées** :

$$\forall \alpha, \beta > 0, \lim_{n\to+\infty} \frac{(\ln n)^\alpha}{n^\beta} = 0$$

$$\forall \alpha \in \mathbb{R}, \forall \beta > 1, \lim_{n\to+\infty} \frac{n^\alpha}{\beta^n} = 0$$

Or, si $\frac{u_n}{v_n} \underset{n\to+\infty}{\to} 0$, *a fortiori* le quotient est borné et $u_n = O(v_n)$. Ainsi, on a les relations suivantes :

$$\forall \alpha, \beta > 0, (\log_2 n)^\alpha = O(n^\beta)$$

$$\forall \alpha \in \mathbb{R}, \forall \beta > 1, n^\alpha = O(\beta^n)$$

De plus, si $\alpha \geq \beta > 0$, $n^\beta = O(n^\alpha)$, $(\log_2 n)^\beta = O((\log_2 n)^\alpha)$ et $\beta^n = O(\alpha^n)$.

On se ramène souvent à des complexités qui sont des grands O de produits de ces suites.

Ordre de grandeur et relation Θ

On vient de voir que $\log_2 n = O(n)$, mais on a également $\log_2 n = O(n^2)$. Quand on cherche à caractériser la complexité par un grand O, on va souvent chercher le grand O le plus proche de la suite.

Il est possible de définir cela précisément en considérant des suites qui sont chacune des grands O l'une de l'autre.

Par exemple, on a vu que $n \log_2 n + n = O(n \log_2 n)$, mais on a également $n \log_2 n = O(n + n \log_2 n)$.

Quand $u_n = O(v_n)$ et $v_n = O(u_n)$, on note $u_n = \Theta(v_n)$, qui est une relation symétrique qui correspond à la notion *avoir le même ordre de grandeur*. Très souvent, lorsque l'on parle de complexité, on utilise des grands O quand, en fait, on exprime des Θ. Par exemple, l'accès à un élément dans un tableau est en $O(1)$ et il ne serait pas précis de dire que c'est en $O(n)$ **même si c'est parfaitement correct**.

On peut visualiser cette relation Θ en considérant qu'il existe ainsi $M, M' > 0$ tels que $u_n \leq M v_n$ et $v_n \leq M' u_n$. Mais on a alors

$$\frac{1}{M'} v_n \leq u_n \leq M v_n$$

Ainsi, $u_n = \Theta(v_n)$ signifie qu'on peut encadrer $(u_n)_n$ entre deux multiples de $(v_n)_n$.

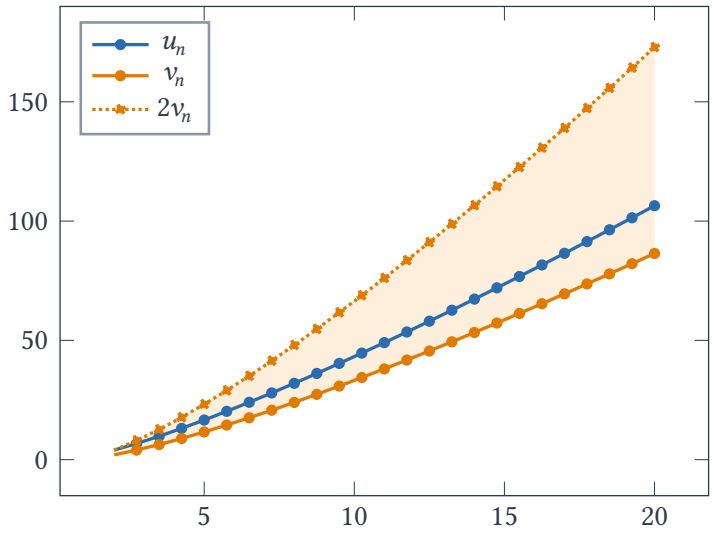


Figure 34.4 — $u_n = \Theta(v_n)$: u_n est encadrée entre v_n et $2v_n$

En reprenant la figure précédente, on observe visuellement, pour $n \geq 2$,

$$n \log_2 n \leq n \log_2 n + n \leq 2n \log_2 n$$

Avoir $u_n = \Theta(v_n)$ signifie donc que u_n évolue entre deux guides suivant les variations de v_n.

Définition 34.27

Pour des suites strictement positives, on note $u_n = \Omega(v_n)$ lorsque $v_n = O(u_n)$, c'est-à-dire lorsqu'il existe $m > 0$ tel que pour tout $n \in \mathbb{N}$, $u_n \geq m v_n$.

Ainsi, $u_n = \Theta(v_n)$ si et seulement si $u_n = O(v_n)$ et $u_n = \Omega(v_n)$. Un grand O exprime une majoration de la complexité, un Ω une minoration : par exemple, la recherche d'un élément dans un tableau de taille n par la fonction **A** ci-dessus a une complexité dans le pire des cas en $\Omega(n)$, car il faut parcourir tout le tableau quand l'élément est absent.

Opérations sur les grands O

Si $u_n = O(w_n)$ et $v_n = O(w_n)$ alors $u_n + v_n = O(w_n)$. Ainsi, des grands O de même ordre s'ajoutent.

Remarque 34.28 Comme on l'a vu précédemment, un grand O n'est pas très précis, et il est possible que par ajout on puisse obtenir un meilleur grand O. Par exemple : $n = O(n)$ et $\log_2 n - n = O(n)$ mais $n + \log_2 n - n = \log_2 n$, qui est un $O(\log_2 n)$. Comme on ne considère ici que des suites positives, ce phénomène de compensation n'aura pas lieu.

Si $u_n = O(v_n)$ et w_n est une autre suite de réels strictement positifs, alors $u_n w_n = O(v_n w_n)$. On en déduit ainsi un principe qui nous sera utile par la suite : $nO(1) = O(n)$.

34.4.3 Complexités en temps classiques

On parle ici de complexité par raccourci pour parler de complexité dans le pire des cas en temps.

Complexité constante

On dit qu'un algorithme a une complexité constante quand $C_n^t = O(1)$. Il existe ainsi une constante M telle que le temps pris par l'algorithme **sur une entrée quelconque** soit inférieur à M.

De nombreuses opérations sont en temps constant sur les structures de données usuelles. Parmi celles-ci, citons-en deux essentielles :

- accéder à une case d'indice quelconque dans un tableau ;
- accéder à la tête ou à la queue (la liste privée de son premier élément) d'une liste chaînée.

Les algorithmes ou opérations en temps constant jouent un rôle primordial dans l'analyse de la complexité d'algorithmes, comme on le verra dans la partie suivante, car elles permettent de se concentrer sur les répétitions de ces opérations pour déterminer la complexité : une boucle qui se répète n fois et n'effectue que des opérations en temps constant dans son corps sera de complexité $nO(1) = O(n)$.

Complexité linéaire

On dit qu'un algorithme a une complexité linéaire quand $C_n^t = O(n)$.

Cette complexité correspond à un traitement de temps constant sur chaque élément d'une entrée de taille n. C'est le cas de la recherche d'un élément dans un tableau ou de la recherche de son maximum.

Pour la recherche linéaire d'un élément, correspondant par exemple au programme ci-dessous, le pire cas correspond à ne pas avoir `x` dans `tableau`, ce qui oblige à effectuer toutes les itérations. On a bien une complexité temporelle en pire cas de $O(n)$.

OCaml

```
let recherche t x =
    let n = Array.length t in
    let i = ref 0 in
    while !i < n && t.(!i) <> x do
        incr i
    done;
    if !i < n then Some !i else None
```

C

```
int recherche(int *tableau, size_t taille, int x)
{
    /* renvoie le plus petit indice i tel que
       tableau[i] = x ou -1 si x n'est pas dans
       le tableau */
    for (size_t i = 0; i < taille; i++)
    {
        if (tableau[i] == x)
            return i;
    }

    return -1;
}
```

Python

```
def recherche(t, x):
    for i in range(len(t)):
        if t[i] == x:
            return i
    return None
```

Complexité quadratique, polynomiale

On dit qu'un algorithme a une complexité quadratique quand $C_n^t = O(n^2)$. Par extension, on dit qu'il a une complexité polynomiale quand il existe $k \in \mathbb{N}$ tel que $C_n^t = O(n^k)$. On parle parfois aussi de complexité polynomiale pour des complexités plus précises en $O(n^\alpha)$ où α est un réel strictement positif.

L'exemple classique d'un algorithme quadratique est celui dû à un double parcours d'un tableau. On reprend ici l'algorithme de tri par sélection vu dans la partie Exemple du tri par sélection.

Afin d'analyser sa complexité, on procède fonction par fonction pour un tableau de taille n :

- `echange` est en temps constant : $O(1)$.
- `indice_minimum` réalise un parcours d'une partie du tableau et effectue des opérations en temps constant à chaque étape. La complexité est donc linéaire : $O(n)$.
- `tri_par_selection` réalise également un parcours du tableau mais à chaque étape, on appelle `indice_minimum` qui est en $O(n)$, la complexité est donc en $nO(n) = O(n^2)$: elle est quadratique.

Plus précisément, l'appel `indice_minimum t i (n-1)` effectue $n-1-i$ itérations, donc le tri effectue $\sum_{i=0}^{n-1}(n-1-i) = \frac{n(n-1)}{2}$ comparaisons entre éléments du tableau, quelle que soit l'entrée : la complexité est en $\Theta(n^2)$.

Complexité logarithmique

On dit qu'un algorithme a une complexité logarithmique quand $C_n^t = O(\log_2 n)$.

Pour illustrer cette complexité, on reprend l'algorithme de recherche dichotomique vu dans la partie Exemple de la recherche dichotomique.

Chaque opération effectuée dans le corps de la boucle étant en temps constant, la complexité de cet algorithme est proportionnelle au nombre d'itérations, soit, pour la version récursive, au nombre d'appels récursifs.

Si on considère un sous-tableau de $n = j - i + 1 \geq 1$ éléments au début d'une itération, l'itération suivante éventuelle porte sur un sous-tableau de $p - i = \left\lfloor \frac{n-1}{2} \right\rfloor$ ou de $j - p = \left\lceil \frac{n-1}{2} \right\rceil = \left\lfloor \frac{n}{2} \right\rfloor$ éléments, donc d'au plus $\lfloor n/2 \rfloor$ éléments.

Montrons par récurrence sur $k \in \mathbb{N}$ que si $n < 2^k$, l'algorithme effectue au plus k itérations sur un sous-tableau de n éléments. Pour $k = 0$, on a $n = 0$ et aucune itération n'a lieu. Si $n < 2^{k+1}$, après une première itération, il reste un sous-tableau d'au plus $\lfloor n/2 \rfloor < 2^k$ éléments, donc au plus k itérations par hypothèse de récurrence, soit au plus $k + 1$ en tout.

Pour $n \geq 1$, en prenant $k = \lfloor \log_2 n \rfloor + 1$, qui vérifie bien $n < 2^k$, on obtient au plus $\lfloor \log_2 n \rfloor + 1$ itérations. Donc le nombre d'itérations est en $O(\log_2 n)$ et c'est ainsi la complexité de l'algorithme.

Complexité quasi-linéaire

On dit qu'un algorithme a une complexité quasi-linéaire quand $C_n^t = O(n \log_2 n)$. C'est le cas de la plupart des algorithmes efficaces de tri de n éléments. On peut même montrer qu'il s'agit de la complexité optimale pour les tris qui ne procèdent que par comparaisons entre éléments.

Comme de nombreux algorithmes commencent par effectuer un tri avant d'effectuer un traitement linéaire, on retrouve des algorithmes quasi-linéaires par simple utilisation de ce tri.

Complexité exponentielle

On dit qu'un algorithme a une complexité exponentielle quand $C_n^t = O(a^n)$ pour un réel $a > 1$ (et qu'il n'est pas de complexité polynomiale).

Un exemple fondamental d'un tel algorithme est celui de l'énumération de données, par exemple pour chercher une solution par force brute. En effet, il y a 2^n entrées codées sur n bits et un algorithme cherchant une solution ainsi parmi ces entrées aura une complexité en $O(2^n)$ si le traitement de chaque entrée est en temps constant.

Estimation de l'impact des complexités sur le temps

Afin de mesurer l'impact d'une complexité, on va considérer un algorithme qui s'exécute en 1 seconde sur une entrée de taille n, et on va calculer combien de temps prendrait ce même algorithme sur une entrée de taille $10n$ ou $100n$.

Pour simplifier, on considère à chaque fois que C_n^t correspond exactement à l'ordre du grand O.

Complexité	Temps pour $10n$	Temps pour $100n$
1	1 s	1 s
$\log_2 n$	1,003 s	1,007 s
n	10 s	1 min 40 s
$n \log_2 n$	14,7 s	3 min 13 s
n^2	1 min 40 s	2 h 46 min 40 s
2^n	$3 \cdot 10^{19}$ années	$3 \cdot 10^{289}$ années

Remarque 34.29 Pour déterminer ces valeurs, on a pris la milliseconde comme unité, de sorte que $C_n^t = 1000$, afin d'en déduire une valeur de n.

Ainsi, si $\log_2 n = 1000$ on a $n = 2^{1000}$ et $\log_2(10n) = 1000 + \log_2 10 \approx 1003,3$. Bien sûr, ici, ce nombre 2^{1000} n'est pas réaliste. Dans un contexte de mémoire finie, une complexité logarithmique est identifiable à une complexité constante. Cela justifie la terminologie quasi-linéaire.

Si $n \log_2 n = 1000$ alors $n \approx 140,2$. Or, $1402 \log_2 1402 \approx 14\,700$ ms.

Si $2^n = 1000$, alors $n \approx 10$. Or 2^{100} ms $\approx 10^{30}$ ms, soit environ $3 \cdot 10^{19}$ années.

34.4.4 Calculer des complexités

Deux principes fondamentaux pour calculer des complexités :

- Si on effectue deux passes successives chacune en $O(u_n)$ alors la complexité globale est en $O(u_n)$. Il ne s'agit que de reformuler l'addition des grands O. Quand on a deux passes de complexités différentes, il suffit d'utiliser la plus grande complexité. Par exemple, un algorithme qui commence par un tri en $O(n \log_2 n)$ et qui effectue ensuite un traitement en $O(n)$ sera de complexité globale $O(n \log_2 n)$ car le traitement est également en $O(n \log_2 n)$.
- Si on effectue u_n itérations et que chaque itération est en $O(v_n)$ alors l'algorithme a une complexité de $O(u_n v_n)$. Cela permet de compter le nombre de boucles imbriquées et de se contenter de regarder ce qui se passe dans le corps des boucles.

34.4.5 Complexité à plusieurs paramètres

Jusqu'ici on a considéré des entrées dépendant d'un unique paramètre n, mais il est possible d'avoir des données dépendant de plusieurs paramètres.

On adapte directement la notation des grands O : si $(u_{n,p})$ et $(v_{n,p})$ sont deux suites de réels dépendant de deux paramètres, la seconde étant à termes non nuls, on note toujours $u_{n,p} = O(v_{n,p})$ quand le quotient est borné.

Données multidimensionnelles

Le cas le plus usuel de complexité dépendant de plusieurs paramètres est celui des données multidimensionnelles comme une image.

Si on considère une opération effectuant un traitement en temps constant sur chaque pixel d'une image de $w \times h$ pixels, cette opération aura une complexité en $O(wh)$. On ne peut plus parler de complexité linéaire ou quadratique ici car cela dépend d'une éventuelle relation entre w et h : si on ne travaille que sur des images de taille $1 \times h$ alors la complexité est $O(h)$, mais si on ne travaille que sur des images carrées, donc pour lesquelles $w = h$, la complexité est $O(h^2)$.

Plus généralement, si on considère des données organisées dans des tableaux imbriqués, on effectuera un traitement sur chaque donnée à l'aide de boucles imbriquées non conditionnelles. La complexité sera alors celle du corps de boucles multipliée par le produit du nombre d'itérations de chaque boucle.

Compromis entre paramètres

Dans certains cas, en particulier pour les graphes, on peut effectuer des traitements successifs dont la complexité ne s'exprime pas en fonction du même paramètre. Imaginons par exemple un programme ayant la structure suivante :

OCaml

```
for i = 0 to n-1 do
    (* corps de boucle en O(1) *)
done;

for j = 0 to p-1 do
    (* corps de boucle en O(1) *)
done
```

C

```c
for (int i = 0; i < n; i++)
{
    /* corps de boucle en O(1) */
}

for (int j = 0; j < p; j++)
{
    /* corps de boucle en O(1) */
}
```

Python

```python
for i in range(n):
    # corps de boucle en O(1)

for j in range(p):
    # corps de boucle en O(1)
```

La complexité de la première boucle est en $O(n)$ et celle de la deuxième en $O(p)$. La complexité globale est en $O(n+p)$ car $n \leq n+p$ et $p \leq n+p$.

34.4.6 Complexité en moyenne

On reprend ici les notations de la partie Complexité dans le pire des cas.

Définition 34.30

Lorsque pour tout $n \in \mathbb{N}$, I_n est fini et non vide, on appelle :

- **complexité temporelle en moyenne** la suite $(C_n^{t,m})_n$ définie par $C_n^{t,m} = \frac{1}{|I_n|}\sum_{e\in I_n} t(e)$;
- **complexité spatiale en moyenne** la suite $(C_n^{s,m})_n$ définie par $C_n^{s,m} = \frac{1}{|I_n|}\sum_{e\in I_n} s(e)$.

On peut étendre cette définition à un cadre infini en considérant une distribution de probabilité sur I_n et T_n la variable aléatoire associée à t sur I_n. Si T_n est d'espérance finie, on pourra parler de complexité en moyenne pour la suite des $E(T_n)$. Concrètement, pour I_n dénombrable, on considère alors une fonction $p_n : I_n \to [0,1]$ telle que $\sum_{e\in I_n} p_n(e) = 1$ et, lorsque la somme converge, on note ainsi

$$C_n^{t,m} = \sum_{e\in I_n} p_n(e)t(e)$$

$$C_n^{s,m} = \sum_{e\in I_n} p_n(e)s(e)$$

Un exemple usuel de calcul de complexité en moyenne est celui des tris. En effet, même si les entrées de taille n sont en nombre infini, on peut considérer qu'un tableau de valeurs deux à deux distinctes est l'image par une permutation du tableau trié. Si le tableau est de taille n, on aura ainsi $n!$ permutations ce qui permet, du moment que le comportement de l'algorithme de tri considéré ne dépend que de cette permutation, de calculer la complexité en moyenne sur l'ensemble des permutations, supposées équiprobables.

Remarque 34.31 Les permutations d'un ensemble sont les applications bijectives de cet ensemble dans lui-même. Si l'ensemble contient n éléments, il y a $n!$ permutations.

Par exemple, les six permutations sur l'ensemble $\{1,2,3\}$ correspondent aux diagrammes sagittaux suivants :

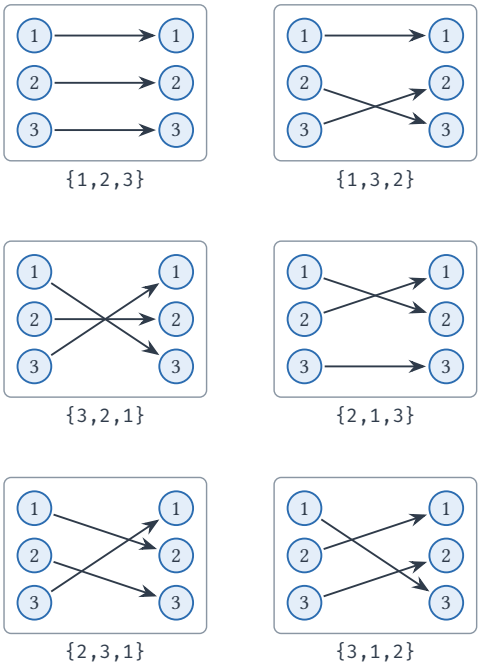


Figure 34.5 — Les six permutations de $\{1, 2, 3\}$ et les tableaux associés

```
Ces six permutations correspondent elles-mêmes, de gauche à droite et de haut
en bas, aux tableaux `{1,2,3}`, `{1,3,2}`, `{3,2,1}`, `{2,1,3}`, `{2,3,1}` et
`{3,1,2}`.
```

Exemple de calcul de complexité temporelle en moyenne

On considère la recherche linéaire vue dans la partie Complexité linéaire. L'ensemble des entrées est ici infini, on va donc supposer pour faire le calcul qu'on ne considère que des tableaux de valeurs deux à deux distinctes et qu'on recherche un élément présent dans le tableau, chaque élément étant équiprobable.

Si on recherche le i-ème élément du tableau, l'algorithme effectue i itérations avant d'y accéder et de renvoyer son indice. Ainsi, le temps pour cette entrée est de iC où C est le coût d'une itération (en négligeant les opérations effectuées hors de la boucle).

La complexité temporelle en moyenne est alors $C_n^{t,m} = \sum_{i=1}^{n} \frac{1}{n} iC = \frac{(n+1)C}{2} = \Theta(n)$. On retrouve ici le même ordre de grandeur que la complexité dans le pire des cas. La sortie prématurée de la boucle divise en moyenne le temps par deux environ, mais n'a donc aucune influence sur l'ordre de grandeur de la complexité.

34.4.7 Complexité amortie

Dans le cadre de l'étude des structures de données, il est fréquent de considérer non pas la complexité dans le pire des cas d'une opération mais celle d'une succession d'opérations divisée par le nombre d'opérations effectuées. Ainsi, on peut très bien avoir une opération ponctuellement plus coûteuse que les autres, mais en procédant ainsi on lisse le surcoût sur l'ensemble des opérations. On parle alors de **complexité amortie**.

L'étude de la complexité amortie est traitée dans le chapitre Complexité amortie.

34.4.8 Pertinence de la complexité spatiale

Même si la complexité temporelle est le plus souvent celle qui est importante à calculer, certains algorithmes ont une complexité temporelle faible mais en contrepartie une complexité spatiale élevée. On parle alors de compromis temps-mémoire.

Un exemple classique d'un tel compromis est celui de la programmation dynamique où on passe d'une complexité temporelle exponentielle à une complexité temporelle polynomiale en stockant des valeurs intermédiaires pour ne pas les recalculer. En procédant ainsi, on utilise de la mémoire supplémentaire pour stocker ces valeurs, souvent dans un tableau de taille polynomiale.

Cela est illustré dans le programme suivant qui permet de déterminer le n-ième terme de la suite de Fibonacci, ce qui n'a pas d'intérêt informatique mais est caractéristique des récurrences que l'on résoudra par la programmation dynamique. Le nombre d'appels de la version exponentielle est calculé dans la partie Complexité en nombre d'appels du chapitre sur la récursivité.

OCaml

```
(* Fibonacci exponentiel *)
let rec fibo n =
    if n = 0
    then 0
    else if n = 1
    then 1
    else fibo (n-1) + fibo (n-2)

(* Fibonacci linéaire *)
let fibo n =
    (* n+2 cases pour que prec.(1) existe même si n = 0 *)
    let prec = Array.make (n+2) 0 in
    prec.(1) <- 1;
    for i = 2 to n do
        prec.(i) <- prec.(i-1) + prec.(i-2)
    done;
    prec.(n)
```

C

```
/* Fibonacci exponentiel */
int fibo_exp(int n)
{
    if (n == 0)
        return 0;
    else if (n == 1)
        return 1;
    else
        return fibo_exp(n-1) + fibo_exp(n-2);
}

/* Fibonacci linéaire */
int fibo(int n)
{
    /* n+2 cases pour que prec[1] existe même si n = 0 */
    int *prec = malloc((n+2) * sizeof(int));
    prec[0] = 0;
    prec[1] = 1;
    for (int i = 2; i <= n; i++)
        prec[i] = prec[i-1] + prec[i-2];
    int res = prec[n];
    free(prec);
    return res;
}
```

Python

```
### Fibonacci exponentiel
def fibo(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibo(n-1) + fibo(n-2)

### Fibonacci linéaire
def fibo(n):
    # n+2 cases pour que prec[1] existe même si n = 0
    prec = [0] * (n+2)
    prec[1] = 1
```

```
    for i in range(2, n+1):
        prec[i] = prec[i-1] + prec[i-2]
    return prec[n]
```

Remarque 34.32 Sur cet exemple, la version exponentielle n'est pas pour autant économe en mémoire : la profondeur maximale des appels récursifs est n, donc sa complexité spatiale est déjà en $\Theta(n)$ à cause de la pile d'appels (voir le chapitre Récursivité). La version linéaire utilise un tableau de taille $n+2$, donc aussi un espace en $\Theta(n)$; on pourrait même se contenter de conserver les deux derniers termes calculés pour obtenir une complexité spatiale constante. Le compromis temps-mémoire apparaît vraiment quand le tableau des valeurs intermédiaires est plus grand que la profondeur de récursion, ce qui sera fréquent en programmation dynamique.

34.5 Exercices

Exercice

On considère le programme suivant :

OCaml

```
let multiplication x y c =
    let m = ref 0 in
    let vy = ref y in
    let vx = ref x in
    while !vy <> 0 do
        m := !m + !vx * (!vy mod c);
        vx := !vx * c;
        vy := !vy / c
    done;
    !m
```

C

```
int multiplication(int x, int y, int c)
{
    int m = 0;
    while (y != 0)
    {
        m = m + x * (y % c);
        x = x * c;
        y = y / c;
    }
    return m;
}
```

Python

```
def multiplication(x, y, c):
    m = 0
    while y != 0:
        m = m + x * (y % c)
        x = x * c
        y = y // c
    return m
```

On suppose que $c \geq 2$ et que $x, y \in \mathbb{N}$.

1. Montrer que cet algorithme termine.
2. Montrer qu'il renvoie le produit xy.
3. Réécrire ce programme en récursif. Peut-on déduire des preuves précédentes que l'algorithme termine et qu'il est correct ?

Corrigé

1. La quantité y est un entier naturel, et tant que la condition de boucle est vérifiée, $y > 0$; comme $c \geq 2$, on a alors $\lfloor y/c \rfloor \leq y/2 < y$. Ainsi y est un variant de boucle.

2. On va commencer par noter x_0 et y_0 les valeurs initiales respectives de x et de y. On va également tenir compte du nombre d'itérations, ce qui revient à rajouter un compteur i au programme. On va également noter $y_0 = \sum_{j=0}^{d} a_j c^j$ l'écriture de y_0 en base c, avec $a_j \in [\![0, c-1]\!]$ et $a_d \neq 0$ si $y_0 \neq 0$.

   - $y = \left\lfloor \frac{y_0}{c^i} \right\rfloor = \sum_{j=i}^{d} a_j c^{j-i}$ et $x = x_0 c^i$ sont des invariants directement validés. En particulier, au début de l'itération i, $y \bmod c = a_i$, et la boucle s'arrête quand $y = 0$, c'est-à-dire après $d+1$ itérations (aucune si $y_0 = 0$).
   - On va maintenant prouver l'invariant $I(i, m) := m = x_0 \sum_{j=0}^{i-1} a_j c^j$.
     - **Initialisation** : avant la première itération, on a $I(0, 0)$ qui est vérifié car la somme est vide.
     - **Hérédité** : si l'invariant est vérifié au début de l'itération i, on a $m + x(y \bmod c) = x_0 \sum_{j=0}^{i-1} a_j c^j + x_0 c^i a_i = x_0 \sum_{j=0}^{i} a_j c^j$, donc $I(i+1, m + x(y \bmod c))$ est vérifié en fin d'itération.

   Ainsi, en sortie de boucle, on a $m = x_0 \sum_{j=0}^{d} a_j c^j = x_0 y_0$ (et $m = 0 = x_0 y_0$ si $y_0 = 0$).

   ::: {.claude kind=« redige » raison=""}

   Remarque : on peut aussi utiliser l'invariant plus simple $m + xy = x_0 y_0$. En effet, en notant $y = cq + r$ la division euclidienne, $m + xr + (xc)q = m + x(cq + r) = m + xy$. En sortie de boucle $y = 0$, donc $m = x_0 y_0$.

   :::

3. Une version récursive :

   ::: {.onglet lang=« ocaml »}

   ```
   let rec multiplication x y c =
       if y = 0
       then 0
       else x * (y mod c) + multiplication (x*c) (y/c) c
   ```

   :::

   ::: {.onglet lang=« c »}

   ```
   int multiplication(int x, int y, int c)
   {
       if (y == 0)
           return 0;
       return x * (y % c) + multiplication(x * c, y / c, c);
   }
   ```

   :::

   ::: {.onglet lang=« python »}

   ```
   def multiplication(x, y, c):
       if y == 0:
           return 0
       return x * (y % c) + multiplication(x * c, y // c, c)
   ```

   :::

   ::: {.claude kind=« modifie » raison=« remarque de l'auteur développée en preuve par récurrence forte »}

   Les idées des preuves précédentes s'adaptent directement, mais il faut les présenter différemment, sans invariant de boucle. La terminaison vient de ce que le paramètre y est un entier naturel qui diminue

strictement à chaque appel récursif. Pour la correction, on montre par récurrence forte sur $y \in \mathbb{N}$ que `multiplication x y c` renvoie xy pour tout x : c'est vrai pour $y = 0$ et, si $y > 0$, en écrivant $y = cq + r$ avec $q = \lfloor y/c \rfloor < y$, l'hypothèse de récurrence donne le résultat $xr + (xc)q = x(cq + r) = xy$. :::

Exercice

On considère un polynôme $P = \sum_{i=0}^{n} c_i X^i$ représenté comme un tableau de $n + 1$ coefficients tel que pour tout $i \in [\![0, n]\!]$, $P[i] = c_i$.

On veut calculer $P(a)$ pour une valeur a. Pour simplifier, on va considérer ici que les valeurs sont toutes entières.

OCaml

```
let horner p a =
    let v = ref 0 in
    for i = Array.length p - 1 downto 0 do
        v := a * !v + p.(i)
    done;
    !v
```

C

```
int horner(const int *P, int n, int a)
{
    int v = 0;
    for (int i = n; i >= 0; i--)
    {
        v = a * v + P[i];
    }
    return v;
}
```

Python

```
def horner(P, a):
    v = 0
    # rappel : reversed(P) parcourt P
    # en sens inverse
    for c in reversed(P):
        v = a * v + c
    return v
```

1. Montrer que ce programme est correct.
2. Combien effectue-t-il de multiplications et d'additions ?
3. Comparer avec l'algorithme obtenu en ajoutant chaque $c_i a^i$ pour i croissant et en maintenant une variable pour a^i.

Corrigé

1. On considère l'invariant $I(v, i) \coloneqq v = c_{i+1} + c_{i+2}a + \ldots + c_n a^{n-i-1}$, où i est l'indice de l'itération qui commence.

   - **Initialisation** : avant la première itération on a $I(0, n)$ vérifié car $v = 0$ et la somme est vide.
   - **Hérédité** : si l'invariant est vérifié en début d'itération, on passe à $I(av + c_i, i - 1)$ en fin d'itération. Or, si $v = c_{i+1} + \ldots + c_n a^{n-i-1}$, on a bien $av + c_i = c_i + c_{i+1}a + \ldots + c_n a^{n-i}$ et donc l'invariant est vérifié en fin d'itération.

   En sortie de boucle, on a alors $I(v, -1)$ donc $v = c_0 + c_1 a + \ldots + c_n a^n = P(a)$.

2. On effectue $n+1$ itérations et à chaque itération une multiplication et une addition, donc $n+1$ multiplications et $n+1$ additions (la première itération calcule $a \times 0 + c_n$: on pourrait économiser une multiplication et une addition en initialisant v à c_n).
3. Pour cet algorithme, on devrait faire une multiplication de plus à chaque itération pour maintenir a^i : il effectue de l'ordre de $2n$ multiplications au lieu de n, pour le même nombre d'additions.

Exercice

On considère le problème du drapeau hollandais : étant donné un tableau t et un indice i, on note `p=t[i]`, on cherche à permuter les éléments de `t` de sorte qu'il y ait trois zones dans le tableau : les éléments $< p$, les éléments $= p$ puis les éléments $> p$.

Ainsi, si on considère `t = [5, 2, 3, 5, 1, 4]` et `p=t[2]=3` on pourra obtenir `[2, 1, 3, 5, 5, 4]` à l'issue de cet algorithme.

1. Écrire un programme résolvant ce problème en temps linéaire.
2. Prouver sa correction.

Corrigé

Le programme :

```
void echange(int *t, int i, int j)
{
    int temp = t[i];
    t[i] = t[j];
    t[j] = temp;
}

void drapeau(int *t, int nb, int i)
{
    int v = t[i];
    int l = 0;
    int c = nb-1;
    int r = nb-1;

    while (l <= c)
    {
        int w = t[l];
        if (w < v)
        {
            l = l+1;
        }
        else if (w == v)
        {
            echange(t, l, c);
            c = c-1;
        }
        else
        {
            echange(t, l, c);
            echange(t, c, r);
            c = c-1;
            r = r-1;
        }
    }
}
```

Sur l'exemple de l'énoncé, ce programme produit `[1, 2, 3, 5, 4, 5]`, qui convient aussi : la solution n'est pas unique.

L'idée de cet algorithme est d'avoir trois indices l, c et r (pour *left*, *center* et *right*) qui délimitent quatre zones :

- celle des indices 0 à $l-1$ qui contient des valeurs $< v$;
- celle des indices l à c qui contient les valeurs pas encore traitées ;
- celle des indices $c+1$ à r qui contient des valeurs égales à v ;
- celle des indices $r+1$ à $nb-1$ qui contient des valeurs $> v$.

Au départ, les trois zones de valeurs traitées sont vides et au fur et à mesure de l'algorithme, elles augmentent jusqu'à couvrir l'ensemble du tableau. Comme, à chaque itération, soit l croît de 1, soit c décroît de 1, la quantité $c-l$, positive tant qu'on entre dans la boucle, est un variant de boucle qui diminue d'exactement 1 à chaque itération : on effectue exactement nb itérations, chacune en temps constant, et la complexité est linéaire.

On va montrer que le découpage des zones est effectivement un invariant (la zone des valeurs non traitées n'étant soumise à aucune condition) :

- **Initialisation** : au départ, comme on l'a vu, les trois zones de valeurs traitées sont vides donc l'invariant est trivialement vérifié.
- **Hérédité** : si l'invariant est vrai au début d'une itération, on va faire trois cas selon la position de `w = t[l]` par rapport à `v` :
  - Si `w < v` alors, après incrémentation de l, la zone des indices de 0 à $l-1$ contient toujours des valeurs $< v$ et les autres zones n'ont pas bougé, l'invariant est encore vérifié.
  - Si `w = v` alors on place cette valeur `w` à l'indice c, au début de la zone centrale une fois c décrémenté, et la valeur non traitée qui était à l'indice c passe à l'indice l. Les autres zones ne sont pas touchées, donc l'invariant est encore vérifié.
  - Si `w > v`, le premier échange place `w` à l'indice c et la valeur non traitée d'indice c à l'indice l. Le second échange distingue deux cas :
    - si $c < r$, la zone centrale est non vide et `t[r]` vaut v : l'échange place v à l'indice c et `w` à l'indice r. Après décrémentation de c et r, la zone centrale est formée des indices $c+1$ (ancien c, qui contient v) à r (ancien $r-1$), qui contiennent tous v, et la zone de droite commence à l'ancien indice r, qui contient `w` $> v$;
    - si $c = r$, la zone centrale est vide, le second échange ne fait rien, et après décrémentation la zone de droite commence à l'ancien indice $c = r$ qui contient `w` $> v$, la zone centrale restant vide.

    Dans les deux cas, la première zone n'a pas bougé et l'invariant est vérifié.

On a donc validé l'invariant. En sortie de boucle, on a $l = c+1$: la zone des valeurs non traitées est vide, donc les trois autres zones couvrent l'intégralité du tableau, ce qui est le résultat voulu. Enfin, le tableau n'ayant été modifié que par des échanges, il contient les mêmes éléments qu'au départ.

CHAPITRE 35

Complexité amortie

DANS CE CHAPITRE

35.1 Introduction
35.2 Techniques de calcul

35.1 Introduction

Dans le cadre de l'étude des structures de données, il est fréquent de considérer non pas la complexité dans le pire des cas d'une opération mais celle d'une succession d'opérations divisée par le nombre d'opérations effectuées. Ainsi, on peut très bien avoir une opération ponctuellement plus coûteuse que les autres, mais en procédant ainsi on lisse le surcoût sur l'ensemble des opérations. On parle alors de **complexité amortie**.

> **Remarque 35.1** Cette notion ne masque pas le fait qu'une opération puisse prendre ponctuellement plus de temps. Dans des contextes temps réel où il est important de maîtriser pleinement les complexités, il est peu judicieux d'utiliser de telles complexités. Par exemple, dans une visualisation en 3D, pour maintenir un débit constant d'images par seconde, chaque image doit prendre un temps similaire. Se reposer sur une structure de données ayant une faible complexité amortie mais une complexité dans le pire des cas importante, c'est risquer d'avoir des saccades avec une image qui prendrait plus de temps pour être calculée.

> **Remarque 35.2** La complexité amortie n'est pas une complexité en moyenne : il n'y a ici aucune hypothèse probabiliste sur les entrées. On majore le coût total de **n'importe quelle** suite d'opérations, dans le pire des cas.

On va reprendre ici deux exemples simples déjà traités par ailleurs.

35.1.1 Implémentation d'une file avec deux piles

On considère une file réalisée avec deux piles `in` et `out` (voir le chapitre Piles et files).

Comme on l'a vu, on enfile en $O(1)$ en empilant sur la pile `in` et on défile :

- soit en $O(1)$ en dépilant sur `out` si elle est non vide ;
- soit en $O(p)$, où p est le nombre d'éléments de `in`, en déversant `in` dans `out` puis en se ramenant au cas précédent.

Ainsi, en pire cas, on est en $O(n)$ pour l'opération *défiler*, où n est le nombre d'éléments de la file, mais la bascule permettra de faire ensuite cette même opération en $O(1)$.

35.1.2 Les tableaux dynamiques

Les tableaux dynamiques sont une structure de données de haut niveau pour implémenter le type abstrait des listes. La différence principale entre cette structure de données et celle des tableaux de C est qu'on peut ajouter et supprimer des éléments.

Un tableau dynamique d'entiers est un triplet `(t,c,l)` où `t` est un tableau de taille `c`, appelée la capacité du tableau dynamique, et `l` est un autre entier représentant la longueur logique du tableau. À tout moment $c \geq l$. Dans `t` il y a ainsi $c - l$ cases déjà allouées qui permettent de rajouter un élément en temps constant. Quand $c = l$, on alloue une nouvelle zone mémoire, souvent de taille $2c$, on recopie le tableau `t` dans cette zone et on a donc, pour les ajouts suivants, un certain nombre de cases d'avance.

La figure suivante présente une succession d'ajouts :

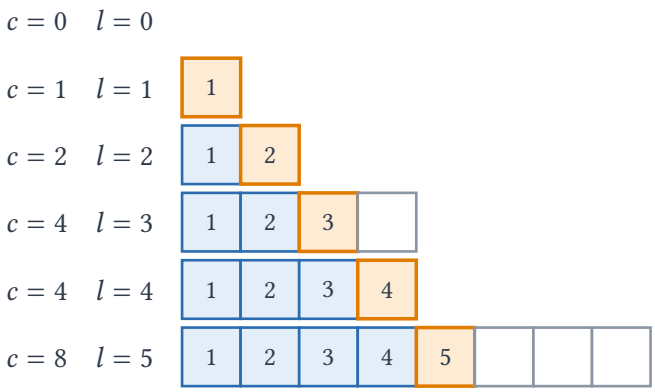


Figure 35.1 — Ajouts successifs dans un tableau dynamique (élément ajouté en orange)

Une implémentation de ces opérations est proposée dans le programme suivant :

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

typedef struct {
    int *t;
    size_t c;
    size_t l;
} tableau_dynamique;

tableau_dynamique tableau_dynamique_creer(void)
{
    tableau_dynamique d;

    d.t = NULL;
    d.c = 0;
    d.l = 0;

    return d;
}

void tableau_dynamique_ajout(tableau_dynamique *d, int x)
{
    if (d->c == d->l)
    {
        if (d->c == 0)
        {
            d->c = 1;
            d->t = malloc(sizeof(int));
        }
        else
        {
            d->c = 2 * d->c;
            d->t = realloc(d->t, d->c * sizeof(int));
        }
        assert(d->t != NULL);
    }
    d->t[d->l] = x;
    d->l++;
```

```c
}

void tableau_dynamique_detruire(tableau_dynamique *d)
{
    free(d->t);
    d->t = NULL;
    d->c = 0;
    d->l = 0;
}

void tableau_dynamique_print(tableau_dynamique d)
{
    printf("c=%zu\tl=%zu\t|", d.c, d.l);
    for (size_t i = 0; i < d.l; i++)
    {
        printf("%d|", d.t[i]);
    }
    for (size_t i = d.l; i < d.c; i++)
    {
        printf(" |");
    }
    printf("\n");
}

int main(void)
{
    tableau_dynamique d = tableau_dynamique_creer();
    tableau_dynamique_print(d);
    for (int i = 1; i < 6; i++)
    {
        tableau_dynamique_ajout(&d, i);
        tableau_dynamique_print(d);
    }
    tableau_dynamique_detruire(&d);
    return 0;
}
```

Ce programme, une fois exécuté, produit la sortie suivante qui permet de retrouver exactement le comportement attendu :

```
c=0 l=0 |
c=1 l=1 |1|
c=2 l=2 |1|2|
c=4 l=3 |1|2|3| |
c=4 l=4 |1|2|3|4|
c=8 l=5 |1|2|3|4|5| | | |
```



Remarque 35.3 La fonction `realloc` recopie, si nécessaire, le contenu de l'ancienne zone dans la nouvelle : c'est cette recopie de l éléments qui fait le coût d'un doublement.

Avec une analyse en pire cas, l'ajout d'un élément peut entraîner un doublement de taille et une recopie, donc coûter $O(c)$ opérations où c est la capacité avant doublement. Cependant, on constate que ce doublement permettra plusieurs ajouts ensuite en temps constant, ainsi, le pire cas ne semble pas le plus pertinent.

35.2 Techniques de calcul

35.2.1 Cadre

On considère une structure de données, pour simplifier initialement vide, et une succession $o_1, \ldots, o_n$ d'opérations **valides** sur cette structure de données. En reprenant l'exemple précédent, cela peut être une succession d'ajouts ou de suppressions dans une file mais sans jamais essayer de retirer des éléments à une file vide.

On considère de plus que pour chaque opération o_i, on connaît le coût temporel c_i en termes d'opérations élémentaires. La question que l'on se pose est d'estimer $\frac{1}{n}\sum_{i=1}^{n} c_i$, c'est-à-dire le coût moyen de chaque

opération. L'objectif est de montrer que ce coût moyen est $O(f(n))$, avec assez souvent $O(1)$, c'est-à-dire que $\sum_{i=1}^{n} c_i = O(nf(n))$, **quelle que soit** la suite d'opérations valides considérée. On dit alors que les opérations ont une complexité amortie en $O(f(n))$.

35.2.2 Calcul naïf

Une approche naïve consiste à calculer directement la somme des coûts. On parle aussi de **méthode de l'agrégat**. Tout le problème ici est qu'on ne connaît pas les opérations et l'ordre dans lequel elles ont été effectuées.

Exemple de la file avec deux piles

On va commencer par calculer c_i pour les différentes opérations :

- S'il s'agit d'un ajout sur `in` ou d'un dépilement sur `out`, on a $c_i = 1$.
- Si l'opération est un dépilement nécessitant un renversement, on a $c_i = 2p + 1$ où $p \geq 1$ est la taille de `in` : p dépilements de `in`, p empilements sur `out` puis un dépilement de `out`.

Le problème du calcul naïf ici est de connaître l'ordre des opérations. On peut regrouper les opérations $o_1, ..., o_n$ par paquets : ajouts, suppression avec renversement, ajouts ou suppressions sans renversement, suppression avec renversement... On va noter r le nombre de renversements et p_i la taille de `in` avant le i-ème renversement. On a donc p_1 ajouts avant le premier renversement et p_{i+1} ajouts entre le i-ème et le $(i+1)$-ème renversement. Comme `out` doit être vide pour qu'un renversement ait lieu, on a effectué exactement $p_i - 1$ suppressions sans renversement entre les deux.

Les m éventuelles opérations qui suivent le dernier renversement coûtent chacune 1 : elles ajoutent m au coût total comme au nombre d'opérations. On commence par supposer que o_n est la suppression provoquant le r-ième renversement, c'est-à-dire $m = 0$.

On obtient alors

$$\begin{aligned}\sum_{i=1}^{n} c_i &= p_1 + (2p_1 + 1) + p_2 + (p_1 - 1) + (2p_2 + 1) \\ &\quad + ... + p_r + (p_{r-1} - 1) + (2p_r + 1) \\ &= 3(p_1 + ... + p_r) + (p_1 + ... + p_{r-1}) + 1 \\ &\leq 4(p_1 + p_2 + ... + p_r)\end{aligned}$$

car $p_r \geq 1$.

Pour conclure, il est nécessaire de relier le nombre d'opérations aux p_i. Or, on a

$$\begin{aligned}n &= p_1 + 1 + p_2 + (p_1 - 1) + 1 + ... + p_r + (p_{r-1} - 1) + 1 \\ &= 2p_1 + 2p_2 + ... + 2p_{r-1} + p_r + 1 \geq p_1 + ... + p_r\end{aligned}$$

Ainsi $\frac{1}{n}\sum_{i=1}^{n} c_i \leq 4$. Dans le cas général, en notant $S = p_1 + ... + p_r$, on a $\sum_{i=1}^{n} c_i \leq 4S + m$ et $n \geq S + m$, donc on a encore $\frac{1}{n}\sum_{i=1}^{n} c_i \leq 4$ (et c'est immédiat s'il n'y a aucun renversement, puisque toutes les opérations coûtent alors 1).

> **Remarque 35.4** Cette analyse est assez pénible à mettre en œuvre car elle demande de compter avec précision ce qui se passe.

Exemple du tableau dynamique

Ici, les opérations sont toutes des ajouts d'éléments, mais certains provoquent une recopie tandis que d'autres sont en temps constant. On compte une unité pour l'écriture du nouvel élément et une unité par élément recopié. La suite des c_i est alors toujours $1, 2, 3, 1, 5, 1, 1, 1, 9, 1, 1, 1, ...$.

Plus précisément, $c_i = \begin{cases} 2^p+1 \text{ si } i=2^p+1 \text{ avec } p\in\mathbb{N} \\ 1 \quad \text{sinon} \end{cases}$.

Donc si $n \geq 1$ et $p = \lceil \log_2 n \rceil$, c'est-à-dire $2^{p-1} < n \leq 2^p$, les indices $i \leq n$ de la forme $2^k + 1$ sont ceux pour lesquels $k \in [\![0, p-1]\!]$, et on a $\sum_{i=1}^{n} c_i = n + \sum_{k=0}^{p-1} 2^k = n + 2^p - 1 < 3n$, car $2^p < 2n$.

On en déduit que $\frac{1}{n}\sum_{i=1}^{n} c_i \leq 3$ c'est-à-dire que c'est un $O(1)$.

Améliorer la méthode naïve

Une amélioration de la méthode naïve va consister à considérer non pas le coût réel c_i de l'opération o_i mais un coût amorti c_i' tel qu'à la fin des opérations on ait $\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} c_i'$ où cette dernière somme est facile à majorer.

On fera souvent en sorte que la majoration soit valide tout au long des opérations, c'est-à-dire que

$$\forall k \in [\![1, n]\!], \sum_{i=1}^{k} c_i \leq \sum_{i=1}^{k} c_i'$$

35.2.3 Méthode du banquier

On va imaginer maintenant que les opérations coûtent un certain nombre de crédits, comme si l'ordinateur marchait avec des pièces de monnaie. Cette méthode est aussi appelée **méthode comptable**.

Des opérations vont alors accumuler des crédits dans une réserve et d'autres opérations vont consommer ces crédits. L'idée est de payer un peu plus pour chaque opération peu coûteuse afin de couvrir les coûts des opérations coûteuses.

Précisément, on considère pour chaque opération o_i :

- une quantité $a_i \geq 0$ de crédits qui seront accumulés dans la réserve ;
- et une quantité $d_i \geq 0$ de crédits qu'il faudra dépenser.

On pose alors $c_i' = c_i + a_i - d_i$. On s'arrange pour faire en sorte que $c_i' \leq Af(n)$ où A est une constante. Dans le cas d'une complexité constante amortie, on s'assurera que $c_i' \leq A$.

On va maintenir un invariant sur les crédits qui est que

$$\forall k \in [\![1, n]\!], \sum_{i=1}^{k} a_i \geq \sum_{i=1}^{k} d_i$$

Cela signifie qu'à tout moment la réserve de crédits est positive ou nulle : on ne dépense pas de crédits n'ayant pas été accumulés.

On a alors directement la majoration voulue de $\sum_i c_i$: soit $k \in [\![1, n]\!]$,

$$\sum_{i=1}^{k} c_i' = \sum_{i=1}^{k} c_i + \left(\sum_{i=1}^{k} a_i - \sum_{i=1}^{k} d_i\right) \geq \sum_{i=1}^{k} c_i$$

et donc $\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} c_i' \leq nAf(n)$.

Exemple de la file avec deux piles

Pour chaque élément enfilé on va allouer deux crédits correspondant au futur dépilement de `in` puis empilement sur `out` qui auront lieu lors du renversement.

On pose donc pour chaque opération o_i, $c_i' = c_i + a_i - d_i$ où :

- $a_i = 2$ et $d_i = 0$ si l'opération est un ajout ;
- $a_i = d_i = 0$ si l'opération est un défilement ne nécessitant pas de renversement ;
- $a_i = 0$ et $d_i = 2p$ où p est la taille de `in` dans le cas d'un défilement provoquant un renversement.

Soit $k \in [\![1, n]\!]$. On a donc $\sum_{i=1}^{k} a_i = 2A$ où A est le nombre d'ajouts effectués par les k premières opérations. Si ces opérations ont effectué des renversements successifs de $p_1, p_2, \ldots, p_r$ éléments de `in`, on a

$$\begin{aligned}\sum_{i=1}^{k} d_i &= 2p_1 + \ldots + 2p_r \\ &= 2(p_1 + \ldots + p_r) \\ &\leq 2A \\ &= \sum_{i=1}^{k} a_i\end{aligned}$$

La majoration vient du fait qu'un élément n'est ajouté qu'une seule fois dans `in` et ne quitte `in` qu'une seule fois, donc que la somme $p_1 + \ldots + p_r$ des longueurs de `in` lors des renversements ne peut dépasser le nombre A d'éléments empilés.

On a ainsi vérifié l'invariant sur les crédits et on a donc la majoration $\sum_{i=1}^{k} c_i \leq \sum_{i=1}^{k} c'_i$.

Pour conclure, il suffit de calculer c'_i selon les cas :

- Si l'opération est un ajout, on a $c'_i = 1 + 2 - 0 = 3$.
- Si l'opération est une suppression sans renversement, on a $c'_i = 1$.
- Si l'opération est une suppression avec renversement, on a $c'_i = 2p + 1 + 0 - 2p = 1$.

Dans tous les cas, on a donc $c'_i \leq 3$.

On en déduit la majoration $\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} c'_i \leq 3n$ qui est plus fine que la majoration obtenue précédemment.

Exemple du tableau dynamique

On va considérer qu'on alloue 2 crédits pour chaque opération d'ajout d'un élément (sauf pour le premier) :

- un crédit pour la première recopie de cet élément ;
- un crédit pour copier l'élément qui lui est symétrique par rapport à la demi-capacité :

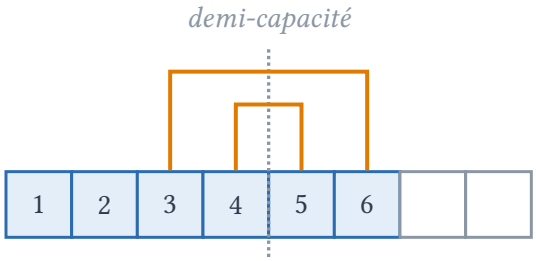


Figure 35.2 — Chaque crédit paie la recopie de l'élément symétrique par rapport à la demi-capacité

Ainsi, juste avant un doublement de capacité, on dispose d'un crédit pour chaque recopie :

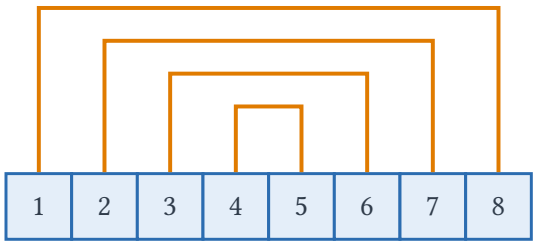


Figure 35.3 — Juste avant un doublement, chaque recopie est payée par un crédit

Plus précisément, on définit pour chaque opération o_i : $c'_i = c_i + a_i - d_i$ où $a_i = \begin{cases}1 \text{ si } i=1 \\ 2 \text{ sinon}\end{cases}$ et $d_i = \begin{cases}2^p \text{ si } i=2^p+1 \text{ avec } p\in\mathbb{N} \\ 0 \text{ sinon}\end{cases}$.

On a donc $c'_1 = 1 + 1 - 0 = 2$, $c'_i = (2^p + 1) + 2 - 2^p = 3$ si $i = 2^p + 1$ et $c'_i = 1 + 2 - 0 = 3$ sinon. Dans tous les cas, $c'_i \leq 3$.

Théorème 35.5

$\forall n \in \mathbb{N}^*, \sum_{i=1}^{n} a_i \geq \sum_{i=1}^{n} d_i$.

Preuve On a $\sum_{i=1}^{n} a_i = 2n - 1$. Si $n = 1$, $\sum_{i=1}^{n} d_i = 0$. Sinon, soit $p \in \mathbb{N}$ tel que $2^p < n \leq 2^{p+1}$, c'est-à-dire $2^p \leq n - 1 < 2^{p+1}$. Les indices $i \leq n$ de la forme $2^k + 1$ sont ceux pour lesquels $k \leq p$, donc

$$\begin{aligned}\sum_{i=1}^{n} d_i &= d_2 + d_3 + d_5 + \ldots + d_{2^p+1} \\ &= 1 + 2 + 2^2 + \ldots + 2^p \\ &= 2^{p+1} - 1 \leq 2(n-1) - 1 \leq 2n - 1\end{aligned}$$

L'inégalité est bien vérifiée.

□

Ainsi,

$$3n \geq \sum_{i=1}^{n} c'_i = \sum_{i=1}^{n} c_i + \left(\sum_{i=1}^{n} a_i - \sum_{i=1}^{n} d_i\right) \geq \sum_{i=1}^{n} c_i$$

On retombe sur une complexité amortie en $O(1)$.

Présentation de la méthode

La présentation qui vient d'être faite de cette méthode est assez lourde bien que précise. En général, on se contente de présenter informellement la distribution de crédits en s'assurant qu'on a couvert les opérations coûteuses.

Une manière de bien s'en assurer est de *décorer* les structures de données en rendant les crédits apparents.

Si on reprend les tableaux dynamiques, on va rajouter sous une case le nombre de crédits dont elle dispose. Si on reprend la séquence d'ajouts précédente on pourra la représenter ainsi (les flèches indiquent les recopies payées par chaque crédit lors d'un doublement) :

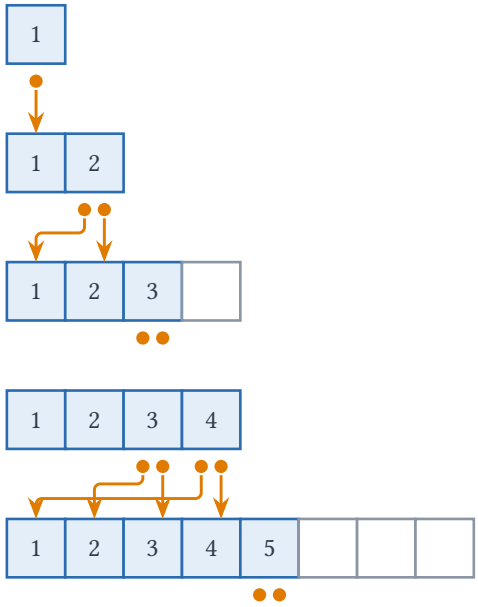


Figure 35.4 — Crédits sous les cases et recopies qu'ils paient lors des doublements

35.2.4 Méthode du potentiel

L'idée de la méthode du potentiel est de définir une fonction Φ qui associe à chaque valeur de la structure de données un nombre réel. Ce réel correspond à la notion physique de potentiel : si on considère un pendule

simple dans sa trajectoire, son énergie potentielle est maximale au sommet quand sa vitesse, et donc son énergie cinétique, est nulle. À l'inverse son énergie potentielle est minimale au point le plus bas et sa vitesse maximale.

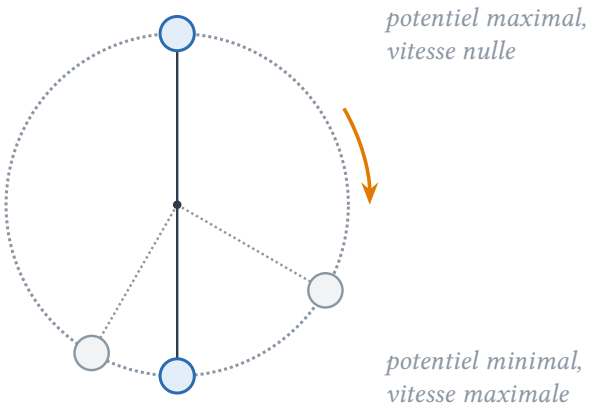


Figure 35.5 — Énergie potentielle d'un pendule : maximale au sommet, minimale en bas

L'idée ici est que le potentiel va correspondre aux gains de temps obtenus par des opérations peu coûteuses. Chaque opération va ainsi faire évoluer le potentiel. On peut prendre l'image d'un empilement de tuiles de bois, comme des Kapla : chaque ajout d'une tuile de bois augmente le potentiel jusqu'à un éventuel effondrement. On retrouve ici la dualité entre les opérations rapides et les opérations coûteuses liées à un basculement.

Précisément, on va noter t_i la structure obtenue à l'issue de l'opération o_i. On considère que t_0 est la structure vide et $\Phi(t_0) = 0$. On pose $c'_i = c_i + \Phi(t_i) - \Phi(t_{i-1})$, ce qui correspond à considérer la différence de potentiel.

On a donc, par télescopage,

$$\sum_{i=1}^{k} c'_i = \sum_{i=1}^{k} c_i + \sum_{i=1}^{k} (\Phi(t_i) - \Phi(t_{i-1})) = \sum_{i=1}^{k} c_i + \Phi(t_k) - \Phi(t_0) = \sum_{i=1}^{k} c_i + \Phi(t_k)$$

Pour conclure par une majoration de $\sum_{i=1}^{k} c_i$, il suffit de s'assurer que le potentiel ne soit jamais négatif : on a alors $\sum_{i=1}^{k} c_i \leq \sum_{i=1}^{k} c'_i$.

Exemple de la file avec deux piles

On va poser $\Phi(t) = 2 \,|\, \text{Fira Sans(in)} \,|$ où $|\, \text{Fira Sans(in)} \,|$ est la taille de la pile d'entrée. Pour reprendre le parallèle physique, on peut s'imaginer que la pile `in` est au-dessus de la pile `out` et que les éléments menacent de s'y déverser, ce qui permet d'avoir l'image du potentiel comme pour le pendule au sommet de sa trajectoire.

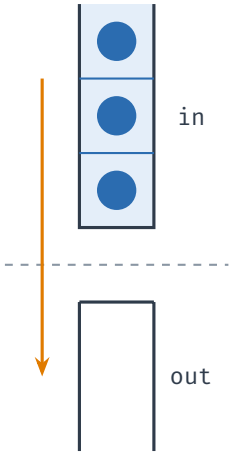


Figure 35.6 — La pile in, au-dessus de la pile out, menace de s'y déverser

On a directement la majoration car $\Phi(t) \geq 0$ et $\Phi(t_0) = 0$.

Pour conclure, il faut calculer c'_i dans chaque cas :

- Si l'opération est un ajout, `in` augmente d'un élément et donc $c'_i = c_i + 2 = 3$.
- Si l'opération est une suppression sans renversement, le potentiel ne change pas et $c'_i = 1$.
- Si l'opération est une suppression avec renversement et que `in` contient p éléments, on a $c'_i = 2p + 1 + 0 - 2p = 1$.

On retombe exactement sur le calcul précédent avec la méthode du banquier.

> **Remarque 35.6** Dans le chapitre Piles et files, on compte un coût $k + 1$ (et non $2k + 1$) pour un renversement de k éléments, et le potentiel | Fira Sans(in) | suffit alors : c'est le même raisonnement avec un autre modèle de coût.

Exemple du tableau dynamique

On pose ici $\Phi(t) = 2l(t) - c(t)$ où $l(t)$ est la longueur de t et $c(t)$ est sa capacité.

Ici, ce potentiel se comprend mieux en écrivant $2(l(t) - c(t)/2)$ car $l(t) - c(t)/2$ correspond au nombre d'éléments ajoutés depuis le dernier doublement.

On a $\Phi(t_0) = 0$ et, comme on ne fait que des ajouts, $c(t) \geq l(t) \geq \frac{c(t)}{2}$ (juste après un doublement, $l(t) = A + 1$ et $c(t) = 2A$), on a donc $\Phi(t) \geq 0$, ce qui valide la majoration.

On calcule c'_i selon les cas d'ajout :

- Le premier ajout fait passer la capacité de 0 à 1 sans recopie : $c'_1 = 1 + \Phi(t_1) - \Phi(t_0) = 1 + (2 - 1) - 0 = 2$.
- Si c'est un ajout sans doublement, alors $l(t_i) = 1 + l(t_{i-1})$, $c(t_i) = c(t_{i-1})$ et ainsi

$$\begin{aligned} c'_i &= c_i + \Phi(t_i) - \Phi(t_{i-1}) \\ &= 1 + 2l(t_{i-1}) + 2 - c(t_{i-1}) - 2l(t_{i-1}) + c(t_{i-1}) = 3 \end{aligned}$$

- Si c'est un ajout avec doublement de capacité, alors $l(t_{i-1}) = c(t_{i-1}) = A \geq 1, l(t_i) = 1 + l(t_{i-1}) = 1 + A, c(t_i) = 2c(t_{i-1}) = 2A$ et $c_i = A + 1$. On a donc :

$$\begin{aligned} c'_i &= c_i + \Phi(t_i) - \Phi(t_{i-1}) \\ &= A + 1 + 2(A + 1) - 2A - 2A + A \\ &= 3 \end{aligned}$$

Ainsi, $3n \geq \sum_{i=1}^{n} c'_i \geq \sum_{i=1}^{n} c_i$. On retombe sur la majoration voulue.

Comparatif des méthodes

On remarque que la méthode du potentiel et la méthode du banquier sont très proches. En fait, on peut montrer qu'elles sont équivalentes : la réserve de crédits après k opérations, $\sum_{i=1}^{k}(a_i - d_i)$, joue le rôle de $\Phi(t_k)$, et réciproquement un potentiel définit les crédits $a_i - d_i = \Phi(t_i) - \Phi(t_{i-1})$. La différence entre les deux est que la méthode du banquier demande de regarder avec précision les opérations pour lesquelles il va falloir allouer des crédits alors que pour la méthode du potentiel il suffit de trouver son expression, qui ne dépend que de l'état de la structure. Cependant, trouver l'expression du potentiel est parfois plus complexe que de définir les crédits.

On cherchera donc la méthode la mieux adaptée aux problèmes considérés. Dans l'exemple de la file, le potentiel est très facilement défini ; dans le cas du tableau dynamique, la méthode du banquier peut paraître plus naturelle.

CHAPITRE 36

Algorithmique exacte

DANS CE CHAPITRE

36.1 Stratégies de résolution
36.2 Compléments de programmation dynamique

Source image : https://www.flickr.com/photos/x6e38/3440634940/

Remarque 36.1 Dans ce chapitre, on étudie des problèmes pour lesquels on va exprimer des algorithmes permettant d'obtenir des solutions exactes. C'est à contraster avec le chapitre sur l'algorithmique approchée.

Une grande partie des stratégies de résolution est basée sur la résolution de sous-problèmes. On verra ainsi trois types de stratégies résumées dans le schéma suivant :

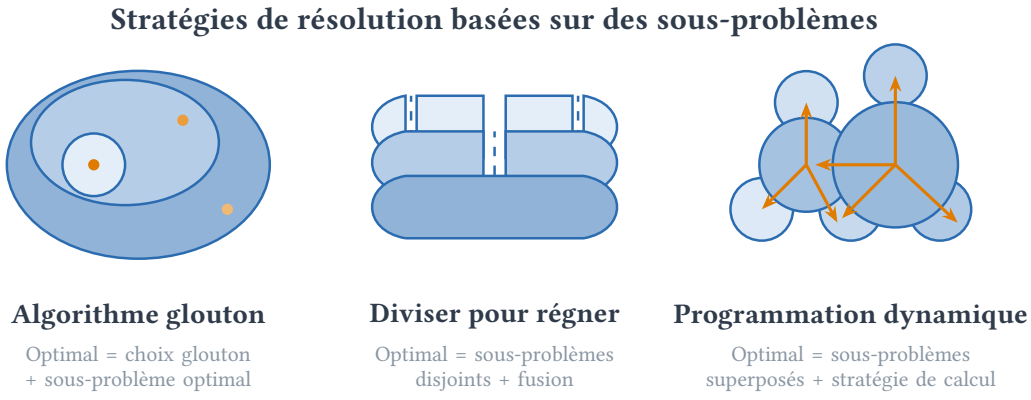


Figure 36.1 — Trois stratégies fondées sur des sous-problèmes.

36.1 Stratégies de résolution

Ces stratégies se comparent bien sur l'arbre des positions qu'on construit en prenant des décisions successives, décrit au début du chapitre sur les algorithmes gloutons et dans le résumé du chapitre sur la programmation dynamique.

- **Recherche exhaustive.** On explore tout l'arbre, en abandonnant au plus tôt les branches qui ne peuvent pas aboutir (retour sur trace). C'est toujours possible, mais l'arbre est le plus souvent de taille exponentielle. Voir le chapitre sur la recherche par force brute.
- **Algorithme glouton.** On ne suit qu'**une** branche, en faisant à chaque étape le choix qui semble localement le meilleur. C'est très efficace, mais il faut **démontrer** que ce choix mène à une solution optimale

(choix glouton et sous-problème optimal), ce qui est faux en général. Voir le chapitre sur les algorithmes gloutons.

- **Diviser pour régner.** On découpe le problème en sous-problèmes **disjoints**, résolus récursivement, dont on fusionne les solutions. Voir le chapitre diviser pour régner.
- **Programmation dynamique.** Quand les sous-problèmes **se superposent**, c'est-à-dire quand on atteint une même position par plusieurs chemins, on calcule la valeur optimale une seule fois par sous-problème, par mémoïsation ou par tabulation, à l'aide d'une équation de récurrence (équation de Bellman). On explore ainsi le **graphe** des positions plutôt que l'arbre. Voir le chapitre sur la programmation dynamique, qui traite le principe général (mémoïsation, tabulation, reconstruction) et de nombreux exemples (chemins monotones, produit de matrices, plus longue sous-séquence croissante, distance d'édition), ainsi que le TP Python sur la somme de sous-ensembles.

Exemple 36.2

Le rendu de monnaie illustre ces différences. Le glouton qui rend toujours la plus grosse pièce possible n'est pas optimal pour le système $(1, 2, 7, 10)$: il rend $14 = 10 + 2 + 2$ alors que $14 = 7 + 7$. La recherche exhaustive trouve l'optimum mais en temps exponentiel, alors que la programmation dynamique calcule le nombre minimal de pièces pour chaque montant de 0 à n, en temps $O(np)$ pour p pièces.

36.2 Compléments de programmation dynamique

Deux exemples classiques supplémentaires, qui suivent la méthode du chapitre sur la programmation dynamique.

36.2.1 Ordonnancement de tâches pondérées

On reprend le problème de la sélection d'activités en associant à chaque activité un poids, par exemple le gain qu'elle rapporte. On ne cherche plus à maximiser le nombre d'activités sélectionnées mais leur poids total. Il ne faut pas confondre ce problème avec l'ordonnancement de tâches avec dates limites du chapitre glouton.

Problème 36.3 — : ordonnancement de tâches pondérées

- Entrée : n activités $(d_1, f_1, w_1), \ldots, (d_n, f_n, w_n)$ avec $d_i < f_i$ et $w_i \in \mathbb{N}$.
- Sortie : un ensemble indépendant $I \subset [\![1, n]\!]$, c'est-à-dire sans deux activités en conflit, tel que $\sum_{i \in I} w_i$ soit maximal.

Comme dans le chapitre glouton, (d_i, f_i) et (d_j, f_j) sont en conflit quand $[d_i, f_i[\cap [d_j, f_j[\neq \emptyset$. Si tous les poids valent 1, on retrouve la sélection d'activités.

Exemple 36.4

On considère les activités suivantes, déjà triées par temps de fin :

i	1	2	3	4
d_i	0	3	2	4
f_i	3	5	6	7
w_i	3	1	6	4

- Le glouton de la sélection d'activités choisit $\{1, 2\}$ de poids 4.
- Le glouton qui choisit les activités par poids décroissant choisit $\{3\}$ de poids 6, car 3 est en conflit avec toutes les autres.
- L'optimum est $\{1, 4\}$ de poids 7.

On trie les activités par temps de fin croissant : $f_1 \leq \ldots \leq f_n$. On s'intéresse alors à la dernière activité n : soit on ne la prend pas, soit on la prend et on ne peut alors garder que des activités finissant avant d_n. Pour $j \in [\![1, n]\!]$, on note

$$\text{pred}\,(j) = \max\left(\{0\} \cup \left\{i \in [\![1, j-1]\!] \mid f_i \leq d_j\right\}\right)$$

Comme les f_i sont croissants, les activités $i < j$ qui finissent avant d_j sont exactement $1, \ldots, \text{pred}\,(j)$.

Pour $j \in [\![0, n]\!]$, on note $\text{Opt}\,(j)$ le poids maximal d'un ensemble indépendant inclus dans $[\![1, j]\!]$.

Théorème 36.5

$\text{Opt}\,(0) = 0$ et pour $j \in [\![1, n]\!]$, $\text{Opt}\,(j) = \max\big(\text{Opt}\,(j-1), w_j + \text{Opt}\,(\text{pred}\,(j))\big)$.

Preuve Soit $j \geq 1$ et $I \subset [\![1, j]\!]$ un ensemble indépendant de poids $\text{Opt}\,(j)$.

- Si $j \notin I$, alors $I \subset [\![1, j-1]\!]$ et $\text{Opt}\,(j) \leq \text{Opt}\,(j-1)$.
- Si $j \in I$, soit $i \in I \setminus \{j\}$. On a $i < j$ donc $f_i \leq f_j$. Si on avait $f_i > d_j$, comme $d_i < f_i \leq f_j$, les activités i et j seraient en conflit. Donc $f_i \leq d_j$, c'est-à-dire $i \leq \text{pred}\,(j)$. Ainsi $I \setminus \{j\}$ est un ensemble indépendant inclus dans $[\![1, \text{pred}\,(j)]\!]$ et $\text{Opt}\,(j) \leq w_j + \text{Opt}\,(\text{pred}\,(j))$.

Réciproquement, un ensemble indépendant de $[\![1, j-1]\!]$ est un ensemble indépendant de $[\![1, j]\!]$, et si J est un ensemble indépendant de $[\![1, \text{pred}\,(j)]\!]$, alors $J \cup \{j\}$ est indépendant car toutes les activités de J finissent avant d_j. Donc $\text{Opt}\,(j)$ est supérieur aux deux valeurs du maximum.

□

Exemple 36.6

Sur l'exemple précédent, $\text{pred} = (0, 1, 0, 1)$ et on obtient successivement $\text{Opt} = (0, 3, 4, 6, 7)$. Pour reconstruire, on part de $j = 4$: $\text{Opt}\,(4) \neq \text{Opt}\,(3)$ donc on prend 4 et on continue en $\text{pred}\,(4) = 1$, où $\text{Opt}\,(1) \neq \text{Opt}\,(0)$ donc on prend 1.

Comme les temps de fin sont triés, on calcule $\text{pred}\,(j)$ par dichotomie. Dans le code, les activités sont numérotées à partir de 0 et `pred.(j)` est le nombre d'activités parmi les $j-1$ premières qui finissent avant le début de la j-ième.

OCaml

```
type activite = { debut : int; fin : int; poids : int }

let ordonnancement_pondere activites =
  let a = Array.copy activites in
  Array.sort (fun x y -> compare x.fin y.fin) a;
  let n = Array.length a in
  (* pred.(j) : nombre d'activités a.(0..k-1) finissant avant le début
     de a.(j-1), obtenu par dichotomie *)
  let pred = Array.make (n + 1) 0 in
  for j = 1 to n do
    let d = a.(j-1).debut in
    (* invariant : a.(0..g-1) finissent avant d,
                   a.(dr..j-2) finissent après d *)
    let g = ref 0 and dr = ref (j - 1) in
    while !g < !dr do
      let m = (!g + !dr) / 2 in
      if a.(m).fin <= d then g := m + 1 else dr := m
    done;
    pred.(j) <- !g
  done;
  (* opt.(j) : poids maximal pour les activités a.(0..j-1) *)
  let opt = Array.make (n + 1) 0 in
  for j = 1 to n do
    opt.(j) <- max opt.(j-1) (a.(j-1).poids + opt.(pred.(j)))
  done;
```

```
  let rec reconstruit j acc =
    if j = 0 then acc
    else if opt.(j) = opt.(j-1) then reconstruit (j-1) acc
    else reconstruit pred.(j) (a.(j-1) :: acc)
  in
  opt.(n), reconstruit n []
```

C

```c
#include <stdlib.h>
#include <stdbool.h>

typedef struct {
    int debut;
    int fin;
    int poids;
} activite;

int compare_activites(const void *x, const void *y)
{
    return ((const activite *)x)->fin - ((const activite *)y)->fin;
}

/* trie a selon les temps de fin et marque les activités choisies */
int ordonnancement_pondere(activite a[], int n, bool choisi[])
{
    qsort(a, n, sizeof(activite), compare_activites);
    int *pred = malloc(sizeof(int) * (n + 1));
    int *opt = malloc(sizeof(int) * (n + 1));
    opt[0] = 0;
    for (int j = 1; j <= n; j++)
    {
        /* dichotomie : nombre d'activités finissant avant a[j-1].debut */
        int g = 0, d = j - 1;
        while (g < d)
        {
            int m = (g + d) / 2;
            if (a[m].fin <= a[j - 1].debut)
                g = m + 1;
            else
                d = m;
        }
        pred[j] = g;
        int avec = a[j - 1].poids + opt[pred[j]];
        opt[j] = avec > opt[j - 1] ? avec : opt[j - 1];
    }
    /* reconstruction */
    for (int i = 0; i < n; i++)
        choisi[i] = false;
    int j = n;
    while (j > 0)
    {
        if (opt[j] == opt[j - 1])
            j--;
        else
        {
            choisi[j - 1] = true;
            j = pred[j];
        }
    }
    int res = opt[n];
    free(pred);
    free(opt);
    return res;
}
```

Python

```python
def ordonnancement_pondere(activites):
    """activites est une liste de triplets (debut, fin, poids)"""
    a = sorted(activites, key=lambda act: act[1])
    n = len(a)
    # pred[j] : nombre d'activités de a[0..j-2] finissant avant le début
    # de a[j-1], obtenu par dichotomie
    pred = [0] * (n + 1)
    for j in range(1, n + 1):
        g, d = 0, j - 1
        while g < d:
            m = (g + d) // 2
            if a[m][1] <= a[j-1][0]:
                g = m + 1
            else:
                d = m
        pred[j] = g
    # opt[j] : poids maximal pour les activités a[0..j-1]
    opt = [0] * (n + 1)
    for j in range(1, n + 1):
        opt[j] = max(opt[j-1], a[j-1][2] + opt[pred[j]])
    # reconstruction
    res = []
    j = n
    while j > 0:
        if opt[j] == opt[j-1]:
            j = j - 1
        else:
            res.append(a[j-1])
            j = pred[j]
    return opt[n], res[::-1]
```

Le tri et les n dichotomies sont en $O(n \log_2 n)$, le remplissage de `opt` et la reconstruction en $O(n)$, d'où une complexité temporelle en $O(n \log_2 n)$.

36.2.2 Plus longue sous-suite commune

Définition 36.7

Soit $u = u_1 \ldots u_n$ un mot. On dit que w est une **sous-suite** (ou un mot extrait) de u s'il existe des indices $1 \leq i_1 < \ldots < i_k \leq n$ tels que $w = u_{i_1} \ldots u_{i_k}$.

Une **plus longue sous-suite commune** (plssc) à deux mots u et v est un mot de longueur maximale qui est une sous-suite de u et de v.

Exemple 36.8

`BCBA` est une plssc de `ABCBDAB` et de `BDCABA`. Ce n'est pas la seule : `BDAB` convient aussi.

Remarque 36.9 Une sous-suite n'est pas un facteur : les lettres ne sont pas forcément consécutives. Ce problème intervient par exemple dans la commande `diff`, qui affiche les différences entre deux fichiers en calculant une plssc de leurs suites de lignes, ou en bio-informatique pour comparer des séquences d'ADN.

Comme il y a 2^n sous-suites de u, on cherche des sous-problèmes. On regarde les dernières lettres de u et v et on se ramène à des préfixes. Pour $u = u_1 \ldots u_n$ et $v = v_1 \ldots v_m$, on note, pour $i \in [\![0, n]\!]$ et $j \in [\![0, m]\!]$, $L(i, j)$ la longueur d'une plssc de $u_1 \ldots u_i$ et $v_1 \ldots v_j$.

Théorème 36.10

- Pour tous i, j, $L(i, 0) = L(0, j) = 0$.
- Pour $i \geq 1$ et $j \geq 1$,

$$L(i, j) = \{1 + L(i-1, j-1) \text{ si } u_i = v_j$$
$$\max(L(i-1, j), L(i, j-1)) \text{ sinon}$$

Preuve Le premier point vient de ce que la seule sous-suite du mot vide est le mot vide. Soient $i, j \geq 1$ et $w = w_1 \ldots w_k$ une plssc de $u_1 \ldots u_i$ et $v_1 \ldots v_j$.

- Si $u_i = v_j$. Si w' est une sous-suite commune à $u_1 \ldots u_{i-1}$ et $v_1 \ldots v_{j-1}$, alors $w'u_i$ est une sous-suite commune à $u_1 \ldots u_i$ et $v_1 \ldots v_j$, donc $L(i,j) \geq 1 + L(i-1, j-1)$. Réciproquement, on a forcément $k \geq 1$ car u_i est une sous-suite commune. Soient $i_1 < \ldots < i_k \leq i$ des indices réalisant w comme sous-suite de $u_1 \ldots u_i$. Si $w_k \neq u_i$, alors wu_i serait une sous-suite commune de longueur $k+1$, ce qui est absurde. Donc $w_k = u_i$ et on peut remplacer i_k par i : les indices restent strictement croissants. Ainsi $w_1 \ldots w_{k-1}$ est une sous-suite de $u_1 \ldots u_{i-1}$ et, de même, de $v_1 \ldots v_{j-1}$. Donc $k - 1 \leq L(i-1, j-1)$.
- Si $u_i \neq v_j$. Toute sous-suite commune à $u_1 \ldots u_{i-1}$ et $v_1 \ldots v_j$ (ou à $u_1 \ldots u_i$ et $v_1 \ldots v_{j-1}$) est une sous-suite commune à $u_1 \ldots u_i$ et $v_1 \ldots v_j$, d'où l'inégalité $\geq$. Réciproquement, si $k \geq 1$, w_k ne peut pas être à la fois égale à u_i et à v_j. Si $w_k \neq u_i$, le dernier indice de w dans u est strictement inférieur à i et w est une sous-suite commune à $u_1 \ldots u_{i-1}$ et $v_1 \ldots v_j$, donc $k \leq L(i-1, j)$. Sinon, $w_k \neq v_j$ et, de même, $k \leq L(i, j-1)$. Si $k = 0$, c'est direct.

□

Chaque valeur $L(i,j)$ dépend de valeurs situées sur la ligne précédente ou à gauche sur la même ligne : on remplit donc le tableau ligne par ligne, de gauche à droite. Pour reconstruire une plssc, on part de (n,m) et on remonte en suivant l'égalité qui a donné $L(i,j)$, ce qui fournit les lettres de la fin vers le début.

OCaml

```
let plssc u v =
  let n = String.length u and m = String.length v in
  (* l.(i).(j) : longueur d'une plssc de u[0..i-1] et v[0..j-1] *)
  let l = Array.make_matrix (n + 1) (m + 1) 0 in
  for i = 1 to n do
    for j = 1 to m do
      l.(i).(j) <-
        if u.[i-1] = v.[j-1] then 1 + l.(i-1).(j-1)
        else max l.(i-1).(j) l.(i).(j-1)
    done
  done;
  (* reconstruction en partant de la fin *)
  let rec reconstruit i j acc =
    if i = 0 || j = 0 then acc
    else if u.[i-1] = v.[j-1] then reconstruit (i-1) (j-1) (u.[i-1] :: acc)
    else if l.(i-1).(j) >= l.(i).(j-1) then reconstruit (i-1) j acc
    else reconstruit i (j-1) acc
  in
  String.of_seq (List.to_seq (reconstruit n m []))
```

C

```
#include <stdlib.h>
#include <string.h>

char *plssc(const char *u, const char *v)
{
    int n = strlen(u), m = strlen(v);
    /* l[i * (m+1) + j] : longueur d'une plssc de u[0..i-1] et v[0..j-1] */
    int *l = malloc(sizeof(int) * (n + 1) * (m + 1));
    for (int i = 0; i <= n; i++)
        for (int j = 0; j <= m; j++)
        {
            if (i == 0 || j == 0)
                l[i * (m + 1) + j] = 0;
            else if (u[i - 1] == v[j - 1])
                l[i * (m + 1) + j] = 1 + l[(i - 1) * (m + 1) + j - 1];
            else
            {
                int a = l[(i - 1) * (m + 1) + j], b = l[i * (m + 1) + j - 1];
                l[i * (m + 1) + j] = a > b ? a : b;
            }
```

```
        }
    /* reconstruction en partant de la fin */
    int k = l[n * (m + 1) + m];
    char *w = malloc(k + 1);
    w[k] = '\0';
    int i = n, j = m;
    while (i > 0 && j > 0)
    {
        if (u[i - 1] == v[j - 1])
        {
            w[--k] = u[i - 1];
            i--;
            j--;
        }
        else if (l[(i - 1) * (m + 1) + j] >= l[i * (m + 1) + j - 1])
            i--;
        else
            j--;
    }
    free(l);
    return w;
}
```

Python

```
def plssc(u, v):
    n, m = len(u), len(v)
    # l[i][j] : longueur d'une plssc de u[0..i-1] et v[0..j-1]
    l = [[0] * (m + 1) for _ in range(n + 1)]
    for i in range(1, n + 1):
        for j in range(1, m + 1):
            if u[i-1] == v[j-1]:
                l[i][j] = 1 + l[i-1][j-1]
            else:
                l[i][j] = max(l[i-1][j], l[i][j-1])
    # reconstruction en partant de la fin
    res = []
    i, j = n, m
    while i > 0 and j > 0:
        if u[i-1] == v[j-1]:
            res.append(u[i-1])
            i, j = i - 1, j - 1
        elif l[i-1][j] >= l[i][j-1]:
            i = i - 1
        else:
            j = j - 1
    return ''.join(reversed(res))
```

La complexité temporelle et spatiale est en $O(nm)$ et la reconstruction en $O(n+m)$ puisque $i+j$ diminue à chaque étape.

CHAPITRE 37

Recherche par force brute

DANS CE CHAPITRE

37.1 Principe
37.2 Recherche par retour sur trace (*backtracking*)
37.3 Stratégies d'énumération
37.4 Droite de balayage

Source image : https://www.flickr.com/photos/x6e38/3440634940/

37.1 Principe

37.1.1 Problème de décision et exploration exhaustive

Considérons un problème du type trouver un $x \in V$ vérifiant une propriété $P(x)$.

Exemple 37.1

- V est l'ensemble des chaînes de caractères et P vérifie si une chaîne est un mot de passe qu'on cherche.
- V est l'ensemble des indices possibles dans un tableau `t` et P vérifie si la valeur à l'indice i est une valeur x que l'on cherche.
- V est l'ensemble des grilles complétées d'un problème de Sudoku et P vérifie si la grille est valide.
- V est l'ensemble d'assemblages de pièces d'un puzzle et P vérifie si le puzzle est correct, c'est-à-dire si deux pièces côte à côte ont des côtés compatibles.

Dans certains problèmes, un tel x n'est pas unique et on cherche à tous les énumérer.

Une recherche par force brute, ou recherche exhaustive, consiste à énumérer l'ensemble V jusqu'à obtenir une solution en testant P pour chaque valeur rencontrée.

Des problèmes précédents, la recherche linéaire est le plus simple et le programme suivant est caractéristique d'une recherche exhaustive :

```
exception Trouve of int

let recherche t x =
    try
        for i = 0 to Array.length t - 1 do
            if t.(i) = x
            then raise (Trouve i)
        done;
        raise Not_found
    with Trouve i -> i
```

La forme usuelle sera alors

```
pour chaque v dans V
    si P(v) est vérifié
       s'arrêter avec la solution v
```

On rappelle que pour pouvoir s'arrêter au cours de l'énumération en OCaml, si on programme en impératif, on utilise en général des exceptions comme dans le chapitre Exceptions en OCaml.

Cela pose naturellement la question de l'énumération des éléments de V. Si c'est immédiat dans l'exemple peu pertinent de la recherche dans un tableau, c'est beaucoup plus complexe pour l'énumération des assemblages de pièces d'un puzzle, par exemple.

Pour la recherche du mot de passe, on pourrait commencer par énumérer les chaînes de longueur 1, puis de longueur 2, et ainsi de suite.

Le plus souvent, l'ensemble V est fini (pour les mots de passe, cela peut consister à limiter la longueur maximale du mot de passe). Ainsi, une recherche par force brute effectue $O(\mid V \mid)$ itérations, chacune comportant un test de P.

37.1.2 Problème d'optimisation et exploration exhaustive

On retrouve la notion d'exploration exhaustive ou force brute pour des problèmes d'optimisation. Il s'agit de problèmes de la forme : déterminer $x \in V$ tel que $f(x)$ soit minimale ou maximale.

L'exploration exhaustive consiste alors à calculer toutes les images par f des éléments de V afin de déterminer un extremum.

37.2 Recherche par retour sur trace (*backtracking*)

37.2.1 Construction itérative de candidats

Dans de nombreux cas, l'ensemble V peut se décrire par un processus itératif de construction de ses éléments. Une manière de voir cela est de parler de positions et de mouvements, ou coups.

Par exemple, considérons un puzzle comme le puzzle Eternity II constitué de 256 pièces carrées. Une configuration finale du puzzle consiste à avoir placé les 256 pièces. Parmi celles-ci, les configurations valides sont celles satisfaisant les contraintes de chaque côté.

Comme pour tous les puzzles, la position initiale est un plateau vide et chaque mouvement consiste à placer une pièce disponible dans un emplacement disponible. Cela correspond à la manière dont on procéderait à la main :



Ainsi, on peut représenter la construction de V sous la forme d'un arbre dont les nœuds sont les positions et les arêtes les mouvements. Les positions complètes sont les feuilles de l'arbre, elles correspondent aux éléments de V et ce sont donc celles-ci qu'on va explorer pour y trouver une solution.

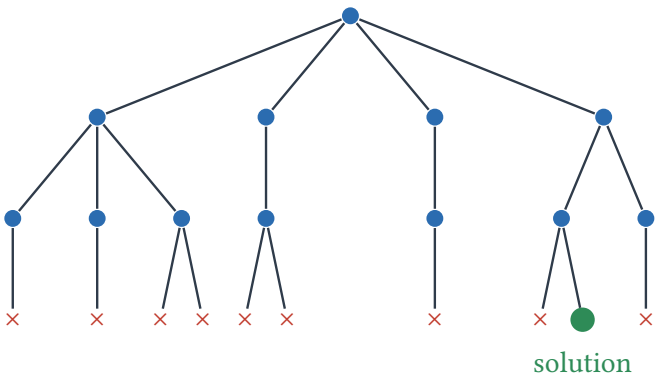


Figure 37.1 — Arbre de construction de V : les feuilles sont les positions complètes.

L'avantage de cette représentation arborescente est qu'elle découle naturellement d'un parcours récursif des positions.

Tout d'abord, il va falloir définir un type de positions partielles comme on a pu le voir dans le chapitre Options en OCaml.

Comme indiqué dans ce chapitre, si le type d'une grille de Sudoku remplie est `int array array`, pour pouvoir représenter des grilles en cours de remplissage, on va utiliser le type `int option array array`. On rappelle que l'utilisation de la valeur `None` permettra de représenter une partie non construite comme une case vide.

On considère ainsi un type `position` des positions partielles, un type `mouvement` et des fonctions

```
(* les mouvements accessibles depuis une position.
  [] si la position est complète *)
val mouvements : position -> mouvement list
(* applique un mouvement *)
val applique : position -> mouvement -> position
(* vérifie si une position est complète *)
val complete : position -> bool
(* vérifie si une position complète est valide *)
val valide : position -> bool
```

Schématiquement, un algorithme d'énumération aura la structure suivante en OCaml :

```
exception Solution of position

let rec enumere pos =
    if complete pos
    then begin
        if valide pos
        then raise (Solution pos)
    end else
        List.iter (fun mouv -> enumere (applique pos mouv))
            (mouvements pos)
```

Remarque 37.2 Ici, l'usage de `List.iter` permet de faire l'équivalent d'une boucle for mais sur une liste. Pour s'en passer, on peut écrire une fonction auxiliaire récursive ou, quand les mouvements sont en petit nombre, une conversion en `array` suivie d'une boucle `for`.

Si la position initiale est `pos0`, on pourra résoudre le problème ainsi :

```
let resout () =
    try
        enumere pos0;
        raise Not_found
    with Solution pos -> pos
```

Cet algorithme de parcours des solutions est appelé le *retour sur trace*, ou *backtracking* en anglais. Il tire parti de la récursivité pour remonter les positions après avoir essayé en vain une construction.

Remarque 37.3 On peut formaliser ce cadre, comme on le fera dans le chapitre sur les algorithmes gloutons : on considère l'ensemble **fini** S des positions, la position initiale s_0 (ici `pos0`) et

- mouv $(s) = \{\texttt{applique}\ s\ m \mid m \in \texttt{mouvements}\ s\}$;
- taille (s) le nombre de mouvements effectués depuis s_0 pour obtenir s (par exemple le nombre de pièces posées ou de cases remplies), de sorte que taille $(s_0) = 0$ et que $x \in \text{mouv}\,(y) \Rightarrow \text{taille}\,(x) > \text{taille}\,(y)$.

Cette dernière propriété assure la terminaison de `enumere` : la quantité $\max_{s \in S} \text{taille}\,(s) - \text{taille}\,(\texttt{pos})$ est un entier naturel qui décroît strictement à chaque appel récursif. Les positions finales, celles où mouv $(s) = \emptyset$, sont ici les positions complètes. Par une récurrence sur cette même quantité, on montre que `enumere pos` lève l'exception `Solution` si et seulement si une position complète valide est accessible depuis `pos`.

Remarque 37.4 Il est possible de reprendre la construction précédente avec des données mutables et une manière de *défaire* les mouvements :

```
val applique : position -> mouvement -> unit
val defaire : position -> mouvement -> unit

let rec enumere pos =
    if complete pos
    then begin
        if valide pos
```

```
        then raise (Solution pos)
    end else
        List.iter (fun mouv ->
                    applique pos mouv;
                    enumere pos;
                    defaire pos mouv)
            (mouvements pos)
```

37.2.2 Évaluation partielle et raccourcis

Si on reprend la construction itérative précédente, on se rend compte qu'elle n'est pas très intelligente : faut-il remplir l'intégralité d'un puzzle avant de se rendre compte qu'il est invalide en raison des deux premières pièces ?

On peut donc raffiner l'approche précédente en introduisant une notion de mouvements valides qui sont les mouvements qui préservent la correction partielle.

En pratique, il suffit de remplacer la fonction `mouvements` par une fonction `mouvements_valides`.

✍ Pour que la recherche reste correcte, il faut que cet élagage ne fasse perdre aucune solution : un mouvement ne doit être écarté que si **aucune** position complète valide n'est accessible depuis la position qu'il produit. C'est le cas quand la correction partielle est une propriété nécessaire, préservée en remontant vers la racine : si une position complète est valide, toutes les positions partielles rencontrées pour la construire sont partiellement correctes. En revanche, avec des mouvements valides, une position non complète peut n'avoir aucun mouvement possible : c'est une impasse et on remonte.

C'est un changement réduit qui peut avoir un grand impact sur l'arbre des positions. En reprenant l'illustration précédente, on peut imaginer que cela reviendrait à ne pas parcourir les mouvements rouges et les sous-arbres associés :

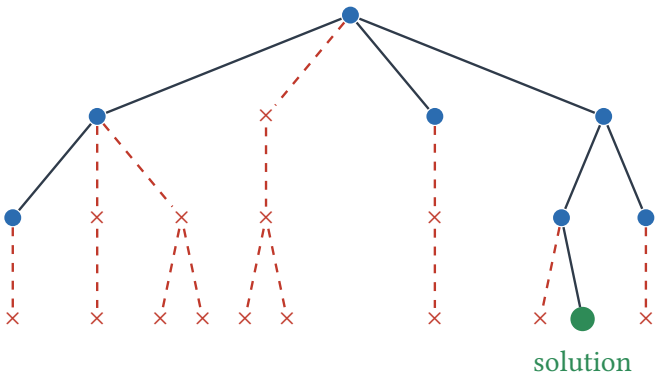


Figure 37.2 — Avec des mouvements valides, les branches rouges et leurs sous-arbres ne sont pas explorés.

Problème : résolution de Sudoku

La recherche par retour sur trace se prête très bien à la résolution de problèmes comme le Sudoku. On va ici tout simplement tenter de remplir chaque case du haut vers le bas tant qu'on satisfait les contraintes du Sudoku. Le programme sera ainsi très proche de la résolution des huit reines présentée plus loin.

Commençons par rappeler le principe du Sudoku :

- On part d'une grille de 81 cases réparties en une grille de 3x3 sous-grilles de 3x3 cases et comportant des chiffres de 1 à 9 dans certaines cases.

1								6
		6		2		7		
7	8	9	4	5		1		3
			8		7			4
				3				
	9				4	2		1
3	1	2	9	7			4	
	4			1	2		7	8
9		8						

Figure 37.3 — Une grille de Sudoku.

- L'objectif est de remplir chaque case avec un chiffre de 1 à 9 de sorte que chaque ligne, chaque colonne et chaque sous-grille 3x3 comporte une et une seule fois chaque chiffre.
- Par convention, une grille de Sudoku bien posée admet une unique solution (c'est le cas de la grille précédente).

Pour représenter une grille de Sudoku en OCaml on utilise un `(int option) array array`, la valeur `None` signifiant que la case est vide et la valeur `Some x` qu'elle est remplie avec la valeur x.

```
type grille = (int option) array array
```

On fait le choix de représenter la grille par un tableau de lignes, ce qui signifie que pour accéder à la case de coordonnées (x, y) dans `g` il faut écrire `g.(y).(x)`.

Le problème donné précédemment est alors représenté par la valeur suivante :

```
let probleme = [|
    [| Some 1; None; None;    None; None; None;   None; None; Some 6 |];
    [| None; None; Some 6;    None; Some 2; None;   Some 7; None; None |];
    [| Some 7; Some 8; Some 9;    Some 4; Some 5; None;   Some 1; None; Some 3 |];

    [| None; None; None;    Some 8; None; Some 7;   None; None; Some 4 |];
    [| None; None; None;    None; Some 3; None;   None; None; None |];
    [| None; Some 9; None;    None; None; Some 4;   Some 2; None; Some 1 |];

    [| Some 3; Some 1; Some 2;    Some 9; Some 7; None;   None; Some 4; None |];
    [| None; Some 4; None;    None; Some 1; Some 2;   None; Some 7; Some 8 |];
    [| Some 9; None; Some 8;    None; None; None;   None; None; None |];
    |]
```

Afin de définir la fonction de résolution, on définit une première fonction `suivant` de signature :

```
val suivant : grille -> (int * int) -> (int * int) option
```

telle que l'appel à `suivant g (x,y)` renvoie `Some (xi,yi)` quand (x_i, y_i) sont les coordonnées de la prochaine case libre, dans l'ordre gauche à droite puis haut vers bas, à partir de (x, y) inclus, ou `None` quand il n'existe pas de telle case libre. Cela signifie alors que la grille est entièrement remplie.

```
let rec suivant g (x,y) =
    if y > 8
    then None
    else if g.(y).(x) = None
    then Some (x,y)
```

```
    else if x < 8 then suivant g (x+1, y)
    else suivant g (0, y+1)
```

On définit également une fonction `valide` de signature

```
val valide : grille -> int -> int -> bool
```

telle que l'appel à `valide g x y` renvoie `true` si et seulement si la valeur placée en coordonnées (x, y) n'invalide pas la grille, en supposant qu'elle était valide avant ce placement. Ne pas prendre cette valeur en paramètre permet d'écrire un peu plus simplement cette fonction. La fonction est assez directe : étant donné (x, y), on va parcourir sa ligne, sa colonne et sa sous-grille pour vérifier qu'un nombre n'a pas été placé deux fois à l'aide d'un tableau de drapeaux :

```
let valide g x y =
    let v = ref true in
    let vus_colonne = Array.make 9 false in
    for y0 = 0 to 8 do
        match g.(y0).(x) with
        | None -> ()
        | Some k ->
                if vus_colonne.(k-1)
                then v := false;
                vus_colonne.(k-1) <- true
    done;
    let vus_ligne = Array.make 9 false in
    for x0 = 0 to 8 do
        match g.(y).(x0) with
        | None -> ()
        | Some k ->
                if vus_ligne.(k-1)
                then v := false;
                vus_ligne.(k-1) <- true
    done;
    let vus_grille = Array.make 9 false in
    let xb = (x / 3) * 3 in
    let yb = (y / 3) * 3 in
    for xd = 0 to 2 do
        for yd = 0 to 2 do
            match g.(yb+yd).(xb+xd) with
            | None -> ()
            | Some k ->
                    if vus_grille.(k-1)
                    then v := false;
                    vus_grille.(k-1) <- true
        done
    done;
    !v
```

On peut alors définir la fonction `resout` qui va résoudre le Sudoku en effectuant tous les remplissages tant qu'on a une grille valide. Dès qu'une solution est trouvée, on s'arrête. Pour cela, on utilise le mécanisme des exceptions pour permettre une sortie prématurée. On a fait le choix de travailler en place dans la grille, ainsi à la fin de l'exécution de la fonction, la grille correspond à la solution.

```
exception Solution

let resout g =
    let rec aux xi yi = match suivant g (xi, yi) with
        | None -> raise Solution
        | Some (x,y) ->
            for i = 1 to 9 do
                g.(y).(x) <- Some i;
                if valide g x y
                then begin
                    aux x y
```

```
                end
            done;
            g.(y).(x) <- None
    in
    try
        aux 0 0
    with Solution -> ()
```

Remarque 37.5 Quand l'exception `Solution` est levée, l'instruction `g.(y).(x) <- None` n'est pas exécutée : c'est ce qui permet de conserver la grille remplie. Si la grille n'a pas de solution, toutes les cases libres sont remises à `None` et `resout` termine normalement en laissant la grille inchangée.

La résolution du Sudoku donné précédemment par ce programme est présentée dans la vidéo suivante :

37.2.3 Énumération de toutes les solutions

Le problème précédent du Sudoku n'avait par définition qu'une unique solution. Cependant, il existe des problèmes pour lesquels plusieurs solutions existent et pour lesquels on souhaite les énumérer.

La fonction précédente pourra alors devenir :

```
let rec enumere pos =
    if complete pos
    then begin
        if valide pos
        then [ pos ]
        else []
    end else
        List.concat
            (List.map (fun mouv -> enumere (applique pos mouv))
                (mouvements_valides pos))
```

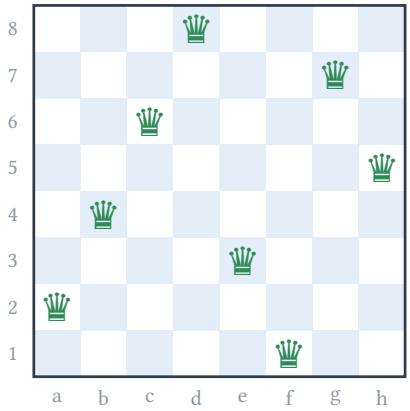
Remarque 37.6 La fonction `List.concat` permet de concaténer une `'a list list` d'un coup comme dans :

```
### List.concat [ [1;2]; [3]; []; [4;5;6] ];;
- : int list = [1; 2; 3; 4; 5; 6]
```

Problème des huit reines

L'exemple classique de ce problème est celui des huit reines : étant donné un échiquier, peut-on placer huit reines de sorte qu'aucune reine ne puisse prendre une autre reine ? Plus précisément : sur un plateau de 8x8 cases, peut-on placer huit pions tels que deux pions quelconques ne soient jamais sur la même ligne, la même colonne ou la même diagonale ?

Exemple de solution :



Ce problème admet effectivement des solutions partielles en ne considérant que k reines à placer. Pour énumérer les solutions, on peut même se contenter de solutions partielles où les k reines sont placées sur les k premières rangées, à raison d'une par rangée (deux reines sur une même rangée se prendraient).

Voici ainsi un algorithme pour énumérer les solutions :

- Supposons que k reines aient été placées et qu'on dispose d'une solution partielle.
  - Si $k = 8$ alors toutes les reines sont placées et la solution est complète, on la comptabilise.
  - Sinon, on continue la recherche pour chaque position de la $(k+1)$-ième reine sur la $(k+1)$-ième rangée qui préserve le fait d'être une solution partielle.

Ici, quand on dit qu'on continue la recherche, ce qu'on signifie, c'est qu'on effectue un appel récursif.

Pour programmer cette méthode, on va définir une fonction récursive de signature :

```
val resout_reines : (int * int) list -> (int * int) list list
```

Un appel à `resout_reines part` va ainsi renvoyer la liste des solutions complètes construites à partir de la solution partielle `part`. Les solutions sont représentées par des listes de couples de coordonnées (x, y) sur l'échiquier, donc dans $[\![0, 7]\!]^2$, où x est la colonne et y la rangée.

Voici une implémentation où on explore les solutions à l'aide d'une boucle impérative dans l'appel récursif. La fonction `valide` permet de tester si le placement d'une reine est possible avant d'effectuer un appel : comme les reines de `part` sont sur des rangées différentes de celle de la nouvelle reine, il suffit de tester les colonnes et les diagonales.

```
let rec valide (x1,y1) l =
    match l with
    | [] -> true
    | (x2,y2)::q ->
        x1 <> x2 && abs (x2-x1) <> abs(y2-y1) && valide (x1,y1) q

let rec resout_reines part =
    let k = List.length part in
    if k = 8
    then [ part ]
    else begin
        let resultats = ref [] in
        for x = 0 to 7 do
            let essai = (x,k) :: part in
            if valide (x,k) part
            then begin
                resultats := (resout_reines essai) @ !resultats;
            end
        done;
        !resultats
    end
```

et, ici, une autre implémentation purement récursive à l'aide d'une fonction auxiliaire récursive.

```
let rec resout_reines part =
    let k = List.length part in
    if k = 8
    then [ part ]
    else
        let rec aux x acc =
            if x < 0
            then acc
            else let essai = (x,k) :: part in
                 let nacc = if valide (x,k) part
                            then (resout_reines essai) @ acc
                            else acc in
                    aux (x-1) nacc
```

```
        in
        aux 7 []
```

L'appel `resout_reines []` renvoie bien les 92 solutions du problème.

Une partie de l'arbre de recherche, celle issue de la position où les deux premières reines sont placées en f1 et c2, est présentée sur l'image suivante :

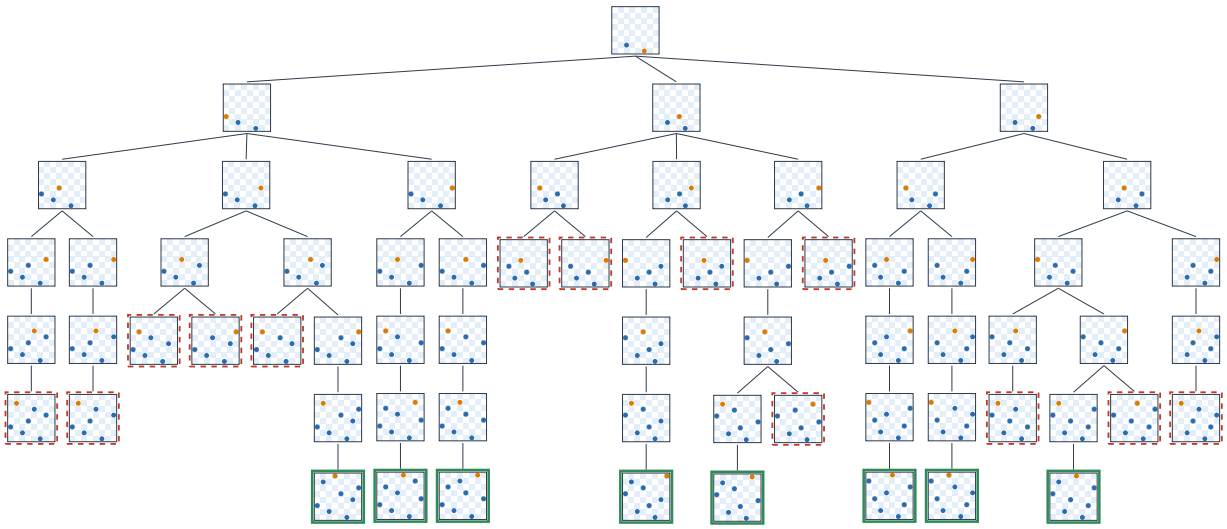


Figure 37.4 — Sous-arbre issu de f1, c2 : solutions en vert, impasses en rouge, dernière reine posée en orange.

L'arbre complet comporte 2057 nœuds (racine comprise) et 736 feuilles : 92 feuilles de profondeur 8, qui correspondent aux solutions du problème, et 644 impasses, positions partielles depuis lesquelles on ne peut placer aucune reine supplémentaire. À titre de comparaison, l'arbre exhaustif correspondant à faire tous les choix de placement à raison d'une reine par rangée compterait $8^8 = 16777216$ feuilles et $\sum_{i=0}^{8} 8^i = 19173961$ nœuds. On voit bien que le backtracking est plus économe en exploration.



Remarque 37.7 En remplaçant 8 par n (taille de l'échiquier et nombre de reines), le même programme compte les solutions du problème des n reines. On obtient les valeurs connues : 1, 0, 0, 2, 10, 4, 40, 92, 352, 724 solutions pour $n = 1, \ldots, 10$.

37.2.4 TP : jeu du solitaire

On considère ici le jeu du solitaire. On a un plateau comportant 33 emplacements et initialement 32 pions, représentés par des ronds noirs, et un emplacement vide au centre :



Figure 37.5 — Position initiale : 32 pions et un emplacement vide au centre.

Les différents mouvements possibles consistent à passer d'une configuration • • ◦ à ◦ ◦ • et ainsi à diminuer d'un pion le nombre total de pions. Ces configurations peuvent être rencontrées dans les directions horizontales ou verticales.

On considère que la partie est gagnée quand il n'y a plus qu'un pion sur le plateau.

Première implémentation naïve

On va représenter un plateau par le type OCaml :

```
type case = Vide | Pion | Invalide
type plateau = case array array
let n = 7
(* Un plateau est une matrice 7x7 avec des cases invalides aux coins *)
```

Question 1 Définir une fonction `print_plateau` qui affiche un plateau sous un format textuel lisible.

Corrigé

```
let print_plateau p =
    for j = 0 to n-1 do
        for i = 0 to n-1 do
            let c = match p.(i).(j) with
            | Invalide -> ' '
            | Vide -> '.'
            | Pion -> '#'
            in Printf.printf "%c" c
        done;
        print_newline ()
    done;
    print_newline ()
```

Question 2 Écrire une fonction `plateau_initial : unit -> plateau` qui renvoie un plateau correspondant à la configuration de départ.

Corrigé

```
let plateau_initial () =
    [|
        [| Invalide; Invalide; Pion; Pion; Pion; Invalide; Invalide |];
        [| Invalide; Invalide; Pion; Pion; Pion; Invalide; Invalide |];
        [| Pion; Pion; Pion; Pion; Pion; Pion; Pion |];
        [| Pion; Pion; Pion; Vide; Pion; Pion; Pion |];
        [| Pion; Pion; Pion; Pion; Pion; Pion; Pion |];
        [| Invalide; Invalide; Pion; Pion; Pion; Invalide; Invalide |];
        [| Invalide; Invalide; Pion; Pion; Pion; Invalide; Invalide |]
    |]
```

Un mouvement peut être assimilé à un triplet de coordonnées décrivant, dans l'ordre $\bullet\bullet\circ$, les trois cases concernées. Il se trouve que la case centrale est toujours le milieu des deux autres, on peut donc se contenter de donner un couple de coordonnées pour décrire un mouvement.

```
type mouvement = (int * int) * (int * int)
```

Question 3 Écrire une fonction `mouvements : plateau -> mouvement list` qui renvoie la liste des mouvements possibles sur le plateau passé en paramètre.

Corrigé

```
let possible p (i1,j1) (i3,j3) =
    let i2, j2 = (i1+i3)/2, (j1+j3)/2 in
    p.(i1).(j1) = Pion && p.(i2).(j2) = Pion && p.(i3).(j3) = Vide

let mouvements p =
    (* on utilise une pile *)
    let l = ref [] in
```

```
    for i = 0 to n-1 do
        for j = 0 to n-3 do
            if possible p (i,j) (i,j+2)
            then l := ( (i,j), (i,j+2) ) :: !l;
            if possible p (i,j+2) (i,j)
            then l := ( (i,j+2), (i,j) ) :: !l;
            if possible p (j+2,i) (j,i)
            then l := ( (j+2,i), (j,i) ) :: !l;
            if possible p (j,i) (j+2,i)
            then l := ( (j,i), (j+2,i) ) :: !l
        done
    done;
    !l
```

Question 4 Écrire une fonction `compte_pions : plateau -> int` qui renvoie le nombre de pions sur un plateau.

En déduire une fonction `valide : plateau -> bool` qui indique si un plateau correspond à une partie gagnante.

Corrigé

```
let compte_pions p =
    let c = ref 0 in
    for i = 0 to n-1 do
        for j = 0 to n-1 do
            if p.(i).(j) = Pion
            then incr c
        done
    done;
    !c

let valide p = compte_pions p = 1
```

Question 5 Écrire deux fonctions `applique` et `defaire` de type `plateau -> mouvement -> unit` permettant de faire et défaire un mouvement.

Corrigé

```
let applique p mouv =
    let (i1,j1), (i3,j3) = mouv in
    let i2, j2 = (i1+i3)/2, (j1+j3)/2 in
    p.(i1).(j1) <- Vide;
    p.(i2).(j2) <- Vide;
    p.(i3).(j3) <- Pion

let defaire p mouv =
    let (i1,j1), (i3,j3) = mouv in
    let i2, j2 = (i1+i3)/2, (j1+j3)/2 in
    p.(i1).(j1) <- Pion;
    p.(i2).(j2) <- Pion;
    p.(i3).(j3) <- Vide
```

Question 6 Écrire une fonction `enumere : plateau -> mouvement list -> unit` telle que `enumere pos chemin` énumère les plateaux accessibles depuis `pos` jusqu'à obtenir une solution, sachant que `chemin` est la liste de mouvements, du plus récent au plus ancien, qui ont conduit jusqu'à `pos`.

En cas de succès, on lèvera une exception `Solution of mouvement list` renvoyant la liste des mouvements ayant conduit à une solution.

En déduire une fonction `resout : unit -> mouvement list` qui renvoie une liste de mouvements permettant de résoudre le solitaire. On prendra garde à renverser le chemin obtenu pour que le premier mouvement de la liste soit le premier mouvement effectué.

Corrigé

```
exception Solution of mouvement list

let rec enumere pos chemin =
    if valide pos
    then raise (Solution chemin)
    else let l = mouvements pos in
        List.iter (fun mouv ->
                    applique pos mouv;
                    enumere pos (mouv :: chemin);
                    defaire pos mouv) l

let resout () =
    let pos = plateau_initial () in
    try
        enumere pos [];
        raise Not_found
    with Solution l -> List.rev l
```

Ce code ne permet pas de calculer la solution car il prend beaucoup trop de temps en raison du nombre de positions étudiées.

Cache des mauvaises positions

On se rend compte que de nombreuses positions sont réétudiées alors qu'on sait déjà qu'elles ne peuvent permettre d'aboutir à une solution. En effet, il y a souvent des coups indépendants pouvant être joués au même moment, ce qui fait qu'on peut aboutir à une même position de beaucoup de manières différentes, ce qui augmente considérablement le nombre d'appels récursifs.

Une stratégie consiste à maintenir un ensemble de configurations mauvaises. Pour réaliser un tel ensemble, on va utiliser une table de hachage dont les clés sont les positions et les valeurs `unit`. Si une position a une valeur associée dans la table, c'est qu'elle est mauvaise.

Cela pose la question de la représentation persistante et immuable des positions. Une première stratégie peut consister à transformer le plateau en `case list list`. Cette stratégie est beaucoup trop coûteuse et elle ne permet pas de répondre instantanément. On va profiter du fait qu'il n'y ait que 49 cases dans le plateau pour le représenter par un entier sur 49 bits : $a_{00} + a_{10}2 + a_{20}2^2 + \ldots + a_{60}2^6 + a_{01}2^7 + \ldots + a_{66}2^{48}$ où a_{ij} vaut 1 lorsqu'il y a un pion sur la j-ième ligne et la i-ième colonne, c'est-à-dire quand `p.(i).(j) = Pion`, et 0 sinon.

Indication : l'entier `1 lsl n` est 2^n. `lsl` signifie qu'on décale le chiffre `1` de `n` bits vers la gauche dans son écriture binaire.

Question 7 Écrire une fonction `code : plateau -> int` qui renvoie le numéro associé à un plateau.

Corrigé

```
let code p =
    let c = ref 0 in
    for j = 0 to 6 do
    for i = 0 to 6 do
        if p.(i).(j) = Pion
        then c := !c + 1 lsl (7*j+i)
    done
    done;
    !c
```

Pour manipuler un ensemble, on va définir

```
let mauvaises = Hashtbl.create 42
let ajoute x = Hashtbl.add mauvaises x ()
let contient x = Hashtbl.mem mauvaises x
```

L'appel à `ajoute x` rajoute `x` dans l'ensemble des mauvaises positions et `contient x` vérifie si `x` est dans cet ensemble.

Question 8 Reprendre la fonction `enumere` avec un ensemble de mauvais codes.

Corrigé

```
let rec enumere pos chemin =
    if valide pos
    then raise (Solution chemin)
    else
        let c = code pos in
        if not (contient c)
        then begin
            let l = mouvements pos in
            List.iter (fun mouv ->
                        applique pos mouv;
                        enumere pos (mouv :: chemin);
                        defaire pos mouv) l;
            (* Si on est ici c'est que le noeud ne permet pas de trouver
               une solution *)
            ajoute c
        end
```

Le code permet alors de réaliser la résolution en une fraction de seconde une fois compilé avec `ocamlopt` (quelques secondes dans l'interpréteur) : environ 340 000 positions mauvaises sont mémorisées avant de trouver une solution en 31 coups.

Remarque 37.8 Possibles extensions :

- rajouter un affichage de la résolution
- déterminer la proportion de recalculs évités
- gérer les symétries des plateaux

37.3 Stratégies d'énumération

37.3.1 Énumération d'arbres

Imaginons que l'on souhaite énumérer des arbres binaires non étiquetés pour trouver le premier arbre binaire à n nœuds vérifiant une certaine propriété. On suppose ainsi défini un type

```
type arbre = Nil | Noeud of arbre * arbre
```

et une fonction pour tester le prédicat :

```
val valide : arbre -> bool
```

Une première possibilité est d'effectuer un simple parcours récursif :

```
let rec recherche a =
    valide a
    || ( match a with
         Nil -> false
         | Noeud(g,d) -> recherche g || recherche d)
```

Cette fonction n'est pas récursive terminale et elle va donc échouer, faute d'emplacement suffisant sur la pile, si l'arbre passé en entrée est trop grand et qu'il est nécessaire de l'explorer pleinement.

37.4 Droite de balayage

37.4.1 Principe

Il est parfois possible d'ordonner V pour permettre de trouver une solution plus vite, voire d'ordonner des données de base pour ne pas énumérer V mais énumérer un $V' \subset V$ plus petit.

C'est un procédé classique dans le contexte de la géométrie algorithmique dans le plan : étant donné un ensemble V de candidats qu'on déduit d'un ensemble de points du plan, par exemple l'ensemble des paires de points, on va énumérer V à l'aide d'un parcours des points de gauche à droite (ou toute autre direction géométrique) pour ne pas énumérer tout V mais seulement une partie plus petite. Tout se passe comme si on balayait avec une droite l'ensemble des points, d'où le nom de droite de balayage.

Du point de vue de la complexité temporelle, on obtient le plus souvent un algorithme en $O(n \log n)$ où n est le nombre de points. Cela signifie que l'étape la plus coûteuse en temps est le tri initial.

On va présenter ici deux exemples d'utilisation d'une droite de balayage. Il est assez clair qu'il sera nécessaire dans ces cas de réfléchir géométriquement. Les exemples présentés sont donc assez simples d'un point de vue informatique, mais plutôt complexes d'un point de vue mathématique.

37.4.2 Plus proche paire

On considère le problème

Problème 37.9 — : PlusProchePaire

- Entrée : un ensemble P de $n \geq 2$ points dans le plan.
- Sortie : une paire $\{p, p'\}$ de points de P telle que $\text{dist}(p, p') = \| \overrightarrow{pp'} \|$ soit minimale.

Recherche exhaustive

Considérons le problème *PlusProchePaire* qui, étant donné un ensemble de n points ($n \geq 2$), détermine la paire constituée des deux points les plus proches.

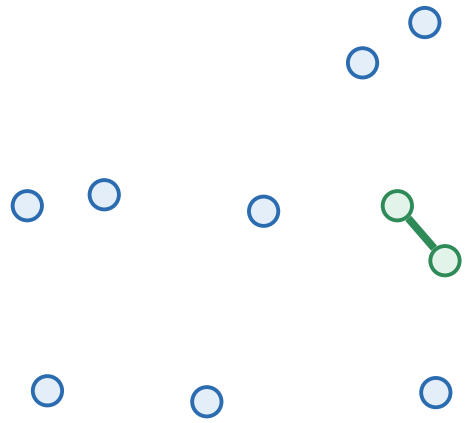


Figure 37.6 — La plus proche paire (en vert) parmi dix points.

Une implémentation naïve de la recherche par force brute consiste à énumérer les $\frac{n(n-1)}{2}$ paires et donc à effectuer $O(n^2)$ itérations.

```
let distance (x1,y1) (x2,y2) =
    sqrt ((x1-.x2)*.(x1-.x2) +. (y1-.y2)*.(y1-.y2))

let plus_proche_paire points =
    let n = Array.length points in
    let min_paire = ref (distance points.(0) points.(1), (0, 1)) in
    for i = 0 to n - 1 do
        for j = i+1 to n - 1 do
            let d = distance points.(i) points.(j) in
```

```
            if d < fst !min_paire
            then min_paire := (d, (i, j))
        done
    done;
    snd !min_paire
```

Raffinement : droite de balayage

Il est parfois possible d'accélérer la recherche par force brute en ordonnant le parcours des candidats pour pouvoir éviter de tester certains d'entre eux.

En géométrie algorithmique, une approche classique consiste à ordonner les objets selon leur abscisse et à parcourir les objets par abscisse croissante. On parle alors de **droite de balayage** (en anglais, *sweep line*) car cela revient à balayer le plan par une droite verticale en ne traitant que les objets avant cette ligne.

Reprenons le problème précédent, on considère que les points sont triés par abscisse croissante : $(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})$. On va parcourir les points dans cet ordre en maintenant un ensemble de points à gauche du point courant, appelés *points actifs*, et en ne calculant que les distances avec les points actifs.

Si on a parcouru les N premiers points et qu'on a obtenu que la plus petite distance était d, lorsqu'on considère le point (x_N, y_N), il est inutile de tester les points qui sont forcément à distance $> d$ de celui-ci. C'est-à-dire qu'on peut éliminer du test les points qui ne sont pas dans le rectangle $[x_N - d, x_N] \times [y_N - d, y_N + d]$. Les points dont l'abscisse est $< x_N - d$ peuvent être éliminés définitivement vu que l'on raisonne par abscisse croissante et que d ne peut que diminuer ; par contre, les points d'ordonnées hors de $[y_N - d, y_N + d]$ doivent être conservés pour les points ultérieurs.

Ce rectangle est représenté sur le schéma suivant ainsi qu'une ligne imaginaire qui correspond à l'abscisse du point courant et qu'on peut imaginer parcourant le plan de gauche à droite pour traiter les points au fur et à mesure.

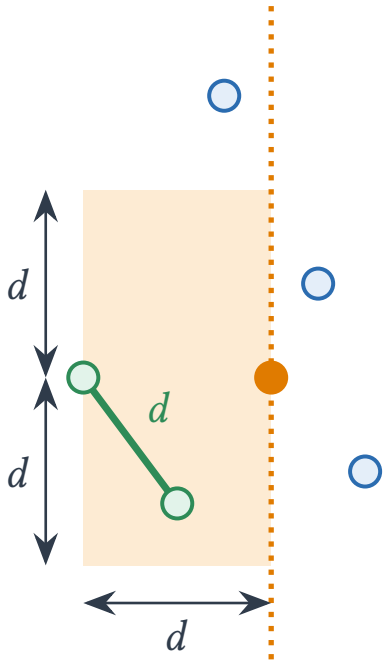


Figure 37.7 — Point courant (orange), rectangle des points actifs à tester, distance minimale d (vert).

Afin de déterminer la complexité de cet algorithme, il est nécessaire de connaître le nombre maximal de points actifs dans le rectangle. Comme ces points ont été pris en compte précédemment, ils sont forcément à distance au moins d les uns des autres. Il s'agit donc de déterminer le nombre maximum de points qu'on peut placer dans ce rectangle à distance au moins d. On remarque tout d'abord qu'on peut placer six points ainsi :

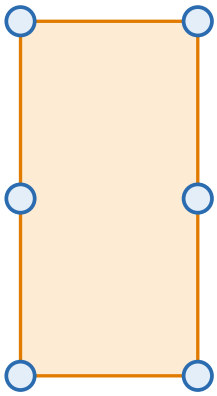


Figure 37.8 — Six points à distance au moins d dans le rectangle.

Si jamais on avait au moins sept points, on peut voir qu'il y a forcément un des six sous-rectangles suivants qui contiendrait au moins deux points :

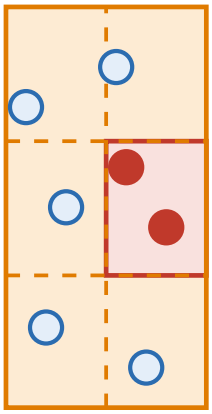


Figure 37.9 — Avec sept points, un des six sous-rectangles en contient deux (en rouge).

Or, ces sous-rectangles sont de longueur $\frac{1}{2}d$ et de hauteur $\frac{2}{3}d$, donc la distance maximale entre deux de leurs points correspond à la longueur des diagonales : $\sqrt{\frac{1}{4}+\frac{4}{9}}d = \frac{5}{6}d < d$.

Il y a donc toujours au plus 6 points actifs dans ce rectangle, et donc au plus 6 distances à calculer pour le point courant.

Voici le principe de l'algorithme que l'on va implémenter :

- On trie le tableau `points` par abscisse croissante. **Complexité :** $O(n \log n)$
- On initialise la plus petite distance `d` courante à la distance entre les deux premiers points
- On crée un ensemble `actifs`, ordonné par les ordonnées, de points contenant initialement les deux premiers points
- Pour chaque point (x, y) en partant du troisième :
  - On supprime les points (x', y') tels que $x' < x - d$ de `actifs`. **Complexité :** sur l'ensemble des itérations on ne pourra jamais supprimer deux fois un point, donc on effectue au maximum n suppressions chacune en $O(\log n)$ donc $O(n \log n)$.
  - On parcourt les points de `actifs` dont les ordonnées sont comprises entre $y - d$ et $y + d$ en mettant à jour `d`. **Complexité :** pour récupérer le premier point de l'ensemble, il faut $O(\log n)$ en pire cas (tous les points actifs) et ensuite on effectue au plus 6 itérations comme on vient de le prouver.
  - On ajoute (x, y) à `actifs`. **Complexité :** $O(\log n)$.

L'animation suivante présente le déroulement de cet algorithme. La bande active est indiquée en gris et le rectangle autour du point courant en gris foncé :

[figure : animation disponible sur le site]

On remarque ainsi que la complexité en temps et en pire cas de cet algorithme est de $O(n \log n)$. Ici, le fait d'avoir la structure `actifs` ordonnée par les ordonnées est crucial pour garantir la complexité. Pour la réalisation d'une structure d'ensemble ordonnée ayant ces complexités, voir le chapitre sur les dictionnaires (arbres binaires de recherche équilibrés).

Ici, on utilise le module `Set` d'OCaml pour réaliser la structure d'ensemble, pour cela on commence par créer le module `PointSet` pour les ensembles de points. Les points sont comparés selon leur ordonnée, puis selon leur abscisse en cas d'égalité : sinon, deux points distincts de même ordonnée seraient considérés comme égaux et l'un d'eux disparaîtrait de l'ensemble.

```
module Point = struct
    type t = float * float
    let compare (x1,y1) (x2,y2) = Stdlib.compare (y1,x1) (y2,x2)
end

module PointSet = Set.Make(Point)
```

Puis on définit une fonction permettant de parcourir les points entre deux ordonnées :

```
let set_iter_entre f set bas haut =
    try
        let e = PointSet.find_first (fun p -> snd p >= bas) set in
        let seq = PointSet.to_seq_from e set in
        let rec aux seq =
            match seq () with
            | Seq.Nil -> ()
            | Seq.Cons (p, seq_suite) ->
                    if snd p <= haut
                    then begin
                        f p;
                        aux seq_suite
                    end
        in aux seq
    with Not_found -> ()
```

On implémente alors assez directement l'algorithme décrit précédemment (le tableau `points` est trié en place) :

```
let plus_proche_paire_balayage points =
    let compare (x1,y1) (x2,y2) =
        if x1 = x2
        then if y1 < y2 then -1 else 1
        else if x1 < x2 then -1 else 1
    in
    Array.sort compare points;
    let n = Array.length points in
    let d = ref (distance points.(0) points.(1)) in
    let couple = ref (points.(0), points.(1)) in
    let actifs = ref (PointSet.empty
            |> PointSet.add points.(0) |> PointSet.add points.(1)) in

    let gauche = ref 0 in

    for i = 2 to n-1 do
        let xi, yi = points.(i) in

        while fst points.(!gauche) < xi -. !d do
            actifs := PointSet.remove points.(!gauche) !actifs;
            incr gauche
```

```
        done;

        set_iter_entre (fun pj ->
            let dip = distance points.(i) pj in
            if dip < !d
            then begin
                couple := (points.(i), pj);
                d := dip
            end) !actifs (yi -. !d) (yi +. !d);

        actifs := PointSet.add points.(i) !actifs
    done;
    !d
```

Remarque 37.10 La fonction renvoie la plus petite distance, le couple de points correspondant étant dans `!couple`. Elle a été testée en la comparant à la recherche exhaustive sur des milliers d'ensembles aléatoires, y compris avec de nombreux points de même abscisse ou de même ordonnée.

Problème : test d'intersection pour un ensemble de segments

Considérons le problème suivant *IntersectionEnsemble* : étant donné n segments dans le plan, il s'agit de déterminer si au moins deux des segments s'intersectent.

Remarque 37.11 On peut considérer ici que l'on dispose d'une fonction

```
intersecte : (float * float) * (float * float)
    -> (float * float) * (float * float) -> bool
```

qui teste l'intersection entre deux segments.

Cependant, il est possible d'écrire une telle fonction avec un peu de géométrie élémentaire.

Si on considère que les deux segments sont $[A_1B_1]$ et $[A_2B_2]$, avec $A_1 \neq B_1$ et $A_2 \neq B_2$, alors chaque point du segment $[A_1B_1]$ est de la forme $A_1 + t\overrightarrow{A_1B_1}$ où $t \in [0,1]$. De même les points du segment $[A_2B_2]$ sont de la forme $A_2 + u\overrightarrow{A_2B_2}$ où $u \in [0,1]$.

S'il y a une intersection, c'est qu'il existe $(t,u) \in [0,1]^2$ tel que

$$A_1 + t\overrightarrow{A_1B_1} = A_2 + u\overrightarrow{A_2B_2} \Leftrightarrow \overrightarrow{A_2A_1} + t\overrightarrow{A_1B_1} = u\overrightarrow{A_2B_2}$$

L'idée est alors d'utiliser une opération appelée **produit vectoriel** sur les vecteurs. Comme ici, tout est plan, le produit vectoriel est uniquement déterminé par sa troisième coordonnée, celle qui sort du plan, et on peut se contenter de calculer celle-ci. On note ainsi $(x,y) \times (x',y') = xy' - yx'$ cette coordonnée, qui est aussi le déterminant des deux vecteurs. On a donc $\vec{v} \times \vec{v} = 0$ et $\vec{v} \times \vec{w} = -\vec{w} \times \vec{v}$.

On peut alors composer l'égalité par $\times\, \overrightarrow{A_2B_2}$:

$$\overrightarrow{A_2A_1} \times \overrightarrow{A_2B_2} + t\left(\overrightarrow{A_1B_1} \times \overrightarrow{A_2B_2}\right) = 0$$

Notons $\Delta = \overrightarrow{A_1B_1} \times \overrightarrow{A_2B_2}$, si $\Delta \neq 0$, alors

$$t = -\frac{\overrightarrow{A_2A_1} \times \overrightarrow{A_2B_2}}{\Delta} = \frac{\overrightarrow{A_1A_2} \times \overrightarrow{A_2B_2}}{\Delta}$$

On procède de même avec $\times\, \overrightarrow{A_1B_1}$ pour obtenir une expression de u : $\overrightarrow{A_2A_1} \times \overrightarrow{A_1B_1} = u\left(\overrightarrow{A_2B_2} \times \overrightarrow{A_1B_1}\right) = -u\Delta$ et donc

$$u = -\frac{\overrightarrow{A_2A_1} \times \overrightarrow{A_1B_1}}{\Delta} = \frac{\overrightarrow{A_1A_2} \times \overrightarrow{A_1B_1}}{\Delta}$$

Si $\Delta \neq 0$, on peut donc alors exprimer u et t et vérifier qu'ils sont dans $[0,1]$.

Si $\Delta = 0$ c'est que les deux segments sont de directions parallèles ou confondues.

- Si $\overrightarrow{A_1A_2} \times \overrightarrow{A_1B_1} \neq 0$ alors $\overrightarrow{A_1A_2}$ et $\overrightarrow{A_1B_1}$ sont non colinéaires donc les deux segments sont sur des droites parallèles distinctes et ne peuvent s'intersecter.
- Sinon, les segments reposent sur une même droite et il s'agit de vérifier leurs positions sur la droite. Pour cela, on exprime $A_2 = A_1 + t_A\overrightarrow{A_1B_1}$ et de même $B_2 = A_1 + t_B\overrightarrow{A_1B_1}$. Plus précisément, on calcule $\overrightarrow{A_1A_2} \cdot \overrightarrow{A_1B_1} = t_A \parallel \overrightarrow{A_1B_1} \parallel^2$ à l'aide du produit scalaire et on a $t_A = \frac{\overrightarrow{A_1A_2}\cdot\overrightarrow{A_1B_1}}{\|\overrightarrow{A_1B_1}\|^2}$. De même, $t_B = \frac{\overrightarrow{A_1B_2}\cdot\overrightarrow{A_1B_1}}{\|\overrightarrow{A_1B_1}\|^2}$. On doit alors vérifier si l'intervalle $[t_A, t_B]$ (ou $[t_B, t_A]$ selon leur position) intersecte $[0, 1]$.

Voici une fonction OCaml qui correspond à ce raisonnement

```
let intersecte (a1,b1) (a2,b2) =
    let vec (x1,y1) (x2,y2) = (x2-.x1,y2-.y1) in
    let cross (x1,y1) (x2,y2) = x1 *. y2 -. y1 *. x2 in
    let dot (x1,y1) (x2,y2) = x1 *. x2 +. y1 *. y2 in
    let proche0 x = let eps = 1e-20 in
        if x < 0. then -.x < eps else x < eps in
    let a1b1 = vec a1 b1 in let a2b2 = vec a2 b2 in
    let a1a2 = vec a1 a2 in let a1b2 = vec a1 b2 in

    let delta = cross a1b1 a2b2 in

    if proche0 delta
    then
         if proche0 (cross a1a2 a1b1)
         then let na1b1 = dot a1b1 a1b1 in (* colinéaires *)
              let tA = (dot a1a2 a1b1) /. na1b1 in
              let tB = (dot a1b2 a1b1) /. na1b1 in
              if tA < tB
              then not (tB < 0. || tA > 1.)
              else not (tA < 0. || tB > 1.)
         else false (* parallèles *)
    else let t = (cross a1a2 a2b2) /. delta in (* se croisent *)
         let u = (cross a1a2 a1b1) /. delta in
         t >= 0. && t <= 1. && u >= 0. && u <= 1.
```

La recherche par force brute va alors énumérer l'ensemble des paires de segments distincts et tester deux à deux les intersections. On peut ainsi écrire le programme suivant qui est assez simple et effectuera effectivement $O(n^2)$ itérations dans le pire cas, c'est-à-dire lorsqu'il n'y a pas d'intersection.

```
exception Trouve

let intersection_ensemble (v: ((float * float) * (float * float)) array) : bool =
    let n = Array.length v in
    try
        for i = 0 to n - 1 do
            for j = i+1 to n-1 do
                if intersecte v.(i) v.(j)
                then raise Trouve
            done
        done;
        false
    with Trouve -> true
```

CHAPITRE 38

Algorithmes gloutons

DANS CE CHAPITRE

38.1 Principe
38.2 Construction de l'arbre de Huffman
38.3 Sélection d'activités
38.4 Principe général des preuves d'optimalité
38.5 Ordonnancement de tâches
38.6 Exercices

38.1 Principe

38.1.1 Problème d'optimisation combinatoire

On considère ici un problème d'énumération comme dans le chapitre sur le retour sur trace : un ensemble **fini** S muni d'une valeur $s_0 \in S$ appelée la **position** initiale et de deux fonctions

$$\text{mouv} : S \to \mathscr{P}(S)$$

$$\text{taille} : S \to \mathbb{N}$$

telles que $\text{taille}(s_0) = 0$ et que

$$\forall x, y \in S, x \in \text{mouv}(y) \Rightarrow \text{taille}(x) > \text{taille}(y)$$

Cette dernière propriété, combinée à la finitude de S, assure qu'en partant de s_0 et en effectuant une série de *mouvements* on aboutit forcément à une position finale, c'est-à-dire pour laquelle $\text{mouv}(s) = \emptyset$. On note $S^f \subset S$ les positions finales.

On peut donc ainsi tracer l'arbre des positions où s_0 est à la racine et les enfants d'une position sont celles données par mouv. Attention, dans cet arbre, on peut distinguer deux positions selon les mouvements qui nous y amènent. En fait, si on partage ces positions, il s'agit d'un graphe orienté acyclique.

Exercice

Prouver cette affirmation.

Sur la représentation suivante, on a fait figurer en vert les éléments de S^f.

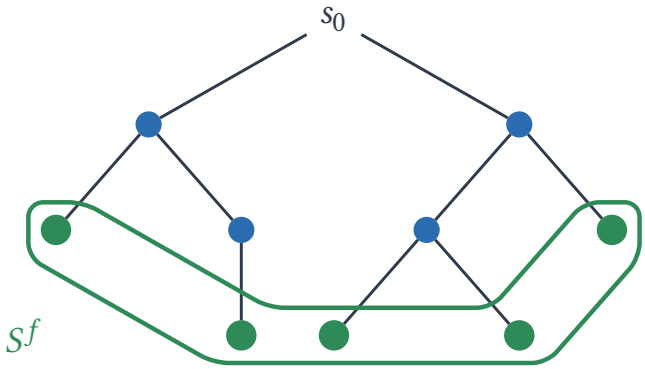


Figure 38.1 — Arbre des positions issues de s_0 ; les positions finales S^f sont en vert.

Dans le retour sur trace, on cherchait une position finale vérifiant une certaine propriété. Ici, on considère une fonction objectif :

$$\chi : S^f \to \mathbb{R}$$

qui associe une valeur numérique à chaque position. On considère un problème dit de maximisation : déterminer $x \in S^f$, tel que $\chi(x) = \max_{y \in S^f} \chi(y)$. On notera souvent $x = \operatorname{argmax}_{y \in S^f} \chi(y)$. Quand le problème considéré dépend d'une entrée E, on note Opt (E) la valeur de l'optimal.

Remarque 38.1

- En considérant $y \mapsto -\chi(y)$, on transforme un problème de maximisation en un problème de minimisation. On parle plus généralement de problèmes d'optimisation combinatoire.
- Il y a une ambiguïté sur $\operatorname{argmax}_{y \in S^f} \chi(y)$ quand plusieurs éléments de S^f réalisent ce maximum. Dans la plupart des algorithmes gloutons qu'on va considérer, on commence par donner un ordre sur S et on considère le plus petit y pour cet ordre réalisant le maximum. L'ordre choisi est alors crucial dans la preuve de correction. C'est aussi une des raisons pour lesquelles les algorithmes gloutons sont souvent de complexité temporelle $O(n \log_2 n)$.

Une première stratégie très élémentaire consiste alors à énumérer S, de manière exhaustive ou avec une stratégie plus fine comme le retour sur trace, puis à déterminer un élément maximal de manière directe.

Cela revient donc à déterminer l'arbre des solutions puis à trouver une feuille maximisant l'objectif :

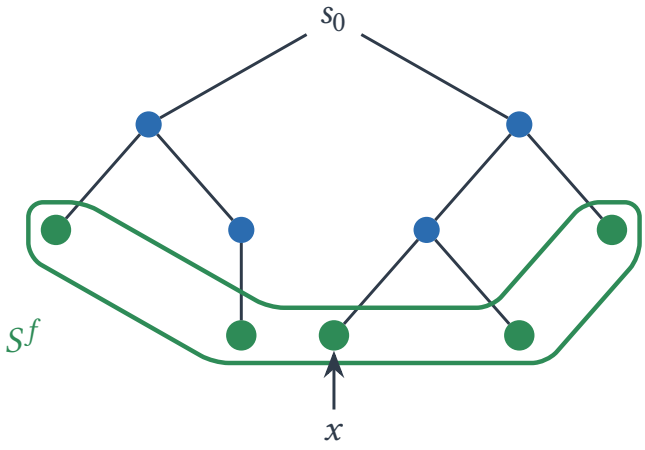


Figure 38.2 — Recherche exhaustive : on parcourt tout l'arbre pour trouver la position finale optimale x.

38.1.2 Algorithme glouton

Le nombre de nœuds de l'arbre est en général exponentiel en la taille de l'entrée. Ainsi, une exploration exhaustive est optimale mais très coûteuse.

Un algorithme glouton va suivre une approche beaucoup plus efficace : à chaque étape de construction de la solution, on choisit la branche qui maximise la fonction d'objectif. Pour cela, on considère que la fonction d'objectif χ se prolonge à toutes les positions dans S.

Si la position courante est $x \in S$, la position suivante sera $y = \text{argmax}_{z \in \text{ mouv }(x)} \chi(z)$.

On dit que y est le **mouvement glouton** ou **choix glouton**.

Sur l'arbre précédent, cela reviendrait à n'emprunter qu'une seule branche indiquée en orange sur le schéma.

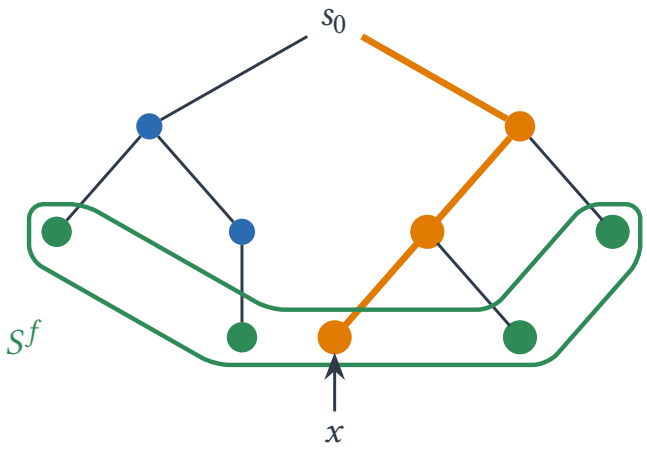


Figure 38.3 — Une suite de choix gloutons (en orange) qui aboutit ici à l'optimum x.

Cela a l'air très efficace mais il y a un problème majeur : il n'y a aucune garantie qu'on aboutisse ainsi à une solution, encore moins à une solution optimale. En effet, on aurait très bien pu faire les choix suivants :

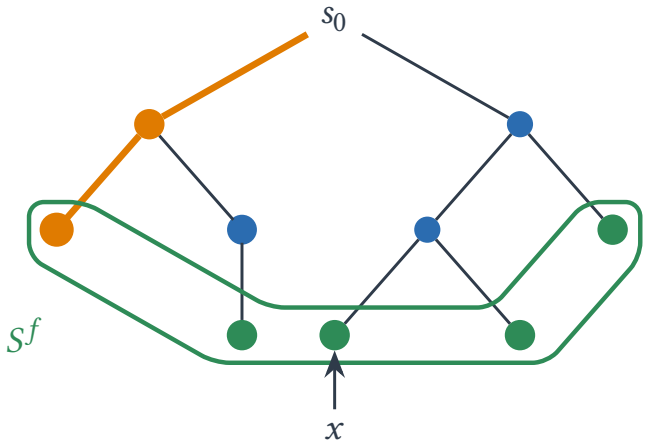


Figure 38.4 — D'autres choix gloutons (en orange) aboutissent à une position finale non optimale.

et ne pas aboutir à une solution.

38.1.3 Cas du rendu de monnaie

Considérons par exemple le problème du **rendu de monnaie** : étant donné une liste de valeurs faciales de pièces $P = (v_1, ..., v_p) \in (\mathbb{N}^*)^p$ avec $1 = v_1 < ... < v_p$ et une somme $n \in \mathbb{N}^*$, on cherche la manière d'exprimer cette somme avec le plus petit nombre de pièces possible.

Plus précisément, l'ensemble des positions est $S = \left\{(k_1, ..., k_p) \in \mathbb{N}^p \mid k_1 v_1 + ... + k_p v_p \leq n\right\}$, l'ensemble des positions finales est $S^f = \left\{(k_1, ..., k_p) \in \mathbb{N}^p \mid k_1 v_1 + ... + k_p v_p = n\right\}$ et la fonction d'objectif est $\chi : (k_1, ..., k_p) \mapsto k_1 + ... + k_p$.

On cherche alors $x = \text{argmin}_{y \in S^f} \chi(y)$.

Comme $1 = v_1$, $S^f \neq \emptyset$ car $(n, 0, ..., 0) \in S^f$ et ainsi $\chi(x) \leq n$.

Exemple 38.2

Si $P = (1, 2, 5, 10)$ et $n = 14$ on a $(3, 3, 1, 0) \in S^f$ car $n = 3 + 3 \times 2 + 1 \times 5$.

L'algorithme glouton va utiliser la plus grande pièce possible à chaque étape : si on doit décomposer n, on cherche $v_i \leq n$ maximale, on décompose $n - v_i$ et on rajoute la pièce i pour en déduire une décomposition de n. On s'arrête quand le nombre à décomposer vaut 0 qui correspond à l'unique rendu vide.

Exemple 38.3

Si $P = (1, 2, 5, 10)$ et $n = 14$.

- On utilise la plus grande pièce possible $10 \leq 14$ puis on exprime $4 = 14 - 10$
- Ici, la plus grande pièce est 2 et on continue avec $2 = 4 - 2$
- La plus grande pièce est encore 2 et on s'arrête car $0 = 2 - 2$.
- En conclusion, on a obtenu $x = (0, 2, 0, 1)$.
- Une exploration exhaustive permet de s'assurer qu'on a effectivement obtenu une décomposition minimale. En effet, ici l'ensemble des décompositions est : { (14,0,0,0), (12,1,0,0), (10,2,0,0), (8,3,0,0), (6,4,0,0), (4,5,0,0), (2,6,0,0), (0,7,0,0), (9,0,1,0), (7,1,1,0), (5,2,1,0), (3,3,1,0), (1,4,1,0), (4,0,2,0), (2,1,2,0), (0,2,2,0), (4,0,0,1), (2,1,0,1), (0,2,0,1) }.

Exemple 38.4

On considère maintenant $P = (1, 2, 7, 10)$ et $n = 14$

- L'algorithme glouton va ici procéder comme dans l'exemple précédent et on va obtenir $x = (0, 2, 0, 1)$.
- Mais on remarque que ce n'est pas un minimum car $x' = (0, 0, 2, 0)$ convient avec $\chi(x') = 2 < 3 = \chi(x)$.

Remarque 38.5 L'algorithme glouton n'est pas nécessairement optimal. Un système de pièces, on parle de système monétaire, tel que l'algorithme glouton soit optimal est appelé un *système canonique*. Un exercice en fin de chapitre demande de démontrer qu'un certain système est canonique.

On peut se poser la question des problèmes pour lesquels l'algorithme glouton aboutit nécessairement à une solution optimale. Il se trouve qu'il y a un cadre général mais hors programme dont on pourra voir se dégager la nature à la lumière des preuves d'optimalité qui vont suivre.

38.2 Construction de l'arbre de Huffman

38.2.1 Description

Remarque 38.6 Ce paragraphe décrit l'étape cruciale du principe de compression de Huffman. Celui-ci sera présenté complètement dans le chapitre Algorithmique des textes.

On va étudier ici un principe de compression parfaite (sans perte d'information à la décompression) de données appelé l'algorithme de Huffman et qui repose sur ce principe simple : coder sur moins de bits les caractères les plus fréquents.

Par exemple si on considère le mot `abaabc`, en le codant avec un nombre de bits fixes, par exemple 2 avec le code `a=00,b=01,c=10`, on aurait besoin de 12 bits pour représenter le mot. Mais si on choisit le code suivant : `a=0,b=10,c=11`, il suffit de 9 bits. On a donc gagné 3 bits soit un facteur de compression de 75%.

On remarque que pour pouvoir décompresser, il n'aurait pas été possible de faire commencer le code de `b` ou `c` par un `0`, sinon on aurait eu une ambiguïté avec la lecture d'un `a`. On parle alors de code préfixe :

Définition 38.7

Soit $X \subset \{0,1\}^*$, on dit que X est un code préfixe lorsque pour tous $x, y \in X$, x n'est pas un préfixe de y et y n'est pas un préfixe de x.

On se pose alors la question du code préfixe optimal pour un texte donné.

Plus précisément, étant donné un alphabet fini Σ et une application $f : \Sigma \to \mathbb{N}^*$ associant à chaque lettre son nombre d'occurrences dans le texte considéré (on ne retient dans Σ que les lettres apparaissant dans le texte). Ainsi $\sum_{x\in\Sigma} f(x)$ est la longueur du texte. On cherche un code préfixe X et une application $c : \Sigma \to X$ telle que $\sum_{x\in\Sigma} f(x) \mid c(x) \mid$ soit minimale car cela correspond au nombre de bits après codage.

Remarque 38.8 On utilise aussi la notion de fréquence d'une lettre qui est son nombre d'occurrences rapporté à la longueur du texte. Un des avantages de la notion de fréquence est qu'il est possible de considérer une table de fréquence déjà construite comme celle de la langue française.

L'application de codage c peut être représentée par un arbre binaire où les arêtes gauches correspondent à 0, les arêtes droites à 1 et les feuilles aux éléments de Σ dont les étiquettes des chemins y menant depuis la racine de l'arbre correspondent à leur image par c.

Par exemple, pour le code $a = 0, b = 10, c = 11$ on aurait l'arbre :

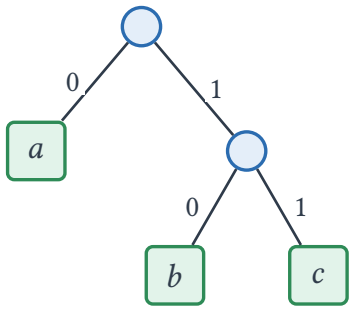


Figure 38.5 — Arbre du code préfixe $a = 0$, $b = 10$, $c = 11$.

Avec un tel arbre, il est très simple de décoder le texte codé car il suffit de suivre un chemin dans l'arbre jusqu'à tomber sur une feuille, produire la lettre correspondante, puis repartir de la racine de l'arbre. La longueur du code associé à une lettre est alors égale à la profondeur de la feuille correspondante. L'optimalité du codage préfixe est ainsi équivalente à la minimalité de l'arbre vis-à-vis de la fonction d'objectif $\varphi(t) = \sum_{x\in\Sigma} f(x)p(t,x)$ où $p(t,x)$ est la profondeur de la feuille d'étiquette x dans l'arbre t ou 0 si x n'est pas une des étiquettes, cette extension permettant d'étendre la fonction d'objectif aux solutions partielles.

38.2.2 Algorithme glouton et implémentation

L'algorithme d'Huffman va construire un arbre correspondant à un codage optimal à l'aide d'une file de priorité d'arbres. On étend pour cela l'application f à de tels arbres en définissant que si t est un arbre de feuilles $x_1, ..., x_n$ alors $f(t) = f(x_1) + ... + f(x_n)$.

- Au départ, on place dans la file des arbres réduits à une feuille pour chaque élément $x \in \Sigma$ et dont la priorité est $f(x)$.
- Tant que la file contient au moins deux éléments
  - on retire les deux plus petits éléments x et y de la file, de priorités $f(x)$ et $f(y)$
  - on ajoute un arbre $z = Noeud(x, y)$ de priorité $f(z) = f(x) + f(y)$.
- On renvoie l'unique élément restant dans la file.

L'implémentation de cet algorithme est alors assez directe avec une file de priorité.

- En OCaml, on réutilise le module `Tas` du TP Huffman (fonctions `Tas.cree`, `Tas.ajoute`, `Tas.extrait_maximum` et `Tas.taille`). Comme il s'agit d'un tas max sur des couples comparés d'abord selon leur première

composante, on insère le couple $(-f(x), x)$: extraire le maximum revient à extraire l'arbre de plus petite valeur pour f.
- En C, on écrit directement un tas min de couples (fréquence, arbre).
- En Python, le module `heapq` fournit un tas min sur une liste. On intercale dans les triplets un numéro unique pour départager les égalités de fréquence : sans lui, `heapq` devrait comparer deux arbres, ce qui échoue entre un entier et un couple.

Dans les trois cas, le tableau `occ` de taille 256 donne le nombre d'occurrences de chaque octet et une feuille est étiquetée par un octet. La file contient au plus 256 arbres à tout instant.

OCaml

```
type arbre = Feuille of int | Noeud of arbre * arbre

let construit_arbre occ =
    let arbres = Tas.cree 256 (0, Feuille 0) in
    for i = 0 to 255 do
        let f = occ.(i) in
        if f > 0 (* on ignore les occurrences nulles *)
        then Tas.ajoute (-f, Feuille i) arbres
    done;
    if Tas.taille arbres = 0
    then invalid_arg "construit_arbre : texte vide";
    while Tas.taille arbres > 1 do
        let fx, x = Tas.extrait_maximum arbres in
        let fy, y = Tas.extrait_maximum arbres in
        Tas.ajoute (fx+fy, Noeud(x,y)) arbres
    done;
    snd (Tas.extrait_maximum arbres)
```

C

```
#include <assert.h>
#include <stdbool.h>
    #include <stdlib.h>

/* une feuille est un noeud sans enfant */
typedef struct arbre {
    int caractere; /* n'a de sens que pour une feuille */
    struct arbre *gauche;
    struct arbre *droite;
} arbre;

arbre *feuille(int c)
{
    arbre *a = malloc(sizeof(arbre));
    a->caractere = c;
    a->gauche = NULL;
    a->droite = NULL;
    return a;
}

arbre *noeud(arbre *g, arbre *d)
{
    arbre *a = malloc(sizeof(arbre));
    a->caractere = -1;
    a->gauche = g;
    a->droite = d;
    return a;
}

void libere_arbre(arbre *a)
{
    if (a == NULL)
        return;
    libere_arbre(a->gauche);
    libere_arbre(a->droite);
```

```
    free(a);
}

/* tas min de couples (frequence, arbre), au plus 256 elements */
typedef struct {
    unsigned int frequence;
    arbre *arbre;
} element;

typedef struct {
    element elements[256];
    int taille;
} tas;

void echange(element *v, int x, int y)
{
    element t = v[x];
    v[x] = v[y];
    v[y] = t;
}

void ajoute(tas *t, element e)
{
    assert(t->taille < 256);
    int pos = t->taille;
    t->elements[pos] = e;
    t->taille++;
    while (pos > 0 && t->elements[(pos - 1) / 2].frequence
                        > t->elements[pos].frequence)
    {
        echange(t->elements, (pos - 1) / 2, pos);
        pos = (pos - 1) / 2;
    }
}

element extrait_minimum(tas *t)
{
    assert(t->taille > 0);
    element x = t->elements[0];
    t->taille--;
    t->elements[0] = t->elements[t->taille];
    int pos = 0;
    while (true)
    {
        int min = pos;
        int g = 2 * pos + 1;
        int d = 2 * pos + 2;
        if (g < t->taille
            && t->elements[g].frequence < t->elements[min].frequence)
            min = g;
        if (d < t->taille
            && t->elements[d].frequence < t->elements[min].frequence)
            min = d;
        if (min == pos)
            break;
        echange(t->elements, pos, min);
        pos = min;
    }
    return x;
}

/* renvoie NULL pour un texte vide */
arbre *construit_arbre(const unsigned int occ[256])
{
    tas arbres = { .taille = 0 };
    for (int i = 0; i < 256; i++)
    {
        if (occ[i] > 0) /* on ignore les occurrences nulles */
            ajoute(&arbres, (element) { occ[i], feuille(i) });
```

```
    }
    if (arbres.taille == 0)
        return NULL;
    while (arbres.taille > 1)
    {
        element x = extrait_minimum(&arbres);
        element y = extrait_minimum(&arbres);
        ajoute(&arbres, (element) { x.frequence + y.frequence,
                                    noeud(x.arbre, y.arbre) });
    }
    return extrait_minimum(&arbres).arbre;
}
```

Python

```
import heapq


def construit_arbre(occ):
    arbres = []
    for i in range(256):
        if occ[i] > 0:  # on ignore les occurrences nulles
            heapq.heappush(arbres, (occ[i], i, i))
    if len(arbres) == 0:
        raise ValueError("construit_arbre : texte vide")
    numero = 256
    while len(arbres) > 1:
        fx, _, x = heapq.heappop(arbres)
        fy, _, y = heapq.heappop(arbres)
        heapq.heappush(arbres, (fx + fy, numero, (x, y)))
        numero += 1
    return arbres[0][2]
```

Remarque 38.9 Deux cas limites sont à traiter à part :

- si le texte est vide, $\Sigma = \emptyset$ et il n'y a aucun arbre à renvoyer : on lève une exception (on renvoie `NULL` en C) et c'est à l'appelant de traiter ce cas ;
- si le texte ne contient qu'une seule lettre c, la boucle n'est jamais exécutée et on renvoie la feuille c seule. C'est bien l'arbre optimal au sens de φ (de valeur 0) mais le code associé à c est le **mot vide** : le texte codé est vide et on ne peut retrouver le texte qu'en connaissant sa longueur. C'est pourquoi, en pratique, on mémorise aussi la longueur du texte dans le fichier compressé.

L'algorithme de Huffman est un algorithme glouton car si on considère pour solution partielle la forêt présente dans la file et pour objectif la fonction φ étendue aux forêts en sommant la valeur de φ sur chaque arbre, alors fusionner dans la forêt F deux arbres x et y en la transformant en une forêt F' va avoir l'impact suivant sur la fonction d'objectif :

$$\varphi(F') = \varphi(F) + f(x) + f(y)$$

car, en effet, on va rajouter 1 à la profondeur de chaque feuille de x et de y. En notant Σ_x l'ensemble des étiquettes des feuilles de x, on passe pour la contribution de x de $\varphi(x) = \sum_{c\in\Sigma_x} f(c)p(x,c)$ à $\sum_{c\in\Sigma_x} f(c)(p(x,c)+1) = \varphi(x) + \sum_{c\in\Sigma_x} f(c) = \varphi(x) + f(x)$, et de même pour y.

On remarque ainsi que la fusion qui minimise localement φ est celle qui fusionne les deux arbres de plus petite valeur pour f.

38.2.3 Preuve d'optimalité

Pour montrer que l'algorithme glouton produit ici un codage minimal, on va utiliser une technique classique qui consiste à montrer qu'étant donné une solution optimale, on peut toujours la transformer sans augmenter sa valeur pour obtenir, de proche en proche, la solution renvoyée par le glouton.

Théorème 38.10

On suppose $|\Sigma| \geq 2$. Soient a et b deux lettres distinctes les moins fréquentes, c'est-à-dire telles que $f(a) \leq f(b) \leq f(x)$ pour toute lettre $x \notin \{a, b\}$. Il existe un arbre optimal dont les deux feuilles étiquetées par a et b descendent du même nœud et sont de profondeur maximale.

Preuve Considérons un arbre optimal t et soit c l'étiquette d'une feuille de profondeur maximale h. Comme $|\Sigma| \geq 2$, cette feuille n'est pas la racine. Son père a forcément deux enfants car sinon, on pourrait omettre ce nœud et l'arbre obtenu serait de valeur strictement plus petite par φ (car $f(c) > 0$), ce qui contredit l'optimalité de t. Le frère de cette feuille est lui-même une feuille, car sinon il aurait des feuilles descendantes de profondeur strictement plus grande que h. Soit d l'étiquette de cette feuille sœur.

On commence par remarquer que si x et y sont deux lettres telles que $f(x) \leq f(y)$ et que y étiquette une feuille de profondeur maximale, donc $p(t, x) \leq p(t, y)$, alors échanger les étiquettes x et y préserve le caractère optimal. En effet, seuls les termes associés à ces lettres changent dans l'évaluation de φ. Si on note t' le nouvel arbre obtenu après cet échange, on a

$$\varphi(t') = \varphi(t) - f(x)p(t, x) - f(y)p(t, y) + f(x)p(t, y) + f(y)p(t, x)$$

Or, $f(y) \geq f(x)$ et $p(t, x) \leq p(t, y)$ donc

$$\varphi(t') = \varphi(t) + (f(y) - f(x))(p(t, x) - p(t, y)) \leq \varphi(t)$$

L'échange préserve le caractère optimal. En fait, ici, on a nécessairement une égalité pour ne pas aboutir à une contradiction, donc soit les feuilles étaient à même profondeur, soit les lettres avaient le même nombre d'occurrences.

On applique ce résultat une première fois avec $x = a$ et $y = c$ (si $c \neq a$) : comme a a le plus petit nombre d'occurrences, $f(a) \leq f(c)$. On obtient un arbre optimal où a est à la place de c. Soit alors e l'étiquette de la feuille sœur dans ce nouvel arbre (là où était d) : $e \neq a$ donc $f(b) \leq f(e)$ et on peut appliquer à nouveau le résultat avec $x = b$ et $y = e$ (si $e \neq b$). L'arbre obtenu est optimal et a et b y étiquettent deux feuilles sœurs de profondeur maximale.

□

Le théorème suivant permet de raisonner par récurrence en diminuant le nombre de lettres.

Théorème 38.11

Soit t un arbre ayant x et y comme feuilles sœurs et t' l'arbre obtenu en remplaçant le nœud liant x et y par une feuille étiquetée par z où z est une nouvelle lettre telle que $f(z) = f(x) + f(y)$.

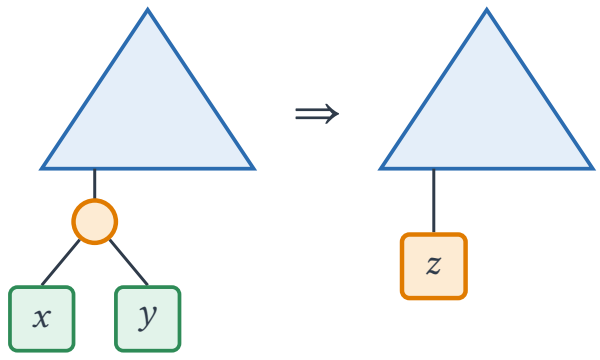


Figure 38.6 — Fusion des feuilles sœurs x et y en une feuille z, avec $f(z) = f(x) + f(y)$

```
On a alors $\varphi(t) = \varphi(t') + f(z)$.
```

Preuve Seuls les termes portant sur x, y et z sont influencés par le changement et, comme $p(t, x) = p(t, y) = p(t', z) + 1$, on a :

$$\begin{aligned}\varphi(t) &= \varphi(t') + f(x)p(t,x) + f(y)p(t,y) - f(z)p(t',z)\\ &= \varphi(t') + f(z)(p(t',z)+1) - f(z)p(t',z)\\ &= \varphi(t') + f(z)\end{aligned}$$

□

Théorème 38.12

L'algorithme de Huffman renvoie un arbre optimal.

Preuve Par récurrence sur $|\Sigma|$.

On montre plus précisément que l'arbre renvoyé est optimal quels que soient les choix effectués par la file de priorité en cas d'égalité des priorités.

Initialisation : si Σ ne contient qu'une lettre, il n'y a qu'un arbre qui est nécessairement optimal.

Hérédité : si la propriété est vraie pour tout alphabet de $n-1 \geq 1$ lettres, alors soit Σ contenant n lettres et x et y les deux lettres retirées de la file lors de la première étape : ce sont deux lettres les moins fréquentes au sens du premier théorème.

On pose Σ' obtenu en remplaçant x et y par une nouvelle lettre z et on pose $f(z) = f(x) + f(y)$. Après la première étape, la file contient les mêmes priorités que la file initiale pour Σ', l'arbre $Noeud(x,y)$ jouant le rôle de la feuille z. La suite de l'exécution est donc une exécution de l'algorithme de Huffman sur Σ', qui renvoie un arbre t', optimal d'après l'hypothèse de récurrence, et l'arbre t obtenu en partant de Σ se déduit de t' en remplaçant z par $Noeud(x,y)$. Le théorème précédent assure alors que $\varphi(t) = \varphi(t') + f(z)$.

Soit t_o un arbre optimal pour Σ dans lequel x et y sont sœurs, possible en vertu du premier théorème, et soit t'_o l'arbre obtenu en remplaçant dans t_o le nœud liant x et y par une feuille étiquetée par z. C'est un arbre pour Σ' et on a ici encore $\varphi(t_o) = \varphi(t'_o) + f(z) \geq \varphi(t') + f(z) = \varphi(t)$ car t' est optimal.

Ainsi, on a bien l'égalité $\varphi(t_o) = \varphi(t)$ et t est optimal.

□

38.3 Sélection d'activités

38.3.1 Description

Étant donné un ensemble d'activités données par leur temps de début et leur temps de fin (on considère les temps comme des entiers pour simplifier), on se pose la question du nombre maximal d'activités que l'on puisse sélectionner sans que deux activités soient en conflits. Cela correspond par exemple à l'organisation du planning d'un employé.

On suppose que chaque activité (d,f) vérifie $d < f$. On dit que deux activités (d_1,f_1) et (d_2,f_2) sont en conflit quand $d_1, f_1[\cap [d_2, f_2[\neq \emptyset$.

![t_3 (en rouge) est en conflit avec t_1 et t_2, qui sont compatibles.{width=50%}

Ici, t_1 et t_2 sont en conflits avec t_3. Mais t_1 et t_2 ne sont pas en conflit. On considère que deux activités peuvent se succéder directement : $f_1 = d_2$.

On considère donc en entrée de ce problème une suite finie $((d_1,f_1),\dots,(d_n,f_n))$ et on cherche un sous-ensemble $I \subset [\![1,n]\!]$ de plus grand cardinal tel que pour tous $i,j \in I$, si $i \neq j$ alors (d_i,f_i) et (d_j,f_j) ne sont pas en conflit. On dit que I est un **ensemble indépendant**.

38.3.2 Algorithme glouton et implémentation

Pour résoudre ce problème, on considère l'algorithme glouton associé à la fonction d'objectif cardinal et **en triant les activités** par ordre croissant de temps de fin.

Cet algorithme est implémenté dans le programme suivant :

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct {
    unsigned int id;
    unsigned int debut;
    unsigned int fin;
    unsigned char selectionnee;
} activite;

int compare_activites(const void *t1, const void *t2)
{
    unsigned int f1 = ((const activite *)t1)->fin;
    unsigned int f2 = ((const activite *)t2)->fin;
    /* pas de f1 - f2 : la différence de deux unsigned ne peut être négative */
    return (f1 > f2) - (f1 < f2);
}

void selectionne(activite *activites, size_t nb_activites)
{
    size_t derniere_activite = 0;
    if (nb_activites == 0)
        return;
    /* on commence par trier en O(n log2 n) les activites
     * selon le temps de fin */
    qsort(activites, nb_activites,
        sizeof(activite), compare_activites);
    activites[0].selectionnee = 1;
    for (size_t i = 1; i < nb_activites; i++)
    {
        if (activites[i].debut >= activites[derniere_activite].fin)
        {
            activites[i].selectionnee = 1;
            derniere_activite = i;
        }
    }
}

int main()
{
    activite activites[] = {
        { 0, 1, 3, 0 }, { 1, 3, 4, 0 }, { 2, 2, 5, 0 },
        { 3, 5, 9, 0 }, { 4, 11, 12, 0 }, { 5, 8, 10, 0 },
        { 6, 0, 7, 0 }
    };

    size_t nb_activites = sizeof(activites) / sizeof(activite);

    selectionne(activites, nb_activites);

    for (size_t i = 0; i < nb_activites; i++)
    {
        printf("Activité %u (%u,%u) : %d\n",
                activites[i].id, activites[i].debut,
                activites[i].fin, activites[i].selectionnee);
    }

    return 0;
}
```

Ce programme produit alors la sortie :

```
Activité 0 (1,3) : 1
Activité 1 (3,4) : 1
Activité 2 (2,5) : 0
Activité 6 (0,7) : 0
Activité 3 (5,9) : 1
```

```
Activité 5 (8,10) : 0
Activité 4 (11,12) : 1
```

Remarque 38.13 Comme l'algorithme commence par effectuer un tri, on a rajouté dans la structure `activite` un champ permettant d'identifier une activité autrement que par son indice.

38.3.3 Preuve d'optimalité

On va prouver que l'algorithme glouton renvoie un ensemble indépendant optimal. Le fait que l'ensemble soit indépendant étant direct, on se concentre sur la preuve d'optimalité en présentant un schéma de preuve qui correspond à celui identifié dans le paragraphe précédent.

Théorème 38.14

Si $a_1, ..., a_n$ (avec $n \geq 1$) sont des activités énumérées dans l'ordre croissant de leur temps de fin, alors il existe un ensemble indépendant optimal contenant a_1.

Remarque 38.15 Cela signifie qu'il fait le même choix que l'algorithme glouton à la première étape.

Preuve Soit I un ensemble indépendant optimal ne contenant pas $a_1 = (d_1, f_1)$ (sinon c'est direct). Comme $\{a_1\}$ est indépendant, I est non vide. Si $a_k = (d_k, f_k)$ est l'activité de plus petit indice dans I, alors $f_k \geq f_1$. Soit $a_i = (d_i, f_i)$ dans $I' = I \setminus \{a_k\}$: on a $f_i \geq f_k$ et a_i et a_k ne sont pas en conflit, donc $d_i \geq f_k$ (car $f_i \leq d_k$ est impossible puisque $d_k < f_k \leq f_i$). Ainsi on a $d_i \geq f_k \geq f_1$ et a_1 et a_i ne sont pas en conflit. Ainsi $I' \cup \{a_1\}$ est un ensemble indépendant contenant a_1 de même cardinal que I donc optimal.

□

Théorème 38.16

Soit $A = \{a_1, ..., a_n\}$ des activités ordonnées par ordre croissant de temps de fin et I un ensemble indépendant optimal contenant $a_1 = (d_1, f_1)$ (ce qui est possible selon le théorème précédent).

$I' = I \setminus \{a_1\}$ est optimal pour $A' = \{(d, f) \in A \mid d \geq f_1\}$.

Preuve Tout d'abord, $I' \subset A'$: si $a_i = (d_i, f_i) \in I'$, a_i n'est pas en conflit avec a_1 et $f_i \geq f_1 > d_1$, donc $d_i \geq f_1$.

Si, par l'absurde, I' n'est pas optimal pour A', alors il existe $J \subset A'$ un ensemble indépendant de cardinal strictement plus grand que celui de I'. Or, toute activité de J commence après la fin de a_1 (et $a_1 \notin A'$ car $d_1 < f_1$), donc $J \cup \{a_1\}$ est indépendant pour l'ensemble des activités et est de cardinal strictement plus grand que I. Contradiction.

□

Théorème 38.17

L'algorithme glouton renvoie un ensemble indépendant optimal.

Preuve Par récurrence forte sur le nombre n d'activités.

- Initialisation : Pour $n = 0$, le glouton renvoie $\emptyset$ qui est directement optimal.
- Hérédité : Soit $n \geq 1$. Si la propriété est vérifiée pour $k < n$ activités, soit $A = \{a_1, ..., a_n\}$ des activités ordonnées par temps de fin. Soit I un ensemble indépendant optimal contenant a_1 et $I' = I \setminus \{a_1\}$. Le théorème précédent assure que I' est optimal sur $A' = \{(d, f) \in A \mid d \geq f_1\}$, qui contient au plus $n - 1$ activités car $a_1 \notin A'$.

  Par hypothèse de récurrence, l'algorithme glouton sur A' produit un ensemble indépendant optimal G', donc tel que $|G'| = |I'|$. Or l'algorithme glouton sur A sélectionne a_1, puis ne sélectionne jamais une activité commençant avant f_1 (la dernière activité sélectionnée finit toujours après f_1) et effectue sur les

activités de A', parcourues dans le même ordre, exactement les mêmes choix que le glouton sur A'. Ainsi l'algorithme glouton sur A renvoie $G = G' \cup \{a_1\}$ de même cardinal que I, donc optimal.

□

38.4 Principe général des preuves d'optimalité

L'idée de la preuve précédente et de celles qui vont suivre est d'identifier dans la solution gloutonne ce qui correspond au choix glouton initial. On va reprendre de manière générale le principe de cette preuve ici dans le cas d'un problème de minimisation.

Si on note g le glouton pour l'entrée E et c ce choix initial, on peut déduire de E la sous-entrée E' correspondant à avoir enlevé c, ce qui peut conduire à changer E ou à supprimer d'autres éléments conséquences du choix c. Comme le glouton est une itération de choix gloutons, on déduit g' de g en omettant ce choix c et g' est le résultat du glouton pour E'.

Pour Huffman, E' se déduit en rajoutant une nouvelle lettre, ce n'est donc pas juste E sans les deux lettres les moins fréquentes.

On va démontrer un premier résultat qui est un résultat d'échange : si le glouton fait le choix c c'est que c est localement minimal, on va alors montrer qu'il existe un optimal o' faisant le choix c en montrant que si un optimal o a fait un autre choix c' alors on peut préserver l'optimalité en remplaçant c' par c.

Dans le cas de Huffman, il s'agissait de placer les deux lettres les moins fréquentes en tant que sœurs d'un nœud profond.

Ensuite, on montre un résultat de réduction en enlevant c qui va relier $\chi(g)$ à $\chi(g')$ et, de même, $\chi(o')$ à $\chi(o'')$, où o'' est la solution pour E' déduite de o' en effectuant la même omission de c. On s'assure que cette relation, qui doit être croissante en $\chi(g')$, permette alors de faire le raisonnement suivant :

$$\chi(g) = \text{expr. de } \chi(g') \text{ et } c$$

Or, $\chi(g') = Opt(E') \leq \chi(o'')$ (par récurrence, le glouton est optimal sur E') donc

$$\chi(g) \leq \text{expr. de } \chi(o'') \text{ et } c = \chi(o') = Opt(E)$$

Et par minimalité de $Opt(E)$, on a donc $\chi(g) = Opt(E)$.

Ce principe est décrit sur le schéma suivant :

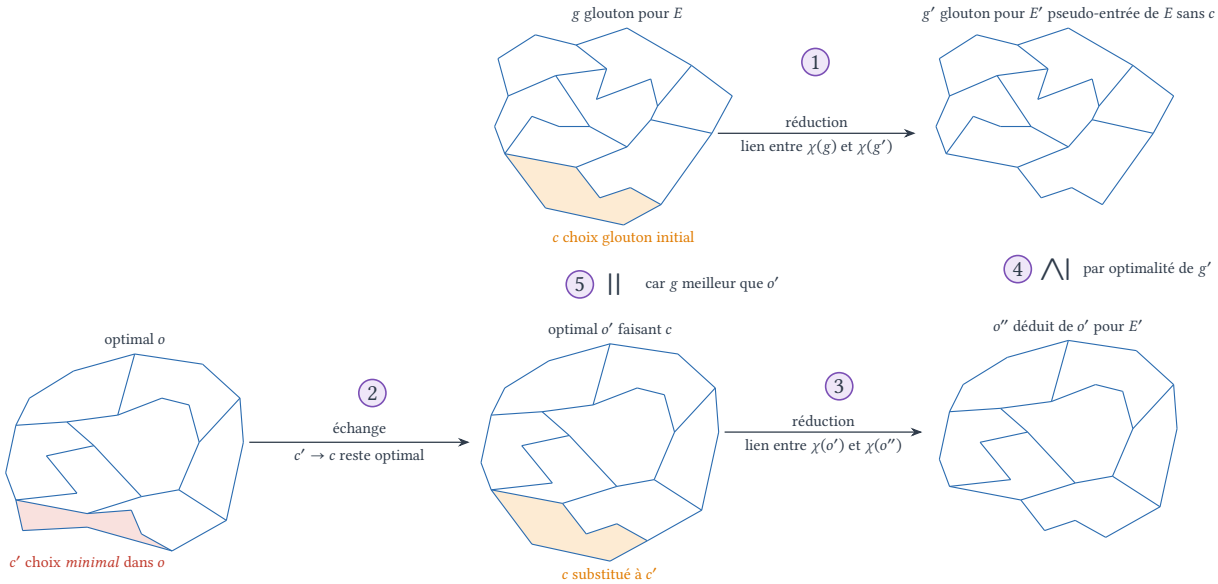


Figure 38.7 — Principe des preuves d'optimalité d'un glouton : échange, réductions et comparaisons.

38.5 Ordonnancement de tâches

38.5.1 Description

On considère ici un problème voisin du problème précédent. On considère n tâches $T = \{t_1, ..., t_n\}$ prenant une unité de temps pour être traitées sur une unique unité de calcul.

Chaque tâche t dispose d'une date limite $f(t) \in [\![1, n]\!]$ (**deadline**) à laquelle elle doit être traitée sans quoi on écope d'une pénalité $p(t) \in \mathbb{N}$.

On appelle stratégie d'ordonnancement une application **bijective** $d : T \to [\![0, n-1]\!]$ qui associe à chaque tâche un unique temps de début $d(t)$, deux tâches ne pouvant pas être traitées en même temps. Selon cette stratégie, on déduit une séparation de T en deux ensembles disjoints :

- $T^+(d)$ l'ensemble des tâches traitées dans les délais : $t \in T^+(d) \Leftrightarrow d(t) < f(t)$.
- $T^-(d)$ l'ensemble des tâches traitées en retard : $t \in T^-(d) \Leftrightarrow d(t) \geq f(t)$.

On note alors $P(d) = \sum_{t \in T^-(d)} p(t)$ la somme des pénalités des tâches en retard.

Exemple 38.18

On considère l'ensemble de tâches :

t_i	1	2	3	4	5	6	7
$f(t_i)$	1	2	3	4	4	4	6
$p(t_i)$	3	6	4	2	5	7	1

Une stratégie d'ordonnancement (les tâches en retard sont en gras) est donnée dans le tableau suivant :

t_i	1	2	3	4	5	6	7
$d(t_i)$	**6**	0	1	**4**	3	2	5

On a alors $P(d) = 5$.

On cherche à obtenir une stratégie d'ordonnancement de valeur $P(d)$ minimale.

On remarque que l'ordonnancement des tâches en retard n'a aucune importance, et on peut donc se contenter de déterminer une stratégie d'ordonnancement pour les tâches traitées dans les délais et la compléter par n'importe quel ordonnancement des autres tâches. On peut ainsi reformuler le problème : déterminer un sous-ensemble $T^+ \subset T$ de tâches **pouvant** être traitées dans les délais tel que $p(T^+) = \sum_{t \in T^+} p(t)$ soit **maximale** (on a alors $P(d) = p(T) - p(T^+)$).

38.5.2 Algorithme glouton et implémentation

On résout maintenant ce problème de maximisation de la somme des pénalités de T^+ par un algorithme glouton :

- On commence avec $T^+ = \emptyset$ et tous les temps de $[\![0, n-1]\!]$ sont marqués comme étant disponibles.
- On parcourt les tâches dans l'ordre décroissant des pénalités.
  - Quand on considère la tâche t s'il existe un temps i disponible tel que $i < f(t)$ alors on marque comme indisponible le temps $i_0 = \max\{i \in [\![0, n-1]\!] \mid i < f(t) \text{ et } i \text{ disponible}\}$ et on rajoute alors t à T^+ en commençant t au temps i_0.
- On place les tâches restantes aux temps disponibles.

Pour les structures de données, on utilise une représentation en tableaux de booléens (des `unsigned char` à `0` ou `1` en C) pour la disponibilités des temps. L'ensemble T^+ est alors implicite car il correspond aux tâches ordonnancées dans la première étape. Utiliser un tableau implique qu'une recherche linéaire soit faite pour chercher un plus grand temps disponible, et donc, la complexité temporelle globale sera en $O(n^2)$.

Remarque 38.19 Il est possible d'améliorer cela pour passer en $O(n \log_2 n)$ (*exercice*).

Le programme C suivant implémente cet algorithme.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct {
    unsigned int id;
    unsigned int date_limite;
    unsigned int penalite;
    int debut; /* -1 tant que la tâche n'est pas ordonnancée */
} tache;

int compare_taches(const void *t1, const void *t2)
{
    unsigned int p1 = ((const tache *)t1)->penalite;
    unsigned int p2 = ((const tache *)t2)->penalite;
    /* ordre décroissant des pénalités */
    return (p1 < p2) - (p1 > p2);
}


void ordonnancement(tache *taches, size_t nb_taches)
{
    unsigned char *temps_occupe = malloc(sizeof(unsigned char) * nb_taches);
    memset(temps_occupe, 0, nb_taches);

    /* tri des activités par ordre décroissant des pénalités */
    qsort(taches, nb_taches, sizeof(tache), compare_taches);

    /* T+ par algorithme glouton */
    for (size_t k = 0; k < nb_taches; k++)
    {
        int i0 = -1;
        for (size_t i = 0; i < nb_taches; i++)
        {
            if (temps_occupe[i] == 0 && i < taches[k].date_limite)
                i0 = i;
        }
        if (i0 >= 0)
        {
            taches[k].debut = i0;
            temps_occupe[i0] = 1;
        }
    }

    /* Complétion par les tâches en retard */
    int i = 0; // indice du dernier temps disponible utilisé

    for (size_t k = 0; k < nb_taches; k++)
    {
        if (taches[k].debut == -1)
        {
            while(temps_occupe[i] == 1)
                i++;
            taches[k].debut = i;
            temps_occupe[i] = 1;
        }
    }

    free(temps_occupe);
}

int main()
{
    tache taches[] = {
```

```
        { 1, 1, 3, -1 }, { 2, 2, 6, -1 }, { 3, 3, 4, -1 },
        { 4, 4, 2, -1 }, { 5, 4, 5, -1 }, { 6, 4, 7, -1 },
        { 7, 6, 1, -1 }
    };

    size_t nb_taches = sizeof(taches) / sizeof(tache);

    ordonnancement(taches, nb_taches);

    for (size_t i = 0; i < nb_taches; i++)
    {
        printf("T%u (f:%u,p:%u) @ %d\n", taches[i].id, taches[i].date_limite,
                taches[i].penalite, taches[i].debut);
    }

    return 0;
}
```

Il produit la sortie :

```
T6 (f:4,p:7) @ 3
T2 (f:2,p:6) @ 1
T5 (f:4,p:5) @ 2
T3 (f:3,p:4) @ 0
T1 (f:1,p:3) @ 4
T4 (f:4,p:2) @ 6
T7 (f:6,p:1) @ 5
```

Ce qui correspond à l'ordonnancement $t_3, t_2, t_5, t_6, t_1, t_7, t_4$. Les tâches t_1 et t_4 sont en retard, donc la pénalité totale est de 5.

38.5.3 Preuve d'optimalité

On va montrer que cet algorithme glouton renvoie un ensemble T^+ optimal. Pour cela, on procède comme précédemment. Tout d'abord, on montre qu'il existe une solution optimale qui effectue le premier choix de l'algorithme glouton.

Théorème 38.20

Soit T un ensemble de tâches et $t \in T$ une tâche de pénalité maximale. Il existe un ensemble T^+ de tâches pouvant être traitées dans les délais, maximal pour les pénalités et tel que $t \in T^+$.

Preuve Soit $T^+ \subset T$ un ensemble maximal, dont on fixe un ordonnancement où toutes ses tâches sont dans les délais. S'il contient t, il convient directement. Sinon, deux cas se présentent :

- si l'un des instants de $[\![0, f(t) - 1]\!]$ n'est occupé par aucune tâche de T^+, alors $T^+ \cup \{t\}$ peut être traité dans les délais en plaçant t à cet instant, et $p(T^+ \cup \{t\}) \geq p(T^+)$: cet ensemble est donc maximal et contient t ;
- sinon, il existe une tâche t' de T^+ qui est traitée à un moment où on pourrait traiter t à temps (par exemple à l'instant 0). On a $p(t') \leq p(t)$ par maximalité de $p(t)$. L'ensemble T' déduit de T^+ en remplaçant t' par t, traitée au même instant, peut être traité dans les délais par construction et $p(T') = p(T^+) - p(t') + p(t) \geq p(T^+)$, donc on a forcément $p(T') = p(T^+)$ par optimalité de T^+ : T' convient.

□

On montre maintenant qu'en enlevant le choix glouton, on obtient une solution optimale du sous-problème.

Théorème 38.21

Soit $T^+ \subset T$ ensemble de tâches pouvant être traitées dans les délais, maximal pour les pénalités et contenant une tâche t de plus grande pénalité. Soit i l'instant auquel la tâche t commence dans un ordonnancement de T^+ où toutes ses tâches sont dans les délais.

On pose $T' = T \setminus \{t\}$ avec des dates limites modifiées :

$$\forall t' \in T', f_{T'}(t') = \begin{cases} f_T(t') & \text{si } f_T(t') \leq i \\ f_T(t') - 1 & \text{sinon} \end{cases}$$

$T^+ \setminus \{t\}$ est alors maximal pour T'.

Preuve Dans T', on a à la fois enlevé t et supprimé l'instant i : les instants $j > i$ de T deviennent les instants $j-1$ de T', et les dates limites sont modifiées en conséquence. Tout ordonnancement dans les délais d'une partie U de T' peut alors être *relevé* en un ordonnancement dans les délais de $U \cup \{t\}$ dans T en décalant d'un instant les tâches commençant à partir de l'instant i et en ordonnançant là la tâche t (qui est dans les délais car $i < f(t)$). Réciproquement, en retirant t de l'ordonnancement de T^+ et en avançant d'un instant les tâches commençant après i, on obtient que $T^+ \setminus \{t\}$ peut être traité dans les délais pour T'.

Ainsi, s'il existait T'^+ pouvant être traité dans les délais pour T' tel que $p(T'^+) > p(T^+ \setminus \{t\}) = p(T^+) - p(t)$ alors $T'^+ \cup \{t\}$ pourrait être traité dans les délais pour T et serait de somme de pénalités strictement plus grande que celle de T^+ supposé maximal.

Donc, $T^+ \setminus \{t\}$ est maximal.

□

On conclut alors directement par récurrence sur le nombre de tâches comme on l'a fait précédemment pour la sélection d'activités :

Théorème 38.22

L'algorithme glouton renvoie un ordonnancement optimal.

Preuve Il suffit de montrer que l'ensemble G des tâches placées dans les délais par l'algorithme glouton est maximal. On procède par récurrence forte sur le nombre n de tâches, le cas $n = 0$ étant immédiat.

Soit $n \geq 1$ et t la première tâche considérée par le glouton : elle est de pénalité maximale et, tous les temps étant disponibles, elle est placée à l'instant $i = f(t) - 1$. D'après le premier théorème, il existe T^+ maximal contenant t. Dans un ordonnancement dans les délais de T^+, t commence à un instant $j \leq f(t) - 1$; en échangeant t avec la tâche éventuellement placée à l'instant $f(t) - 1$, qui est alors avancée et reste donc dans les délais, on peut supposer que t commence à l'instant i. Le théorème précédent assure alors que $T^+ \setminus \{t\}$ est maximal pour T' défini avec cet instant i.

Or, après avoir placé t, le glouton sur T place chaque tâche u au plus grand instant disponible strictement inférieur à $f_T(u)$, l'instant i étant occupé : après renumérotation des instants, c'est exactement le plus grand instant disponible strictement inférieur à $f_{T'}(u)$ dans T'. Le glouton sur T effectue donc ensuite les mêmes choix que le glouton sur T' (une date limite modifiée nulle signifiant simplement que la tâche est forcément en retard). Par hypothèse de récurrence, le glouton sur T' renvoie un ensemble G' maximal, donc $p(G') = p(T^+ \setminus \{t\})$, et $G = G' \cup \{t\}$ vérifie $p(G) = p(T^+)$: il est maximal.

□

38.6 Exercices

Beaucoup de ces exercices sont des adaptations d'exercices du livre Algorithms de Jeff Erikson.

Exercice

Un bibliothécaire souhaite ranger des collections de livres classées par auteur sur une longue étagère. On a ainsi une suite $(a_1, ..., a_n)$ de collections données par la taille de la collection sur l'étagère, par exemple, en nombre de pages.

Un rangement des livres consiste à ordonner les collections de la première à la dernière sur l'étagère. Plus formellement, il s'agit d'une permutation $\sigma \in \mathfrak{S}_n$ où $\sigma(i)$ donne le numéro de la collection placée en i-ème position sur l'étagère.

Pour accéder à un auteur, il est nécessaire de parcourir linéairement l'étagère en partant de la première collection. Le coût d'accès à la k-ième collection de l'étagère est donc $\text{cout}\,(k) = \sum_{i=1}^{k} a_{\sigma(i)}$.

Le coût moyen d'accès aux collections est alors

$$\frac{1}{n}\sum_{k=1}^{n} \text{cout}\,(k)$$

Déterminer un algorithme permettant d'obtenir un rangement de coût minimal et l'analyser (correction, complexité).

Exercice

Soit X un ensemble de n segments de $\mathbb{R}$. On dit que $Y \subset X$ couvre X si $\bigcup_{y\in Y} y = \bigcup_{x\in X} x$. Le cardinal de Y est appelé la taille de la couverture. On cherche ici une couverture de taille minimale.

a) Déterminer un algorithme glouton pour obtenir une couverture *a priori* de petite taille.

b) Renvoie-t-il une couverture de taille minimale ?

Preuve

a) On note $I_1 = [a_1; b_1], ..., I_n = [a_n; b_n]$ les segments. Pour réaliser un algorithme glouton, on va considérer le plus petit élément x de $E = \bigcup I_i$. Comme ce sont des segments et qu'il y en a un nombre fini, ce plus petit élément existe toujours. Or, x doit être couvert et pour cela, on va considérer le plus grand des segments le contenant. Comme c'est le minimum, il est forcément de la forme $I = [x; y]$. On pose alors $I_i' = \overline{I_i \setminus I}$ où

$$\overline{\varnothing} = \varnothing \quad \overline{]a;b]} = \overline{[a;b[} = \overline{[a;b]} = [a;b]$$

Comme x ne peut être intérieur à un des I_i, la différence $I_i \setminus I$ est nécessairement un intervalle, donc d'une de ces quatre formes (le cas $]a;b[$ étant exclu car on retranche un segment à un segment). On a $E = I \cup \bigcup I_i'$ et on déduit une couverture de $\bigcup I_i' = I_{i_1}' \cup ... \cup I_{i_p}'$ par l'algorithme glouton (on omet alors les ensembles vides qui ne sont pas des segments mais ne contribuent pas à la couverture ou à E). Comme $I_i' \subset I_i$ on en déduit une couverture pour E. b) Comme pour les autres preuves d'optimalité, on va la décomposer en deux temps.

- Tout d'abord, il existe une couverture optimale réalisant le choix glouton, c'est direct car une couverture optimale doit couvrir le minimum x et elle contient ainsi un segment de la forme $I' = [x; y'] \subset [x; y] = I$, où I est le plus grand segment contenant x. Donc on peut substituer I à I' tout en préservant la minimalité de la couverture.
- Ensuite, on raisonne par récurrence forte sur le nombre de segments pour montrer que le glouton est optimal. S'il n'y a aucun segment, c'est direct. Si la propriété est vraie pour $0 \leq p < n$ segments et qu'on considère n segments $I_1, ..., I_n$, alors on peut supposer que le choix glouton est I_1 quitte à réordonner les segments. D'après le point précédent, il existe donc un optimal $I_1 \cup I_{i_1} \cup ... \cup I_{i_r} = E$ et le glouton donne lui $I_1 \cup I_{j_1} \cup ... \cup I_{j_s} = E$ avec $1 + s \geq 1 + r$. On pose $I_k' = \overline{I_k \setminus I_1}$ et $E' = \bigcup_{k\geq 2} I_k'$. Par construction, $I_{j_1}', ..., I_{j_s}'$ est la couverture de E' renvoyée par le glouton sur les segments non vides parmi les I_k', $k \geq 2$, qui sont au plus $n - 1$. De plus, $I_{i_1}', ..., I_{i_r}'$ est aussi une couverture de E' : un point $z \in I_k \setminus I_1$ est dans un des I_{i_m}, donc dans I_{i_m}' ; et si $z = y$ est le point ajouté par l'adhérence, les points de I_k immédiatement à droite de y sont couverts par un nombre fini de segments I_{i_m}, donc l'un d'entre eux contient des points arbitrairement proches de y hors de I_1 et $y \in I_{i_m}'$. Or, la couverture $I_{j_1}', ..., I_{j_s}'$ est optimale par hypothèse de récurrence, donc $s \leq r$. On en déduit directement que $s = r$.

Note La preuve peut être simplifiée tout en préservant son intérêt en ne considérant que des *segments* d'entiers $[\![a, b]\!]$. Ici, le problème vient du fait que si on a les segments $[1;3]$ et $[2;4]$, une fois $[1;3]$ choisi, il faut couvrir $[2;4] \setminus [1;3] =]3;4]$ ce qui oblige à considérer $[3;4]$.

□

Exercice

On considère le problème du rendu de monnaie présenté dans le premier paragraphe.

a) Montrer que, pour le système $(b^0, b^1, ..., b^k)$ où $b \in \mathbb{N}, b \geq 2$ et $k \in \mathbb{N}$, l'algorithme glouton est optimal.

b) Déterminer un algorithme renvoyant le nombre de pièces optimal pour un système monétaire quelconque. *Indice : ce n'est certainement pas l'algorithme glouton.*

Preuve

a) Commençons par montrer que si $x \in \mathbb{N}^*$ tel que b^i soit la plus grande pièce telle que $b^i \leq x$, alors il existe un rendu de monnaie optimal utilisant une pièce b^i. En effet, soit $(r_0, r_1, ..., r_k)$ un rendu optimal. S'il n'utilise pas la pièce b^i, alors il utilise uniquement les pièces b^0 à b^{i-1}. Comme $(b-1)\sum_{j=0}^{i-1} b^j = b^i - 1 < b^i \leq x$ il y a forcément une pièce b^j, avec $j < i$, telle que $r_j \geq b$. Comme $r_j b^j = (r_j - b)b^j + b^{j+1}$ et que b^{j+1} est une pièce du système car $j + 1 \leq i \leq k$, on peut réduire le rendu de $b - 1 \geq 1$ pièces ainsi. Comme il ne serait pas optimal, l'optimal utilise nécessairement la pièce b^i. On déduit alors par récurrence rapide que le glouton renvoie l'optimal. Ici, le fait qu'on soit forcé d'utiliser la pièce b^i montre en fait qu'il n'y a qu'un unique rendu optimal. Celui-ci est, en vérité, l'écriture de x en base b pour les nombres $< b^{k+1}$.

b) On remarque que $Opt(x)$, le nombre de pièces optimal pour x, vérifie $Opt(0) = 0$ et, pour $x > 0$, $Opt(x) = 1 + \min_{v_i \leq x} Opt(x - v_i)$. On en déduit un algorithme *a priori* naïf consistant à résoudre cette récurrence en identifiant la pièce v_i réalisant le minimum à chaque appel. On verra dans le chapitre sur la programmation dynamique qu'il est possible de rendre cet algorithme efficace en calculant ces minimums dans le bon ordre ou en se souvenant des résultats des appels.

□

Exercice

On considère un tableau $t = [a_0, ..., a_{n-1}]$ de n entiers relatifs. Un couple (i, j) est appelé un **segment positif pour** t si $0 \leq i \leq j \leq n-1$ et $\sum_{k=i}^{j} a_k \geq 0$.

On dit qu'une suite de segments positifs couvre t si chaque élément positif de t a son indice dans un des segments.

Si on considère le tableau :

```
t = [| 3; -5; 7; -4; 1; -8; 3; -7; 5; -9; 5; -2; 4 |]
```

Alors $(0, 4)$ (de somme $3 - 5 + 7 - 4 + 1 = 2$), $(6, 8)$ (de somme $3 - 7 + 5 = 1$) et $(10, 12)$ (de somme $5 - 2 + 4 = 7$) est une suite de 3 segments couvrant t.

Si t ne contient que des valeurs négatives, on considère qu'il est couvert par la suite vide (de 0 segment).

Déterminer un algorithme permettant d'obtenir une suite couvrant t de plus petit cardinal et analyser celui-ci.

Exercice

On considère le processus suivant : on part de l'entier 1 et à chaque étape on peut soit doubler l'entier courant, soit lui ajouter un. L'objectif est d'atteindre un entier cible n.

Par exemple, on peut atteindre 10 en quatre étapes ainsi :

$$1 \xrightarrow{+1} 2 \xrightarrow{\times 2} 4 \xrightarrow{+1} 5 \xrightarrow{\times 2} 10$$

Tout entier peut être atteint par une succession d'incréments, mais ce n'est pas le plus court en termes de nombre d'étapes.

Déterminer un algorithme permettant d'obtenir le nombre d'étapes minimal pour atteindre un entier n et l'analyser.

Preuve On va considérer la stratégie suivante pour obtenir 1 en un minimum d'étapes en partant de n :

- Si n est pair, alors on le divise par 2
- Si n est impair, on lui retranche 1.

Notons $G(n)$ le nombre d'étapes pour cette stratégie gloutonne. On a par construction $G(1) = 0$ et pour $p \geq 1$, $G(2p) = 1 + G(p)$ et $G(2p+1) = 1 + G(2p) = 2 + G(p)$.

Le plus simple pour montrer que cette stratégie est optimale est de considérer le même type de formule pour l'optimal. En effet, $Opt(1) = 0$ et pour $p \geq 1$, $Opt(2p+1) = 1 + Opt(2p)$ (un entier impair > 1 ne peut être obtenu que par un ajout) et $Opt(2p) = 1 + \min(Opt(p), Opt(2p-1))$. On en déduit ainsi immédiatement que $\forall n \geq 2, Opt(n) \leq 1 + Opt(n-1)$.

On va montrer par récurrence **forte** que $\forall n \geq 1, Opt(n) = G(n)$.

- Initialisation : on a vu que $Opt(1) = G(1) = 0$.
- Hérédité : Soit $n > 1$ tel que la propriété soit vérifiée pour tout entier $1 \leq p < n$. Par disjonction de cas selon que n soit pair ou non :
  - si $n = 2p+1$, alors $G(2p+1) = 1 + G(2p) = 1 + Opt(2p) = Opt(2p+1)$
  - si $n = 2p$, quitte à traiter à part le cas trivial $n = 2$, on peut supposer que $p > 1$. On a $G(2p) = 1 + G(p)$ et $Opt(2p) = 1 + \min(Opt(p), Opt(2p-1))$. Or, par hypothèse de récurrence et comme $p - 1 \geq 1$,

$$\begin{aligned} Opt(2p-1) = G(2p-1) &= 1 + G(2p-2) = 2 + G(p-1) \\ &= 2 + Opt(p-1) > 1 + Opt(p-1) \geq Opt(p) \end{aligned}$$

  Donc $Opt(2p) = 1 + Opt(p) = 1 + G(p) = G(2p)$.

Dans tous les cas, on a bien la propriété voulue.

□

CHAPITRE 39

Division en sous-problèmes

DANS CE CHAPITRE

39.1 Diviser pour régner
39.2 Dichotomie pour passer de décision à optimisation
39.3 Problèmes

Dans ce chapitre, on étudie des algorithmes ayant la particularité de reposer sur une division d'un problème en plusieurs sous-problèmes :

- soit pour les résoudre récursivement, on parle alors de la méthode *diviser pour régner* ;
- soit pour résoudre ces sous-problèmes, peut-être d'une manière différente, afin d'obtenir une solution plus efficace au problème principal, on parle de *rencontre au milieu* (ou *meet in the middle*).

39.1 Diviser pour régner

39.1.1 Principe

Le principe des algorithmes de type *diviser pour régner* est de décomposer un problème en plusieurs sous-problèmes disjoints et de déduire des solutions de ces sous-problèmes une solution au problème de départ.

Le point clé pour ce principe est de pouvoir **fusionner** les solutions de sous-problèmes pour en faire une solution, et de pouvoir le faire dans un temps/espace raisonnable. On procède alors par récursivité en appliquant ce principe pour résoudre les sous-problèmes eux-mêmes jusqu'à tomber sur des sous-problèmes très simples.

Ainsi, si on a une entrée E, on va en déduire des entrées $E_1, \ldots, E_k$ pour des sous-problèmes telles que $|E_i| \leq \left\lceil \frac{|E|}{p} \right\rceil$, où $p \geq 2$ **et** $k \geq 1$ **sont indépendants de** E. Une fois les solutions $S_1, \ldots, S_k$ à ces sous-problèmes obtenues par des appels récursifs, on va en déduire par une **fusion** des solutions la solution S au problème initial.

En termes de complexité temporelle, si $T(n)$ est la complexité de résolution pour une entrée de taille n, on aura pour $n \geq 2$:

$$T(n) \leq kT\left(\left\lceil \frac{n}{p} \right\rceil\right) + f(n)$$

où $f(n)$ est le coût de la division et de la fusion, et en supposant, ce qui est très raisonnable, que T est croissante.

> **Remarque 39.1** La croissance de T est naturellement déduite du fait que le problème sur une entrée de taille n est **plus facile** que le problème sur une entrée de taille $m \geq n$. On peut ainsi penser au tri d'un tableau où il suffit de rajouter des éléments factices au début ou à la fin pour se ramener à un problème sur une entrée de taille plus grande.

On commence par faire des preuves dans des cas particuliers, puis on énoncera un théorème général dans la partie Principe général d'analyse des récurrences.

39.1.2 Somme d'éléments dans un tableau

On considère un tableau t de n nombres dont on cherche à calculer la somme $S(t) = \sum_{i=0}^{n-1} t[i]$. Il est possible de faire ce calcul très aisément avec une simple boucle :

OCaml

```
let somme t =
    let s = ref 0 in
    for i = 0 to Array.length t - 1 do
        s := !s + t.(i)
    done;
    !s
```

C

```
int somme(int *t, size_t nb)
{
    int s = 0;
    for (size_t i = 0; i < nb; i++)
    {
        s = s + t[i];
    }
    return s;
}
```

Python

```
def somme(t):
    s = 0
    for x in t:
        s = s + x
    return s
```

Il est possible, bien que cela ne soit pas naturel, de traiter ce problème avec un algorithme *diviser pour régner*. En effet, on peut couper le tableau en deux, calculer les deux sous-sommes S_1 et S_2, puis les fusionner de manière triviale en calculant la somme $S = S_1 + S_2$.

```
let rec somme_div t i j =
    if i > j
    then 0
    else if i = j
    then t.(i)
    else
        let m = i + (j-i)/2 in
        let s1 = somme_div t i m in
        let s2 = somme_div t (m+1) j in
        s1 + s2

let somme t = somme_div t 0 (Array.length t - 1)
```

Le cas `i > j` ne sert que pour le tableau vide : les appels récursifs se font toujours sur des sous-tableaux non vides.

Si $T(n)$ est le coût d'une somme d'un tableau à n éléments, on a donc $T(1) = 1$ et

$$\forall n \geq 2, T(n) \leq 2T(\lceil n/2 \rceil) + O(1)$$

Remarque 39.2 On constate ici qu'on utilise $\lceil x \rceil$ la partie entière supérieure liée à une majoration. En effet, si on coupe l'entrée en deux et qu'elle est de taille impaire $2q + 1$, une des deux moitiés sera de longueur $q + 1$ qui est la partie entière supérieure.

On aura alors $T(n) = T(q) + T(q + 1) + O(1) \leq 2T(q + 1) + O(1)$ par croissance de T.

Les résolutions de récurrence vont se faire en deux temps : un cas où toutes les divisions tombent juste, c'est-à-dire ici une puissance de deux, et un théorème où on s'y ramène pour un entier quelconque par encadrement et croissance de T.

Lemme 39.3

$\forall n \in \mathbb{N}, T(2^n) = O(2^n)$

Preuve On commence par expliciter le $O(1)$ présent dans la récurrence : il existe $M \geq 1$ tel que $\forall n \geq 2, T(n) \leq 2T(\lceil n/2 \rceil) + M$.

Montrons par récurrence que $\forall n \in \mathbb{N}, T(2^n) \leq M(2^{n+1} - 1)$.

- On a $T(2^0) = 1 \leq M = M(2^1 - 1)$.
- Si $T(2^n) \leq M(2^{n+1} - 1)$ alors $T(2^{n+1}) \leq 2T(2^n) + M \leq 2M(2^{n+1} - 1) + M = M(2^{n+2} - 1)$.

On a ainsi $T(2^n) \leq 2M \cdot 2^n$, donc $T(2^n) = O(2^n)$.

□

Théorème 39.4

$T(n) = O(n)$.

Preuve Soit $n \in \mathbb{N}^*$, il existe $q \in \mathbb{N}$ tel que $2^q \leq n < 2^{q+1}$. Par croissance de T, on a $T(n) \leq T(2^{q+1})$. Or, d'après le lemme, il existe M' tel que $\forall k, T(2^k) \leq M'2^k$. Donc $T(n) \leq M'2^{q+1} \leq 2M'n$.

Donc, $T(n) = O(n)$.

□

39.1.3 Tri fusion

L'algorithme du tri fusion est un des exemples les plus importants d'algorithmes *diviser pour régner* :

- Étant donnée une liste l de taille $n \geq 2$, on va considérer les sous-listes l_1 des valeurs d'indice pair et l_2 des valeurs d'indice impair.
- On trie ensuite l_1 et l_2 pour obtenir l_1' et l_2'.
- On fusionne ces deux listes pour obtenir $l' =$ fusion (l_1', l_2'), liste triée déduite de l.

Comme expliqué dans le paragraphe précédent, les tris de l_1 et l_2 s'effectuent eux aussi à l'aide d'un tri fusion.

Voici une implémentation en OCaml de cet algorithme :

```
let rec separe_en_deux l =
    match l with
    | [] -> ([], [])
    | [x] -> ([x], [])
    | x::y::q -> let l1, l2 = separe_en_deux q in
        (x::l1, y::l2)

let rec fusionne l1 l2 =
    match l1, l2 with
    | [], _ -> l2
    | _, [] -> l1
    | x::q1, y::q2 ->
        if x < y
        then x :: (fusionne q1 l2)
        else y :: (fusionne l1 q2)

let rec tri_fusion l =
    match l with
    | [] -> []
    | [x] -> [x]
    | _ ->
        let l1, l2 = separe_en_deux l in
```

```
        let l1p = tri_fusion l1 in
        let l2p = tri_fusion l2 in
        fusionne l1p l2p
```

✍

Propriété 39.5

Pour toute liste `l`, `tri_fusion l` termine et renvoie une liste triée dans l'ordre croissant contenant les mêmes éléments que `l`, avec les mêmes multiplicités.

Preuve

- `separe_en_deux l` renvoie deux listes de longueurs $\lceil n/2 \rceil$ et $\lfloor n/2 \rfloor$ dont la concaténation est une permutation de `l` (récurrence immédiate de deux en deux sur $n = |\, l \,|$).
- `fusionne l1 l2` termine car $|\, l_1 \,| + |\, l_2 \,|$ décroît strictement à chaque appel. Montrons par récurrence sur $|\, l_1 \,| + |\, l_2 \,|$ que si l_1 et l_2 sont triées, `fusionne l1 l2` est triée et contient exactement les éléments de l_1 et de l_2. C'est clair si une des listes est vide. Sinon, si $x < y$, par hypothèse de récurrence `fusionne q1 l2` est une liste triée formée des éléments de q_1 et l_2 ; tous sont $\geq x$ car l_1 est triée et $x < y \leq$ tout élément de l_2. Donc `x :: fusionne q1 l2` est triée. Le cas $x \geq y$ est symétrique.
- Pour `tri_fusion`, on raisonne par récurrence forte sur $n = |\, l \,|$. Les cas $n \leq 1$ sont immédiats. Si $n \geq 2$, on a $\lceil n/2 \rceil < n$ et $\lfloor n/2 \rfloor < n$, donc par hypothèse de récurrence les deux appels récursifs terminent et renvoient des listes triées formées des éléments de l_1 et l_2. Le point précédent permet de conclure.

□

On se concentre maintenant sur le calcul de la complexité temporelle :

- `separe_en_deux` consiste en un parcours linéaire de la liste `l` donc $O(|\, l \,|)$.
- `fusionne` supprime un élément d'une des deux listes à chaque appel récursif, donc une complexité en $O(|\, l_1 \,| + |\, l_2 \,|)$.
- Pour `tri_fusion` la situation est plus complexe en raison du double appel récursif. On va d'abord traiter le cas des listes contenant 2^k éléments.

Notons t_n la complexité temporelle pour $|\, l \,| = n$.

Lemme 39.6

$t_{2^n} = O(2^n \log_2 2^n)$

Preuve Par l'analyse de complexité des deux fonctions auxiliaires, on a pour $n \in \mathbb{N}$

$$t_{2^{n+1}} = 2t_{2^n} + O(2^n) \leq 2t_{2^n} + M2^n$$

où on peut supposer, quitte à l'augmenter, que $M \geq 1$ et $M \geq t_1$.

On va montrer par récurrence sur $n \in \mathbb{N}^*$ que $t_{2^n} \leq 2Mn2^n$.

- Initialisation : $t_{2^1} \leq 2t_1 + M \leq 3M \leq 4M = 2M \times 1 \times 2^1$.
- Hérédité : si $n \in \mathbb{N}^*$ et l'hypothèse est vérifiée pour t_{2^n}, alors $t_{2^{n+1}} \leq 4n2^nM + M2^n = (4n+1)M2^n \leq 4(n+1)M2^n = 2M(n+1)2^{n+1}$.

Ainsi $t_{2^n} = O(n2^n)$.

□

Théorème 39.7

$t_n = O(n \log_2 n)$

Preuve Le lemme assure qu'il existe M' tel que $\forall q \in \mathbb{N}^*, t_{2^q} \leq M'q2^q$.

Soit $n \geq 2$, on sait qu'il existe $q \in \mathbb{N}^*$ tel que $2^{q-1} \leq n < 2^q$ et donc $q - 1 \leq \log_2 n < q$. Par croissance de t (direct car on a plus d'éléments à trier) on a $t_n \leq t_{2^q} \leq M'q2^q \leq 2M'(1 + \log_2 n)n \leq 4M'n \log_2 n$ car $1 \leq \log_2 n$.

Ainsi, $t_n = O(n \log_2 n)$.

□

Remarque 39.8 On a utilisé implicitement la croissance de t_n ici : plus la liste est longue, plus on effectue d'opérations.

Le programme suivant présente une implémentation du tri fusion reposant sur des tableaux. Les sous-tableaux sont manipulés à l'aide de leurs indices de début et de fin comme pour la recherche dichotomique.

```
let fusionne t1 t2 =
    let n1 = Array.length t1 in
    let n2 = Array.length t2 in
    let i1 = ref 0 in
    let i2 = ref 0 in
    let i = ref 0 in
    let t = Array.make (n1+n2) t1.(0) in (* t1 est supposé non vide *)
    while !i1 < n1 || !i2 < n2 do
        if !i1 = n1 || (!i2 < n2 && t1.(!i1) > t2.(!i2))
        then begin
            t.(!i) <- t2.(!i2);
            incr i2
        end else begin
            t.(!i) <- t1.(!i1);
            incr i1
        end;
        incr i
    done;
    t

let rec tri_fusion_aux t i j =
    let n = j - i in
    if n >= 1
    then begin
        let m = i + n / 2 in
        let t1 = tri_fusion_aux t i m in
        let t2 = tri_fusion_aux t (m+1) j in
        fusionne t1 t2
    end
    else if n = 0
    then [|t.(i)|]
    else [||]

let tri_fusion t =
    let n = Array.length t in
    tri_fusion_aux t 0 (n-1)
```

39.1.4 Recherche dichotomique

On a déjà vu la recherche dichotomique : on coupe le tableau en deux mais on n'effectue qu'un seul appel récursif, la récurrence est donc

$$T(n) = T(\lceil n/2 \rceil) + O(1)$$

et on obtient facilement $T(n) = O(\log_2 n)$.

39.1.5 Principe général d'analyse des récurrences

On remarque ici qu'on retombe souvent sur la même méthode pour analyser des récurrences de la forme

$$\forall n \geq 2, T(n) \leq kT\left(\left\lceil \frac{n}{p} \right\rceil\right) + f(n)$$

Comme on ne considère que des suites positives et croissantes, on peut se ramener à une égalité par majoration. En effet, si S vérifie $S(1) = T(1)$ et $\forall n \geq 2, S(n) = kS\left(\left\lceil \frac{n}{p} \right\rceil\right) + f(n)$ alors on a $\forall n \in \mathbb{N}^*, T(n) \leq S(n)$ par une récurrence forte immédiate (car $\lceil n/p \rceil < n$ pour $n \geq 2$) et d'une majoration de $S(n)$ on en déduira directement une majoration pour $T(n)$.

Cette remarque peut donc se résumer ainsi :

Remarque 39.9 Il est toujours possible de se ramener à une relation de récurrence avec une égalité.

Que signifie cette relation de récurrence ? Que pour résoudre l'instance de taille n on va faire k appels récursifs à des instances de taille au plus $\lceil n/p \rceil$ et que le coût de la fusion des résultats sera $f(n)$.

On peut ainsi représenter les coûts de fusion cumulés qui sont associés à chaque niveau de l'arbre d'appels récursifs. Pour simplifier, on considère que $n = p^l$ ce qui permet de n'avoir que des divisions entières tout au long de l'arbre. On remarque qu'on peut s'y ramener par croissance en majorant n par une puissance de p et qu'on a alors $l = O(\log_p n)$.

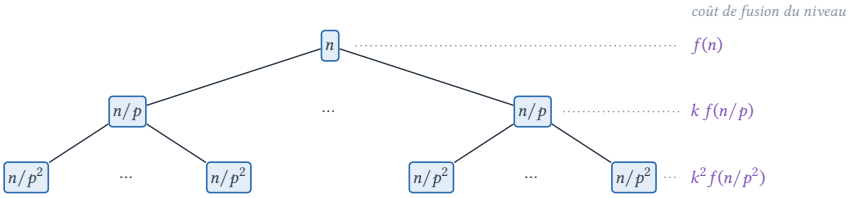


Figure 39.1 — Arbre des appels récursifs et coût de fusion cumulé de chaque niveau.

On a alors, en notant $f(1) = T(1)$ le coût d'une feuille, les coûts de fusion

$$T(n) = f(n) + kf(n/p) + k^2 f(n/p^2) + \ldots = \sum_{i=0}^{l} k^i f(n/p^i)$$

On a trois cas standard pour cette somme :

- Poids sur la racine, c'est $f(n)$ qui l'emporte sur les coûts et le reste de l'arbre n'a pas d'influence sur la complexité. Ici $T(n) = O(f(n))$.
- Poids sur les feuilles, le coût de fusion n'est pas important et c'est le nombre de feuilles qui compte, on a alors $T(n) = O(k^l f(1)) = O(k^l) = O(k^{\log_p n}) = O(n^{\log_p k})$.
- Poids réparti uniformément, on a chaque $k^i f(n/p^i) = O(f(n))$ et donc $T(n) = O(f(n) \log n)$.

Exemple 39.10

- Dans le cas de la multiplication de Karatsuba (voir le problème en fin de chapitre) on a $T(n) = 3T(\lceil n/2 \rceil) + O(n)$ donc les niveaux sont de plus en plus peuplés, on est dans le deuxième cas et le niveau final va l'emporter. Ainsi, $T(n) = O(n^{\log_2 3})$.
- Dans le cas du tri fusion $T(n) = 2T(\lceil n/2 \rceil) + O(n)$ on a un équilibre du travail total de fusion qui reste linéaire pour chaque niveau. Donc, on est dans le troisième cas et $T(n) = O(n \log n)$.

Quand le coût de fusion est polynomial, on peut rendre précise cette discussion. C'est le théorème suivant, souvent appelé **théorème maître**.

Théorème 39.11 — (théorème maître)

Soient $k \geq 1$ un réel, $p \geq 2$ un entier, $\alpha \geq 0$ et $C > 0$ des réels. Soit $T : \mathbb{N}^* \to \mathbb{R}_+$ une fonction **croissante** telle que

$$\forall n \geq 2, T(n) \leq kT\left(\left\lceil \frac{n}{p} \right\rceil\right) + Cn^\alpha$$

On pose $\beta = \log_p k$, de sorte que $k = p^\beta$. Alors :

- si $\alpha < \beta$, $T(n) = O(n^\beta)$ (poids sur les feuilles) ;
- si $\alpha = \beta$, $T(n) = O(n^\alpha \log n)$ (poids réparti) ;
- si $\alpha > \beta$, $T(n) = O(n^\alpha)$ (poids sur la racine).

Preuve *Cas des puissances de p.* Montrons par récurrence sur $l \in \mathbb{N}$ que

$$T(p^l) \leq k^l T(1) + C \sum_{i=0}^{l-1} k^i p^{(l-i)\alpha}$$

Pour $l = 0$, la somme est vide. Si c'est vrai pour l, comme $p^{l+1} \geq 2$ et $\lceil p^{l+1}/p \rceil = p^l$, on a $T(p^{l+1}) \leq kT(p^l) + Cp^{(l+1)\alpha} \leq k^{l+1}T(1) + C\sum_{i=0}^{l-1} k^{i+1}p^{(l-i)\alpha} + Cp^{(l+1)\alpha}$, ce qui est la formule au rang $l+1$ après le changement d'indice $i \leftarrow i+1$.

On pose $q = k/p^\alpha$, de sorte que $\sum_{i=0}^{l-1} k^i p^{(l-i)\alpha} = p^{l\alpha}\sum_{i=0}^{l-1} q^i$. On remarque que $q < 1 \Leftrightarrow \alpha > \beta$, que $q = 1 \Leftrightarrow \alpha = \beta$ et que $k^l = p^{l\beta}$.

- Si $\alpha > \beta$: $\sum_{i=0}^{l-1} q^i \leq \frac{1}{1-q}$ et $p^{l\beta} \leq p^{l\alpha}$, donc $T(p^l) \leq Ap^{l\alpha}$ avec $A = T(1) + \frac{C}{1-q}$.
- Si $\alpha = \beta$: la somme vaut l et $T(p^l) \leq (T(1) + Cl)p^{l\alpha}$.
- Si $\alpha < \beta$: $\sum_{i=0}^{l-1} q^i = \frac{q^l-1}{q-1} \leq \frac{q^l}{q-1}$ et $p^{l\alpha}q^l = k^l$, donc $T(p^l) \leq A'k^l = A'p^{l\beta}$ avec $A' = T(1) + \frac{C}{q-1}$.

Cas général. Soit $n \geq 2$ et $l = \lceil \log_p n \rceil \geq 1$, de sorte que $p^{l-1} < n \leq p^l$ et donc $p^l < pn$. Par croissance de T, $T(n) \leq T(p^l)$. Ainsi :

- si $\alpha > \beta$, $T(n) \leq A(pn)^\alpha = Ap^\alpha n^\alpha$;
- si $\alpha = \beta$, comme $l < 1 + \log_p n \leq \log_2 n + \log_2 n$ (car $n \geq 2$ et $p \geq 2$), $T(n) \leq (T(1) + 2C\log_2 n)p^\alpha n^\alpha = O(n^\alpha \log n)$;
- si $\alpha < \beta$, $T(n) \leq A'(pn)^\beta = A'kn^\beta$.

□

Remarque 39.12

- L'hypothèse de croissance est essentielle pour passer des puissances de p au cas général. On peut également remplacer $\lceil n/p \rceil$ par $\lfloor n/p \rfloor$: la croissance donne la même majoration.
- Si on a en plus une minoration $T(n) \geq kT(\lceil n/p \rceil) + cn^\alpha$ avec $c > 0$, on obtient de même les ordres de grandeur exacts (Θ au lieu de O, admis).
- Le théorème ne s'applique pas directement quand f n'est pas de la forme $O(n^\alpha)$ avec le bon exposant : par exemple, pour $f(n) = n\log n$ et $k = p = 2$, il donne seulement $O(n^{1+\varepsilon})$ pour tout $\varepsilon > 0$. On revient alors à la somme par niveaux.

Exemple 39.13

- Somme d'un tableau : $k = p = 2$, $\alpha = 0 < 1 = \beta$, donc $T(n) = O(n)$.
- Tri fusion : $k = p = 2$, $\alpha = 1 = \beta$, donc $T(n) = O(n\log n)$.
- Recherche dichotomique : $k = 1$, $p = 2$, $\alpha = 0 = \beta$, donc $T(n) = O(\log n)$.
- Karatsuba : $k = 3$, $p = 2$, $\alpha = 1 < \log_2 3 = \beta$, donc $T(n) = O(n^{\log_2 3})$.

Remarque 39.14 Pour retrouver des expressions rapidement, on peut étudier $u_n = \frac{T(p^n)}{k^n}$.

En effet, de la récurrence $T(p^n) = kT(p^{n-1}) + f(p^n)$ on déduit $u_n = u_{n-1} + \frac{f(p^n)}{k^n}$ et donc $u_n = \sum_{i=0}^{n} \frac{f(p^i)}{k^i} = O(g(n))$ qui se simplifie en général (avec toujours la convention $f(1) = T(1)$). On obtient ensuite $T(p^n) = k^n u_n = O(k^n g(n))$ puis, par croissance de T et pourvu que g soit croissante avec $g(n+1) = O(g(n))$, $T(n) = O(n^{\log_p k} g(\log_p n))$.

Par exemple, pour $T(n) = 2T(\lceil n/2 \rceil) + O(n)$ on a $u_n = T(2^n)/2^n = \sum_{i=0}^{n} \frac{O(2^i)}{2^i} = O(n)$ donc $T(2^n) = 2^n O(n) = O(n2^n)$ puis $T(n) = O(n\log n)$.

Autre exemple, pour $T(n) = 2T(\lceil n/2 \rceil) + O(1)$, on a $u_n = \sum_{i=0}^{n} 2^{-i} O(1) = O(1)\frac{1-\frac{1}{2^{n+1}}}{1-\frac{1}{2}} = O(1)$ car $\frac{1}{2^{n+1}} = o(1)$.

Ainsi $T(2^n) = O(2^n)$ puis $T(n) = O(n)$.

39.1.6 Nombre d'inversions

Définition 39.15

Soit t une structure séquentielle (tableau, liste, ...) contenant des valeurs comparables $a_0, ..., a_{n-1}$ et énumérées dans cet ordre au sein de t.

Une paire $(i, j) \in [\![0, n-1]\!]^2$ où $i < j$ est appelée une *inversion* de t lorsque $a_i > a_j$.

On note $I(t)$ le nombre d'inversions de t.

Remarque 39.16

- Le nombre d'inversions permet de mesurer à quel point t est non triée dans l'ordre croissant : $I(t) = 0$ si et seulement si t est triée, et $I(t) \leq \frac{n(n-1)}{2}$ avec égalité pour une suite strictement décroissante.
- Ce concept d'inversion est exactement celui utilisé pour les permutations en mathématiques : si $\sigma \in \mathfrak{S}_n$, il suffit de considérer $(\sigma(1), ..., \sigma(n))$.

On cherche dans ce paragraphe à calculer $I(t)$ efficacement. Remarquons tout d'abord qu'un algorithme naïf est en $O(n^2)$ où $|\, t \,| = n$ en explorant toutes les paires :

```c
size_t inversions(int *t, size_t taille)
{
    size_t inv = 0;
    for (size_t i = 0; i < taille; i++)
    {
        for (size_t j = i+1; j < taille; j++)
        {
            if (t[i] > t[j]) inv++;
        }
    }

    return inv;
}
```

On va maintenant donner un algorithme de type *diviser pour régner* :

- On sépare t en deux moitiés t_1 et t_2.
- On calcule $I(t_1)$ et $I(t_2)$ par des appels récursifs.
- On compte les inversions entre des éléments de t_1 et des éléments de t_2 :
  - Cela ne dépend pas de leur position dans t_1 ou dans t_2.
  - On peut donc trier t_1 en t_1' et t_2 en t_2'.
  - On compte $N(t_1, t_2) = N(t_1', t_2')$ le nombre d'inversions entre t_1' et t_2' en $O(n)$ par l'algorithme ci-dessous.
- On en déduit que $I(t) = I(t_1) + I(t_2) + N(t_1, t_2)$.

Ici, $N(t_1, t_2)$ est le nombre de couples (x, y), x élément de t_1 et y élément de t_2, tels que $x > y$: comme tous les éléments de t_1 précèdent ceux de t_2 dans t, ce sont exactement les inversions de t à cheval sur les deux moitiés, d'où la formule.

Remarque 39.17 Pour calculer le nombre d'inversions entre deux tableaux triés t_1' et t_2', on remarque que si $t_1'[i] > t_2'[j]$ alors $t_1'[i+1] \geq t_1'[i] > t_2'[j]$. Cela signifie qu'on peut compter les inversions de manière cumulative en avançant dans les deux tableaux en même temps :

```
let inversions_croisees t1 t2 =
    let i = ref 0 in
```

```
    let j = ref 0 in
    let n1 = Array.length t1 in
    let n2 = Array.length t2 in
    let inv = ref 0 in
    let cumul = ref 0 in
    while !i < n1 && !j < n2 do
        if t1.(!i) > t2.(!j)
        then ( incr j; incr inv; incr cumul )
        else ( incr i; inv := !inv + !cumul )
    done;
    !inv + (!cumul * (n1 - !i - 1))
```

✍ Ici, `!cumul` est toujours égal à `!j`, le nombre d'éléments de `t2` déjà dépassés, qui sont tous strictement inférieurs à `t1.(!i)`. Quand on avance `i`, on compte d'avance ces `!j` inversions pour le nouvel élément `t1.(!i)`. En sortie de boucle :

- si `!j = n2`, l'élément courant `t1.(!i)` a déjà été compté avec les `n2` éléments de `t2`, et chacun des `n1 - !i - 1` éléments restants forme `n2 = !cumul` inversions ;
- si `!i = n1`, le dernier passage dans la branche `else` a compté `!cumul` inversions pour un élément `t1.(n1)` qui n'existe pas : le terme `!cumul * (n1 - !i - 1) = - !cumul` corrige exactement cet excès.

Pour que cet algorithme fonctionne, il est nécessaire de trier à chaque étape les sous-tableaux, on obtient alors naïvement une récurrence en

$$T(n) \leq 2T(\lceil n/2 \rceil) + O(n \log n)$$

qui se résout en $T(n) = O(n(\log n)^2)$: pour $n = 2^l$, le niveau i de l'arbre d'appels coûte $2^i O\left(\frac{n}{2^i} \log \frac{n}{2^i}\right) = O(n \log n)$ et il y a $l + 1 = O(\log n)$ niveaux. Il est possible de faire mieux en remarquant qu'on peut calculer le nombre d'inversions en décorant le tri fusion car celui-ci fait naturellement calculer les sous-tableaux triés. On a donc une récurrence en

$$T(n) \leq 2T(\lceil n/2 \rceil) + O(n)$$

qui se résout en $T(n) = O(n \log n)$.

Exercice

Écrire le programme réalisant cet algorithme.

Corrigé

```
let inversions_croisees t1 t2 =
    let i = ref 0 in
    let j = ref 0 in
    let n1 = Array.length t1 in
    let n2 = Array.length t2 in
    let inv = ref 0 in
    let cumul = ref 0 in
    while !i < n1 && !j < n2 do
        if t1.(!i) > t2.(!j)
        then ( incr j; incr inv; incr cumul )
        else ( incr i; inv := !inv + !cumul )
    done;
    !inv + (!cumul * (n1 - !i - 1))

let split t =
    let n = Array.length t in
    let m = n/2 in
    Array.sub t 0 m, Array.sub t m (n-m)

let fusion t1 t2 =
    let n1 = Array.length t1 in
    let n2 = Array.length t2 in
```

```
    let t = Array.make (n1+n2) (t1.(0)) in
    let i = ref 0 in let j = ref 0 in
    for k = 0 to n1+n2-1 do
        if !j = n2 || (!i < n1 &&  t1.(!i) < t2.(!j))
        then ( t.(k) <- t1.(!i); incr i )
        else ( t.(k) <- t2.(!j); incr j )
    done;
    t

(* renvoie la version triée de t et son nombre d'inversions *)
let rec inversions_div t =
    let n = Array.length t in
    if n <= 1
    then t, 0
    else let t1, t2 = split t in
        let t1', n1 = inversions_div t1 in
        let t2', n2 = inversions_div t2 in
        let n12 = inversions_croisees t1' t2' in
        fusion t1' t2', n1+n2+n12
```

Le cas de base `n <= 1` traite aussi le tableau vide. Pour $n \geq 2$, les deux moitiés sont non vides, ce qui justifie l'accès à `t1.(0)` dans `fusion`. Ce programme a été testé contre l'algorithme naïf sur des dizaines de milliers de tableaux aléatoires, avec de nombreuses valeurs égales.

39.1.7 Points les plus proches

On considère le problème

Problème 39.18 — : PlusProchePaire

- Entrée : un ensemble P de $n \geq 2$ points dans le plan.
- Sortie : une paire $\{p, p'\}$ de points de P telle que $\text{dist}(p, p') = \| \overrightarrow{pp'} \|$ soit minimale.

Ce problème a déjà été étudié dans le chapitre Recherche par force brute où l'algorithme naïf en $O(n^2)$ a été amélioré en un algorithme en $O(n \log n)$.

On présente ici un algorithme diviser pour régner.

Si $n > 3$, l'idée est de séparer les points en fonction de la médiane des abscisses. On trie les points par abscisse croissante et on note S_1 les $\lceil n/2 \rceil$ premiers points et S_2 les $\lfloor n/2 \rfloor$ autres. Si x est l'abscisse du dernier point de S_1, les points de S_1 sont d'abscisses $\leq x$ et ceux de S_2 d'abscisses $\geq x$ (quand les abscisses sont distinctes, on peut aussi prendre pour x la moitié entre l'abscisse médiane et l'abscisse suivante). Ensuite, on applique l'algorithme récursivement pour obtenir d_1 la plus petite distance dans S_1 et d_2 la plus petite distance dans S_2. On pose $d = \min(d_1, d_2)$ et la question qui se pose est celle de calculer $d' = \min\{\text{dist}(p_1, p_2) \mid p_1 \in S_1, p_2 \in S_2\}$.

Comme on a déjà déterminé d, on peut se contenter de chercher à déterminer s'il existe $d' < d$. Pour cela, on peut se contenter de ne considérer dans S_1 que les points d'abscisse dans $]x - d; x]$ et pour S_2 dans $x; x + d[$. On a ainsi une bande à étudier. Comme il n'est pas possible d'exclure qu'un nombre important de points soit dans cette bande, on a *a priori* une fusion en $O(n)$.

On va considérer les points de bas en haut dans la bande, par exemple avec un tri initial selon les ordonnées. Ainsi, il sera inutile, quand on considère un point p d'ordonnée y, de considérer des points d'ordonnées $< y$ (la paire aura déjà été examinée depuis l'autre point). D'un autre côté, comme on cherche l'existence d'une distance $< d$, il est inutile de considérer des points d'ordonnée $> y + d$.

On se retrouve alors à comparer les points présents dans l'union de deux carrés de côté d de part et d'autre de la séparation :

![Le point p n'est comparé qu'aux points des deux carrés de côté d au-dessus de lui.{width=35%}

Cependant, on sait que les points de S_1 de la bande sont dans le carré gauche, ceux de S_2 dans le carré droit, et que deux points d'un même S_i sont à distance $\geq d_i \geq d$. Il n'est pas possible de placer plus que quatre points

dans un carré de côté d de sorte qu'aucune paire ne soit à distance $< d$: en découpant le carré en quatre carrés de côté $d/2$, dont la diagonale mesure $\frac{d}{\sqrt{2}} < d$, chacun contient au plus un point. La configuration extrémale correspond ainsi à placer les points aux sommets.

On en déduit donc qu'il y a au maximum quatre points dans le carré gauche et quatre points dans le carré droit, dont le point p considéré : il n'est pas nécessaire d'explorer plus que les sept points suivants.

Sur le schéma suivant, on a volontairement séparé les points autour de la frontière pour marquer cette différence.

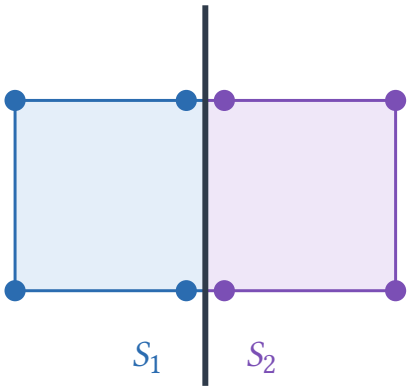


Figure 39.2 — Configuration extrémale : quatre points par carré.

> **Remarque 39.19** Cette analyse est très naïve, on peut démontrer qu'il suffit d'observer cinq autres points (admis).

Dans les programmes suivants, les points sont triés une fois pour toutes par abscisse dans `p_x` et l'ordre selon les ordonnées est maintenu sous la forme d'une liste `p_y` d'**indices** dans `p_x`. Séparer `p_y` selon l'indice (et non selon l'abscisse) garantit que chaque appel récursif travaille exactement sur les points d'indices entre `i` et `j`, même si plusieurs points ont la même abscisse.

OCaml

```
let distance (x1,y1) (x2,y2) =
    let v = (x1-.x2)*.(x1-.x2)
        +.(y1-.y2)*.(y1-.y2) in
    sqrt v

let closest_brute points i j =
    let d = ref (distance points.(i) points.(i+1)) in
    for k = i to j do
        for l = k+1 to j do
            let dist_kl =
                distance points.(k) points.(l) in
            if !d > dist_kl
            then d := dist_kl
        done
    done;
    !d

(* p_x : points triés par abscisse croissante
   p_y : indices dans p_x des points d'indice entre i et j,
         triés par ordonnée croissante *)
let rec closest_rec p_x p_y i j =
    if j-i+1 <= 3
    then closest_brute p_x i j
    else begin
        let m = (i+j)/2 in
        let mx = fst p_x.(m) in
        let p_y_l, p_y_r =
            List.partition (fun k -> k <= m) p_y in
```

```
        let d1 = closest_rec p_x p_y_l i m in
        let d2 = closest_rec p_x p_y_r (m+1) j in
        let d = min d1 d2 in
        let bande =
           Array.of_list
            (List.filter
              (fun k -> abs_float (fst p_x.(k) -. mx) < d) p_y) in
        let m_d = ref d in
        for k = 0 to Array.length bande - 1 do
            for l = k+1
                to min (k+7) (Array.length bande - 1) do
                let d_kl = distance p_x.(bande.(k)) p_x.(bande.(l)) in
                if d_kl < !m_d
                then m_d := d_kl
            done
        done;
        !m_d
    end

let closest points =
    let n = Array.length points in
    let p_x = Array.copy points in
    Array.sort (fun (x1,_) (x2,_) -> Stdlib.compare x1 x2) p_x;
    let p_y = List.sort
        (fun k1 k2 -> Stdlib.compare (snd p_x.(k1)) (snd p_x.(k2)))
        (List.init n (fun k -> k)) in
    closest_rec p_x p_y 0 (n - 1)
```

Python

```
from math import sqrt


def distance(p1, p2):
    x1, y1 = p1
    x2, y2 = p2
    return sqrt((x1 - x2)**2 + (y1 - y2)**2)


def closest_brute(points, i, j):
    m = distance(points[i], points[i+1])
    for k in range(i, j+1):
        for l in range(k+1, j+1):
            d = distance(points[k], points[l])
            if d < m:
                m = d
    return m


def closest_rec(points_x, points_y, i, j):
    # points_y : indices dans points_x des points d'indice entre i et j,
    # triés par ordonnée croissante
    if j-i+1 <= 3:
        return closest_brute(points_x, i, j)
    m = (i+j)//2
    mx = points_x[m][0]
    points_y_l = [k for k in points_y if k <= m]
    points_y_r = [k for k in points_y if k > m]
    d1 = closest_rec(points_x, points_y_l, i, m)
    d2 = closest_rec(points_x, points_y_r, m+1, j)
    d = min(d1, d2)
    bande = [points_x[k] for k in points_y if abs(points_x[k][0] - mx) < d]
    for k in range(len(bande)):
        for l in range(1, 8):
            if k+l >= len(bande):
                break
            d_kl = distance(bande[k], bande[k+l])
            if d_kl < d:
```

```
                d = d_kl
    return d


def closest(points):
    n = len(points)
    points_x = sorted(points, key=lambda p: p[0])
    points_y = sorted(range(n), key=lambda k: points_x[k][1])
    return closest_rec(points_x, points_y, 0, n-1)
```

Chaque appel sur n points effectue, en plus des deux appels récursifs, un travail en $O(n)$ (séparation de `p_y`, construction de la bande, au plus 7 distances par point de la bande). On a donc $T(n) \leq 2T(\lceil n/2 \rceil) + O(n)$, soit $T(n) = O(n \log n)$ d'après le théorème maître, auquel s'ajoutent les deux tris initiaux en $O(n \log n)$. Ces programmes ont été testés contre la recherche exhaustive sur des milliers d'ensembles aléatoires, y compris avec de nombreuses abscisses ou ordonnées égales et des points confondus.

39.2 Dichotomie pour passer de décision à optimisation

39.2.1 Couverture par des segments égaux

On considère ici n points sur la droite réelle. Le i-ème point est identifié par sa coordonnée x_i. On se pose alors, dans un premier temps, la question de savoir si on peut trouver k segments de longueur l tels que chaque point appartienne à au moins un de ces segments. Dans un second temps, on se posera la question de la longueur l minimale de ces segments.

Couverture par des segments de longueur donnée

On va ici résoudre le premier problème :

Problème 39.20 — : ExistenceCouvertureSegment

- Entrées :
  - n points sur la droite réelle $x_1, ..., x_n$
  - un entier $k \geq 1$
  - un réel $l \geq 0$
- Sortie : existe-t-il k segments $S_1, ..., S_k$ de longueur l tels que $\forall i \in [\![1, n]\!], \exists j \in [\![1, k]\!], x_i \in S_j$?

On remarque qu'un segment de longueur l est uniquement caractérisé par son extrémité gauche a : c'est $[a, a + l]$, et il contient x_i si et seulement si $a \in I_i = [x_i - l, x_i]$. Pour chaque x_i, il doit ainsi exister une extrémité gauche dans le segment I_i. On peut ainsi renverser le problème et en faire un problème de couverture de segments par des points : on cherche k points tels que chaque segment I_i contienne au moins un de ces points.

Problème 39.21 — : EnsembleIntersectantLigne

- Entrées :
  - n segments $I_i = [a_i, b_i]$
  - un entier $k \geq 1$
- Sortie : un ensemble P d'au plus k points tel que $\forall i \in [\![1, n]\!], \exists p \in P, p \in I_i$ en cas de succès, un échec sinon.

Ce problème peut se résoudre par un algorithme glouton :

- on commence avec $P = \emptyset$
- on trie les segments par b_i croissant
- pour chaque segment $I_i = [a_i, b_i]$, s'il ne contient aucun élément de P, on rajoute b_i à P.
- on répond avec un succès si $| P | \leq k$ (on peut alors compléter avec des points quelconques pour avoir exactement k points).

Comme les points sont ajoutés dans l'ordre croissant et sont tous $\leq b_i$ au moment où on traite I_i, ce segment contient un élément de P si et seulement si le dernier point ajouté est $\geq a_i$: chaque test est en $O(1)$. Cet algorithme est donc en $O(n \log n)$ en raison du tri initial.

Propriété 39.22

L'ensemble P construit par l'algorithme glouton intersecte tous les segments et est de cardinal minimal parmi les ensembles ayant cette propriété. En particulier, l'algorithme répond correctement.

Preuve Par construction, à l'issue du traitement de I_i, ce segment contient un élément de P, et P ne fait que croître.

Notons $I_{i_1}, ..., I_{i_m}$ les segments ayant provoqué un ajout, dans l'ordre de traitement, de sorte que $P = \{b_{i_1}, ..., b_{i_m}\}$ avec $b_{i_1} \leq ... \leq b_{i_m}$. Pour $s < t$, au moment de traiter I_{i_t}, le point $b_{i_s} \leq b_{i_t}$ n'est pas dans I_{i_t}, donc $b_{i_s} < a_{i_t}$: le segment I_{i_t} est entièrement à droite de I_{i_s}. Ces m segments sont donc deux à deux disjoints et tout ensemble de points intersectant tous les segments en contient au moins m. Ainsi $|P| = m$ est minimal, et il existe une solution à au plus k points si et seulement si $m \leq k$.

□

Longueur minimale

On considère maintenant le problème suivant :

Problème 39.23 — : CouvertureSegmentsMinimale

- Entrées :
  - $n \geq 1$ points à coordonnées entières sur la droite réelle $x_1, ..., x_n$
  - un entier $k \geq 1$
- Sortie : la longueur l minimale pour laquelle il existe une couverture des points par k segments de longueur l.

Deux remarques permettent de se ramener à une recherche parmi un nombre fini d'entiers :

- si une couverture par des segments de longueur l existe, il en existe une pour toute longueur $l' \geq l$ (on allonge les segments) ;
- si une couverture par k segments de longueur l existe, en regroupant les points couverts par un même segment, on obtient au plus k groupes dont l'étendue (plus grande moins plus petite coordonnée) est un entier $\leq l$, donc il existe aussi une couverture de longueur $\lfloor l \rfloor$. La longueur minimale est donc un entier.

On peut alors résoudre ce problème par dichotomie à l'aide de l'algorithme précédent :

- on considère $L = \left\lceil \frac{\max_{1 \leq i < j \leq n} |x_j - x_i|}{k} \right\rceil = \left\lceil \frac{D}{k} \right\rceil$ où le diamètre D se calcule en $O(n)$. Cela correspond à répartir de manière uniforme les segments pour couvrir les points : une telle couverture existe toujours.
- on peut considérer, sans avoir besoin de le calculer au préalable, le tableau V de booléens de longueur $L + 1$ tel que $V[l]$ indique s'il est possible de couvrir les points par k segments de longueur l. Pour obtenir la valeur $V[l]$ il suffit d'appliquer l'algorithme précédent. D'après la première remarque, V est croissant : de la forme [faux, ..., faux, vrai, ..., vrai].
- on effectue alors une recherche dichotomique du plus petit indice l tel que $V[l]$ soit vrai.

On obtient ainsi un algorithme en $O\left(n \log n \log \frac{D}{k}\right)$ qu'on peut considérer comme étant en $O(n \log n)$ en supposant que D est une constante.

Remarque 39.24 En triant les points une seule fois au début, les segments $I_i = [x_i - l, x_i]$ sont déjà triés par extrémité droite pour toute valeur de l : chaque test est alors en $O(n)$ et on obtient $O\left(n \log n + n \log \frac{D}{k}\right)$.

Cette dichotomie est en fait une instance d'un principe fondamental permettant de transformer un problème de décision (existe-t-il ?) en un problème d'optimisation (quelle est la valeur minimale/maximale ?).

Implémentation

La fonction `couvrable` applique l'algorithme glouton aux segments $I_i = [x_i - l, x_i]$, les points étant déjà triés : `dernier` est le dernier point ajouté à P, c'est-à-dire l'extrémité gauche du dernier segment de longueur l placé.

```
(* xs est trié par ordre croissant *)
let couvrable xs k l =
    let nb = ref 0 in
    let dernier = ref min_int in
    Array.iter (fun x ->
        if !dernier < x - l
        then begin
            dernier := x;
            incr nb
        end) xs;
    !nb <= k

let longueur_minimale points k =
    let xs = Array.copy points in
    Array.sort compare xs;
    let n = Array.length xs in
    let bas = ref 0 in
    let haut = ref ((xs.(n-1) - xs.(0) + k - 1) / k) in
    (* invariant : couvrable xs k !haut
       et non couvrable pour toute longueur < !bas *)
    while !bas < !haut do
        let m = (!bas + !haut) / 2 in
        if couvrable xs k m
        then haut := m
        else bas := m + 1
    done;
    !bas
```

Cette fonction a été testée contre une recherche exhaustive (découpage des points triés en au plus k groupes consécutifs) sur des milliers d'exemples aléatoires, y compris avec des points confondus et $k \geq n$.

39.3 Problèmes

39.3.1 Multiplication d'entiers

On considère ici le problème de la multiplication de deux entiers donnés en précision arbitraire par le tableau de leurs chiffres.

Ainsi, l'entier $123 = 1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0$ sera représenté par le tableau `[|3;2;1|]` de telle sorte que le nombre représenté par `t` soit $\sum_{k=0}^{n-1} t[k]10^k$ où $n = |\,t\,|$ la taille de t. On suppose que `t` ne finit pas par des `0`. Autrement dit : soit `t` est vide, soit la dernière valeur de `t` est non nulle.

On se convaincra aisément que les entiers naturels sont en bijection avec de telles représentations.

Question 1 Écrire une fonction `simpl : int array -> int array` qui renvoie un nouveau tableau obtenu en supprimant tous les 0 en fin de tableau : `simpl [|1;2;3;0;0|] = [|1;2;3|]`.

Question 2 Écrire une fonction `add : int array -> int array -> int array` qui calcule la somme de deux entiers en temps linéaire.

On va étudier ici plusieurs techniques de multiplication.

Multiplication naïve

On considère la multiplication posée comme étudiée à l'école primaire :

$$\begin{array}{ccc} 1 & 2 & 3 \\ \times & 4 & 2 \\ \hline 2 & 4 & 6 \end{array}$$

$$\begin{array}{cccc} 4 & 9 & 2 & \\ 5 & 1 & 6 & 6 \end{array}$$

Plus précisément, si $y = \sum_{j=0}^{m} y_j 10^j$, on a

$$x \times y = \sum_{j=0}^{m} (y_j \times x) 10^j$$

Question 3 Écrire une fonction `mult_digit : int -> int array -> int array` telle que `mult_digit a x` renvoie $a \times x$ où `a` est un chiffre et `x` un nombre.

Question 4 Écrire une fonction `mult_dec : int -> int array -> int array` telle que `mult_dec p x` renvoie $x10^p$ où p est un entier et `x` un nombre.

Question 5 Écrire une fonction `mult_naive : int array -> int array -> int array` réalisant cette multiplication.

Question 6 Quelle est la complexité de cette méthode en fonction de la longueur des entrées ?

Première approche diviser pour régner

On remarque que pour $0 \leq a, b, c, d < 10^m$:

$$(10^m a + b)(10^m c + d) = 10^{2m} ac + 10^m (ad + bc) + bd$$

On peut alors calculer récursivement les quatre produits ac, ad, bc et bd.

On en déduit donc une solution de type **diviser pour régner** de la multiplication :

- $mul(x, y, 1) = x \times y$
- pour $n \geq 2$, $mul(x, y, n) = 10^{2m} mul(a, c, m) + 10^m (mul(a, d, m) + mul(b, c, m)) + mul(b, d, m)$ où $x = a10^m + b$, $y = c10^m + d$ et $m = \lceil n/2 \rceil$.

où $0 \leq x, y < 10^n$.

Question 7 Déterminer avec précision la complexité $T(n)$ de $mul(x, y, n)$.

Que peut-on en conclure ?

Algorithme de Karatsuba

Face à ce problème, Karatsuba, alors âgé de 23 ans, est parti de la remarque suivante :

$$ac + bd - (a - b)(c - d) = ad + bc$$

qui permet de voir qu'en calculant ac, bd et le produit $(a - b)(c - d)$ on peut en déduire le coefficient $ad + bc$ de 10^m, et ainsi économiser une multiplication.

On en déduit ainsi une nouvelle solution pour la multiplication :

- $kar(x, y, 1) = x \times y$
- pour $n \geq 2$, $kar(x, y, n) = 10^{2m} X + 10^m (X + Y - Z) + Y$ où $x = a10^m + b$, $y = c10^m + d$, $m = \lceil n/2 \rceil$, $X = kar(a, c, m)$, $Y = kar(b, d, m)$ et $Z = kar(a - b, c - d, m)$.

La présence de nombres négatifs ne complique pas la complexité car il suffit de rajouter un booléen indiquant le signe à côté du tableau des chiffres d'un nombre.

Question 8 Déterminer avec précision la complexité $K(n)$ de $kar(x, y, n)$.

CHAPITRE 40

Programmation dynamique

DANS CE CHAPITRE

40.1 Exemple fondateur : plus long chemin dans un graphe orienté acyclique
40.2 Principe de la programmation dynamique
40.3 Exemples
40.4 Travaux pratiques

40.1 Exemple fondateur : plus long chemin dans un graphe orienté acyclique

On considère ici un graphe orienté acyclique et pondéré $G = (S, A, \pi)$. On a donc une fonction $\pi : A \to \mathbb{R}$ de pondération des arêtes.

Voici un exemple d'un tel graphe :

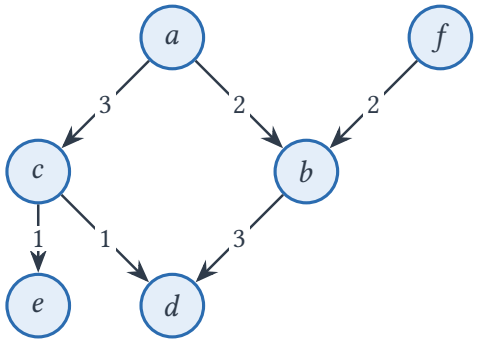


Figure 40.1 — Un graphe orienté acyclique pondéré.

On cherche à résoudre le problème suivant : étant donnés $s, t \in S$ tels que $s \rightsquigarrow t$, déterminer $f(s,t) = \max\{\pi(\varphi) \mid \varphi : s \rightsquigarrow t\}$, voire déterminer le chemin réalisant ce maximum.

Pour un graphe quelconque, ce problème est difficile dans le sens où on ne connaît pas d'autre solution que d'énumérer tous les chemins élémentaires, et il y en a un nombre exponentiel.

> **Remarque 40.1** Dans un graphe quelconque, on se limite aux chemins élémentaires (sinon, un cycle de poids strictement positif permettrait d'obtenir des chemins de poids arbitrairement grand). La version décision de ce problème est alors NP-complète, même dans le cas de poids positifs. On peut s'en convaincre en remarquant qu'un plus long chemin élémentaire dans un graphe non orienté complet à poids strictement positifs va forcément passer par tous les sommets (sinon on pourrait faire grossir le chemin en ajoutant un sommet manquant à son extrémité) : il permet donc de trouver des chemins hamiltoniens, ce qui est au cœur du problème du voyageur de commerce.

On peut établir une récurrence pour calculer $f(s,t)$. En effet, comme le graphe est acyclique, seul le chemin vide relie $s \rightsquigarrow s$ et donc $f(s,s) = 0$. Sinon, on peut essayer les chemins possibles :

$$f(s,t) = \begin{cases} 0 & \text{si } s = t \\ \max\{\pi(s \to s') + f(s',t) \mid s \to s' \in A, s' \rightsquigarrow t\} & \text{sinon.} \end{cases}$$

Comme on sait que $s \rightsquigarrow t$, le maximum du second cas est toujours bien défini car l'ensemble est non vide. En considérant que $\max(\emptyset) = -\infty$, ce qui revient à étendre f par $f(s,t) = -\infty$ quand $s \not\rightsquigarrow t$, on peut alors définir une récurrence plus simple :

$$f(s,t) = \begin{cases} 0 & \text{si } s = t \\ \max\{\pi(s \to s') + f(s',t) \mid s \to s' \in A\} & \text{sinon.} \end{cases}$$

Pour calculer f, on peut alors directement implémenter cette récurrence. En faisant cela, on va sûrement devoir calculer de nombreuses fois les mêmes valeurs, comme c'était le cas avec la suite de Fibonacci.

On pourrait ainsi résoudre ce problème de la même manière qu'on l'a fait pour de telles fonctions récursives : en utilisant un cache mémoire permettant d'éviter le recalcul coûteux de certaines valeurs.

On peut aussi tirer parti d'un parcours en profondeur du graphe. En effet, si on effectue un parcours en profondeur récursif depuis le sommet a dans l'exemple au-dessus et qu'on affiche le nom du sommet **au moment où on sort de la fonction récursive de parcours**, on pourrait avoir l'affichage suivant : *decba*. On a marqué en orange le parcours sur le graphe :

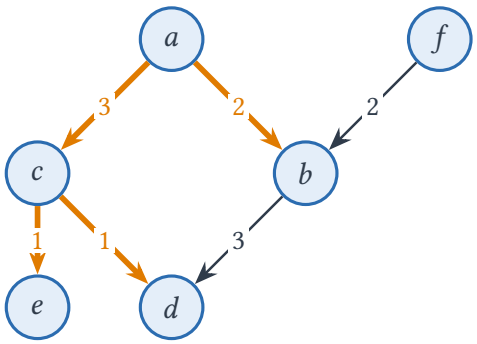


Figure 40.2 — Parcours en profondeur depuis a (en orange).

Le premier sommet affiché est nécessairement un puits, un sommet sans voisins sortants. Plus globalement, si $x \rightsquigarrow y$ avec $x \neq y$, on a forcément y affiché **avant** x. On parle de tri topologique.

> **Remarque 40.2** Cette notion est développée dans le chapitre sur les algorithmes de graphes. Avec la définition de ce chapitre, où un tri topologique place x avant y pour toute arête $x \to y$, l'ordre d'affichage obtenu ici est l'**inverse** d'un tri topologique. Dans tout ce chapitre, on appellera simplement tri topologique un ordre de calcul dans lequel chaque valeur apparaît après toutes celles dont elle dépend.

Ici, ce qui nous intéresse est directement lié au calcul des valeurs de $f(s,t)$. En effet, si on calcule $f(x,t)$ dans l'ordre donné par ce tri, on sera certain que toutes les valeurs dont on a besoin dans les max ont été calculées.

Dans notre exemple, si on cherche un chemin le plus long entre a et d, on va donc remplir le tableau suivant dans cet ordre.

Ici $f(d,d) = 0$.

d	e	c	b	a
0				

Comme il n'y a pas d'arêtes issues de e :

d	e	c	b	a
0	$-\infty$			

Ensuite, pour c, on effectue le maximum $\max(1-\infty, 1+0) = 1$:

d	e	c	b	a
0	$-\infty$	1		

Pour b on a un seul chemin $f(b,d) = 3$:

d	e	c	b	a
0	$-\infty$	1	3	

On a alors $f(a,d) = \max(3 + f(c,d), 2 + f(b,d)) = \max(4,5) = 5$. Ces deux valeurs ont bien été calculées.

d	e	c	b	a
0	$-\infty$	1	3	5

On conclut ainsi que $f(a,d) = 5$.

Notons qu'on aurait pu essayer d'effectuer un parcours glouton depuis a en choisissant l'arête de plus grand poids en descendant vers d. Dans ce cas, on aurait choisi le chemin $a \to c \to d$ de poids 4 qui n'est pas le plus long.

Pour résumer ce qu'on vient de dire :

- Il n'est pas possible de trouver le chemin le plus long avec un algorithme glouton.
- On peut écrire une récurrence exacte pour ce problème.
- Cette récurrence peut être calculée naïvement de manière exponentielle.
- On peut établir un mécanisme de cache pour ne calculer qu'une fois chaque valeur nécessaire.
- En raisonnant sur le graphe, on peut aussi déterminer un ordre permettant de remplir directement les valeurs nécessaires dans un tableau en faisant en sorte que toutes les valeurs utiles à un moment aient été précalculées.

Ce programme d'analyse de certains problèmes est très classique et est en général dénommé *programmation dynamique*.

> **Remarque 40.3** Le terme a été introduit par Richard Bellman dans les années 1950. Le mot *programmation* est à prendre au sens de planification, comme dans *programmation linéaire*, et non au sens de l'écriture de programmes. Selon Bellman lui-même, le mot *dynamique* évoquait l'aspect temporel, par étapes, des problèmes étudiés, et l'ensemble avait l'avantage d'être bien perçu par les financeurs militaires de ses recherches, peu enclins à soutenir des mathématiques.

Le problème du chemin le plus long est emblématique de la programmation dynamique : tout problème s'y ramène. En pratique il n'est pas nécessaire de définir le graphe ou d'exprimer un tri topologique explicitement, cependant, on a fait le choix ici de conserver ces termes tout le long du chapitre par cohérence.

40.2 Principe de la programmation dynamique

40.2.1 Problème et équation de Bellman

On considère ici un cadre assez large où on cherche à calculer une fonction $f : E \to \mathbb{R}$ où E est le domaine d'entrée et où on dispose :

- de valeurs connues pour $E_0 \subset E$;

- d'une équation pour $x \in E \setminus E_0$ permettant d'exprimer $f(x)$ en fonction de $f(y)$ pour $y \in \text{dep}(x)$ où dep : $E \to \mathcal{P}(E)$ est telle que toute chaîne $(a_1, a_2, \ldots)$ où $a_{i+1} \in \text{dep}(a_i)$ est finie et où, pour toute chaîne de longueur maximale, le dernier élément a_p est dans E_0.

En fait, cela signifie que dep induit un graphe orienté acyclique sur les sommets E où $v_+(x) = \text{dep}(x)$: on met une arête $x \to y$ quand le calcul de $f(x)$ utilise $f(y)$.

Il s'agit donc d'une équation de récurrence dans le sens où on exprime $f(x)$ en fonction d'autres appels à f. On parle de sous-problèmes.

Exemple 40.4

- Fibonacci. Ici, c'est un exemple *jouet* uniquement présent pour se concentrer sur l'équation. On a donc $F_0 = 0, F_1 = 1$ et pour $n > 1$, $F_n = F_{n-1} + F_{n-2}$.
- Rendu de monnaie. On cherche à calculer $f(n)$ le plus petit nombre de pièces pour rendre la monnaie dans le système monétaire S, qu'on suppose contenir une pièce de valeur 1 pour que tout montant puisse être rendu. On a $f(0) = 0$ et, pour $n > 0$, l'équation

$$f(n) = 1 + \min\{f(n-p) \mid p \in S, p \leq n\}$$

40.2.2 Résolution par récurrence naïve

On peut résoudre l'équation de récurrence naïvement par une fonction récursive.

Comme on a pu le voir, par exemple avec Fibonacci, les récurrences font rapidement apparaître des arbres d'appels de taille exponentielle.

> **Remarque 40.5** On peut se poser la question de la différence avec la méthode diviser pour régner. Dans diviser pour régner, les sous-problèmes ont un arbre récursif d'appel sans partage possible.

Exemple 40.6

Fibonacci. On a

OCaml

```
let rec fibo n =
    if n <= 1 then n
    else fibo (n-1) + fibo (n-2)
```

C

```
int fibo(int n)
{
    if (n <= 1)
        return n;
    return fibo(n-1) + fibo(n-2);
}
```

Python

```
def fibo(n):
    if n <= 1:
        return n
    return fibo(n-1) + fibo(n-2)
```

Rendu de monnaie. On peut directement écrire :

OCaml

```
let rec rendu s n =
    if n = 0
    then 0
    else
```

```
        let m = ref n in
        for i = 0 to Array.length s - 1 do
            if s.(i) <= n
            then m := min !m (rendu s (n - s.(i)))
        done;
        1 + !m
```

C

```c
int rendu(int *s, int ls, int n)
{
    if (n == 0)
        return 0;
    int m = n;
    for(int i = 0; i < ls; i++)
    {
        if(s[i] <= n)
        {
            int v = rendu(s, ls, n-s[i]);
            if (v < m)
                m = v;
        }
    }
    return 1 + m;
}
```

Python

```python
def rendu(s, n):
    if n == 0:
        return 0
    m = n
    for piece in s:
        if piece <= n:
            m = min(m, rendu(s, n-piece))
    return 1 + m
```

40.2.3 Mémoïsation et cache dynamique

Une manière d'améliorer la résolution par récurrence est d'utiliser un cache d'appel, c'est-à-dire un dictionnaire dont les clés sont dans E et les valeurs dans $\mathbb{R}$: pour calculer $f(x)$ on regarde si on a déjà une correspondance dans le cache, auquel cas on renvoie cette valeur, sinon, on calcule $f(x)$ avec la récurrence (qui utilisera donc le cache), puis on rajoute une entrée dans le cache pour x.

On dit qu'on a **mémoïsé** la fonction f.

En procédant ainsi, on effectue uniquement les calculs nécessaires, et une seule fois chacun.

Pour définir ce cache, on peut utiliser une table de hachage comme celle fournie par le module `Hashtbl` en `OCaml` ou par les dictionnaires de `Python`.

Exemple 40.7

Fibonacci.

OCaml

```
let fibo n =
    let cache = Hashtbl.create (n+1) in
    let rec fibo_aux n =
        try
            Hashtbl.find cache n
        with Not_found ->
            let v =
                if n <= 1 then n
```

```
                else fibo_aux (n-1) + fibo_aux (n-2)
            in Hashtbl.add cache n v;
            v
    in fibo_aux n
```

Python

```
cache = {}

def fibo(n):
    if n in cache:
        return cache[n]
    if n <= 1:
        v = n
    else:
        v = fibo(n-1) + fibo(n-2)
    cache[n] = v
    return v
```

Rendu de monnaie.

OCaml

```
let rendu s n =
    let cache = Hashtbl.create (n+1) in
    let rec rendu_aux n =
        try
            Hashtbl.find cache n
        with Not_found ->
            let v =
                if n = 0 then 0
                else let m = ref n in
                    for i = 0 to Array.length s - 1 do
                        if s.(i) <= n
                        then m := min !m (rendu_aux (n - s.(i)))
                    done;
                    1 + !m
            in Hashtbl.add cache n v;
            v
    in rendu_aux n
```

Python

```
cache = {}

def rendu(s, n):
    if n in cache:
        return cache[n]

    if n == 0:
        v = 0
    else:
        m = n
        for piece in s:
            if piece <= n:
                m = min(m, rendu(s, n-piece))
        v = 1 + m
    cache[n] = v
    return v
```

Attention, dans la version `Python`, le cache est global et ses clés ne dépendent que de `n` : il faut le vider avant d'utiliser `rendu` avec un autre système `s`. On verra plus bas comment éviter ce problème.

On remarque que le code a toujours la même structure : on part de

OCaml

```
let rec f x = expr
```

Python

```
def f(x):
    return expr
```

et on passe à :

OCaml

```
let f x =
    (* ici n doit être proche du nombre d'appels *)
    let cache = Hashtbl.create n in
    let rec f_aux x =
        try
            Hashtbl.find cache x
        with Not_found ->
            (* expr' = expr en changeant f x
               par f_aux x *)
            let v = expr' in
            Hashtbl.add cache x v;
            v
    in f_aux x
```

Python

```
cache = {}

def f(x):
    if x in cache:
        return cache[x]

    v = expr
    cache[x] = v
    return v
```

Remarque 40.8 Il n'est pas possible d'effectuer cette transformation d'expression automatiquement en `OCaml` puisqu'il serait nécessaire d'aller modifier le code de l'expression.

`OCaml` : cacher le cache dans une clôture

De la manière dont on a codé le cache, on va créer un nouveau cache à chaque calcul de $f(x)$ et jeter l'ancien. Il peut être intéressant de conserver un cache entre plusieurs appels. Pour cela, on peut utiliser la notion de clôture : une *clôture* est la donnée d'une fonction et des valeurs connues au moment de sa définition.

Ainsi, en écrivant plutôt :

```
let f =
    let cache = Hashtbl.create n in
    let rec f_aux x =
        try
            Hashtbl.find cache x
        with Not_found ->
            let v = expr' in
            Hashtbl.add cache x v;
            v
    in f_aux
```

On crée une fonction `f_aux` qui connaît le `cache` et peut l'utiliser et ce sera la valeur de `f`. Quand on appelle `f x` puis `f y`, on va en fait appeler `f_aux x` et `f_aux y` qui connaissent et manipulent le même `cache`.

Remarque 40.9 On peut expliciter la fonction avec un `fun` :

```
let f =
    let cache = Hashtbl.create n in
    let rec f_aux x =
        try
```

```
            Hashtbl.find cache x
        with Not_found ->
            let v = expr' in
            Hashtbl.add cache x v;
            v
    in
    fun x -> f_aux x
```

Dans le cas de `fibo`, on aurait alors le programme suivant (ici, il n'y a pas de `n` connu au moment de la création du cache : on donne une taille initiale quelconque, la table s'agrandit au besoin) :

```
let fibo =
    let cache = Hashtbl.create 100 in
    let rec fibo_aux n =
        try
            Hashtbl.find cache n
        with Not_found ->
            let v = if n <= 1
                    then n
                    else fibo_aux (n-1) + fibo_aux (n-2)
            in Hashtbl.add cache n v;
            v
    in fibo_aux
```

`Python` : cacher le cache dans un paramètre optionnel

En `Python`, on peut également passer par une sous-fonction et une clôture, comme dans le code suivant :

```
def fibo(n):
    cache = {}
    def fibo_aux(n):
        if n in cache:
            return cache[n]
        if n <= 1:
            v = n
        else:
            v = fibo_aux(n-1) + fibo_aux(n-2)
        cache[n] = v
        return v
    return fibo_aux(n)
```

Cependant, on utilise en général une unique fonction et un paramètre optionnel dans ce cas-là.

Il est possible de déclarer des paramètres comme optionnels à une fonction en rajoutant, après les arguments non optionnels, des arguments de la forme `argument=valeur_defaut`.

Ainsi,

```
def f(x, y=3):
    return x+y
```

permet de calculer `f(1,2)` pour renvoyer `1+2` mais si on utilise uniquement `f(1)`, la valeur de `y` correspondra à la valeur déclarée par défaut, ainsi la fonction renverra `1+3`.

Une idée est alors de passer le cache comme paramètre optionnel avec une valeur par défaut `{}`. Cependant, **il s'agit ici d'une erreur classique** en `Python` : les valeurs par défaut sont figées au moment de la création de la fonction et tous les appels sans valeur explicite du cache auront le même cache, et pas un nouveau cache créé pour l'occasion, ce qui n'est clairement pas le comportement voulu.

L'approche usuelle est donc de passer une valeur comme `None` au cache par défaut et de le créer dans ce cas-là :

```
def fibo(n, cache=None):
    if cache is None:
        cache = {} # créé à la volée
```

```python
    if n in cache:
        return cache[n]
    if n <= 1:
        v = n
    else:
        # ici crucial d'appeler avec le cache
        v = fibo(n-1,cache) + fibo(n-2,cache)
    cache[n] = v
    return v
```

40.2.4 Graphe de dépendances et tabulation

Comme on a un graphe de dépendance entre sous-problèmes qui est acyclique, on peut en déduire un tri topologique $x_0 < x_1 < \ldots < x_{n-1}$ qui garantit que si $f(x_i)$ a besoin dans son calcul de la valeur $f(x_j)$, alors $j < i$.

On peut alors créer un tableau de n cases où on place à la case i la valeur de $f(x_i)$.

Ce tableau peut se calculer par un remplissage par indice croissant. On dit qu'on a **tabulé** le problème.

Remarque 40.10 Il s'agit ici d'une politique agressive de remplissage du cache de mémoïsation où on remplit le cache sans même savoir si on va utiliser une valeur. Ce qu'on semble perdre en calculant *trop*, on le regagne dans la simplicité de la gestion d'un tableau plutôt qu'une table. Comme souvent, la question va être celle du compromis entre le nombre de valeurs utiles pour le calcul de $f(x)$ et le nombre de valeurs avant x dans le tri topologique considéré.

Exemple 40.11

Fibonacci. Ici, on a directement l'ordre des entiers qui est un tri topologique.

OCaml

```ocaml
let fibo n =
    if n <= 1 then n
    else
        let t = Array.make (n+1) 0 in
        t.(1) <- 1;
        for i = 2 to n do
            t.(i) <- t.(i-1) + t.(i-2)
        done;
        t.(n)
```

C

```c
int fibo(int n)
{
    if (n <= 1)
        return n;
    int *t = malloc(sizeof(int) * (n+1));
    t[0] = 0; t[1] = 1;
    for (int i = 2; i <= n; i++)
        t[i] = t[i-1] + t[i-2];
    int res = t[n];
    free(t);
    return res;
}
```

Python

```python
def fibo(n):
    if n <= 1:
        return n

    t = [ 0 ] * (n+1)
    t[1] = 1
```

```
    for i in range(2,n+1):
        t[i] = t[i-1] + t[i-2]

    return t[n]
```

Rendu de monnaie.

OCaml

```
let rendu s n =
    if n = 0 then 0
    else
        let t = Array.make (n+1) 0 in
        for i = 1 to n do
            let m = ref i in
            for j = 0 to Array.length s - 1 do
                let p = s.(j) in
                if p <= i
                then m := min !m t.(i - p)
            done;
            t.(i) <- 1 + !m
        done;
        t.(n)
```

C

```
int rendu(int *s, int ls, int n)
{
    if (n == 0)
        return 0;

    int *t = malloc(sizeof(int) * (n+1));
    t[0] = 0;
    for(int i = 1; i <= n; i++)
    {
        int m = i;
        for(int j = 0; j < ls; j++)
        {
            if(s[j] <= i)
            {
                int v = t[i-s[j]];
                if (v < m)
                    m = v;
            }
        }
        t[i] = 1 + m;
    }

    int res = t[n];
    free(t);
    return res;
}
```

Python

```
def rendu(s, n):
    if n == 0:
        return 0

    t = [ 0 ] * (n+1)

    for i in range(1, n+1):
        m = i
        for piece in s:
            if piece <= i and t[i-piece] < m:
                m = t[i-piece]
        t[i] = 1 + m
```

```
    return t[n]
```

40.2.5 Reconstruction

La plupart des problèmes ne sont pas juste des problèmes de calcul d'une valeur $f(x)$ mais des problèmes où on considère un ensemble F de valeurs à construire avec une fonction $\Phi : E \to F$ et une fonction d'objectif $\varphi : F \to \mathbb{R}$ et où $f(x) = \varphi(\Phi(x))$.

L'exemple principal est celui d'associer à tout $x \in E$ un ensemble de solutions $\text{sol}(x) \subset F$ et à chercher une solution minimisant (ou maximisant) l'objectif :

$$\Phi(x) \in \text{argmin}_{y \in \text{ sol }(x)} \varphi(y)$$

On a alors deux stratégies pour obtenir $\Phi(x)$ quand on a trouvé comment calculer $f(x)$. La première consiste à calculer Φ et f ensemble. Par exemple, dans le rendu de monnaie, quand on calcule le minimum, on sait quelle pièce le réalise :

OCaml

```
let rendu s n =
    if n = 0 then []
    else
        let t = Array.make (n+1) 0 in
        let util = Array.make (n+1) (-1) in
        for i = 1 to n do
            let m = ref i in
            for j = 0 to Array.length s - 1 do
                let p = s.(j) in
                if p <= i && t.(i-p) < !m
                then begin
                    util.(i) <- p;
                    m := t.(i - p)
                end
            done;
            t.(i) <- 1 + !m
        done;
        let rec rendu_expl n =
            if n = 0
            then []
            else let p = util.(n) in
                p :: rendu_expl (n-p)
        in rendu_expl n
```

Python

```
def rendu(s, n):
    if n == 0:
        return []

    t = [ 0 ] * (n+1)
    util = [ None ] * (n+1)

    for i in range(1, n+1):
        m = i
        for piece in s:
            if piece <= i and t[i-piece] < m:
                m = t[i-piece]
                util[i] = piece
        t[i] = 1 + m

    res = []
    while n > 0:
        res.append(util[n])
        n = n - util[n]
```

```
    return res
```

Cependant, si on connaît le tableau `t` complet, il est facile de constater qu'on peut retrouver une pièce ayant réalisé le minimum en cherchant $p \in S$, $p \leq x$, tel que $t.(x) = 1 + t.(x - p)$. On en déduit une reconstruction de la solution uniquement avec les valeurs.

Il est en général intéressant de faire cela, même si ce n'est pas forcément efficace.

40.2.6 Dimensionnalité réelle

Dans de nombreux cas, le problème n'est pas exprimé sur l'entrée E mais sur un autre ensemble plus petit. Par exemple, on peut chercher une valeur particulière à calculer depuis un tableau et penser que l'entrée est donnée par la taille du tableau, par exemple en raisonnant sur les préfixes, et, en fait, il est nécessaire de considérer tous les sous-tableaux pour calculer la solution. Ainsi, E est alors l'ensemble des couples d'indices (i, j) et on retrouve le problème initial en considérant $(0, n-1)$ où n est la longueur du tableau.

Il n'est ainsi pas rare d'avoir un problème portant sur un objet de taille n mais faisant apparaître des d-uplets et donc avec une taille réelle en n^d.

40.2.7 Complexité

Une fois identifiée la vraie entrée E, si le calcul de chaque valeur à partir de celles dont elle dépend se fait en temps constant, la complexité du calcul de $f(x)$ par tabulation est en $O(|\text{ avant }(x)|)$ où $\text{avant}(x) = \{y \in E \mid y \leq x\}$ dans le tri topologique considéré. Plus généralement, si le calcul de la valeur de y coûte $c(y)$, la complexité est en $O\left(\sum_{y \in \text{ avant }(x)} c(y)\right)$.

Comme vu dans la partie précédente, on a le plus souvent $O(n^d)$ sous-problèmes, où d est la dimensionnalité réelle du problème.

40.2.8 Déterminer un tri topologique

Le plus souvent, on va tabuler le problème sur des tableaux à d dimensions. La question du tri topologique revient alors à déterminer d boucles `for` pour remplir dans le bon ordre.

Il peut être intéressant de réaliser un schéma pour visualiser les dépendances et déterminer un ordre de remplissage. Par exemple, dans le schéma suivant, le calcul de la case orange utilise les cases bleues situées sur sa ligne et sur sa colonne, jusqu'à la diagonale (c'est ce qui se passe pour le produit de matrices traité plus loin).

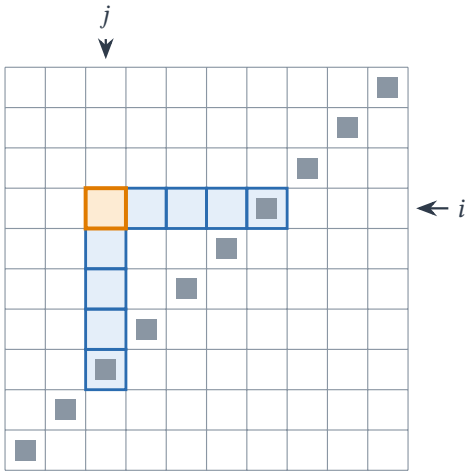


Figure 40.3 — La case orange dépend des cases bleues de sa ligne et de sa colonne, jusqu'à la diagonale.

40.2.9 Résumé

On considère ici que nos données sont comme des plateaux de jeux qu'on construit en effectuant des coups. On considère qu'on a une mesure de qualité sur ces plateaux et on cherche le meilleur plateau possible.

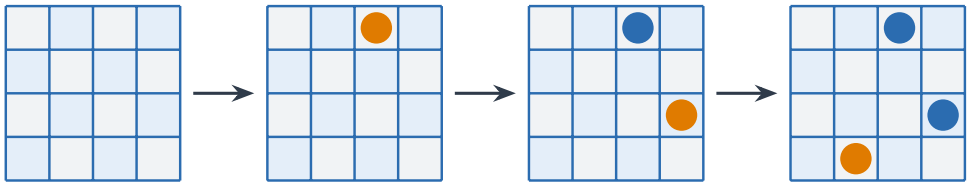


Figure 40.4 — Un plateau construit coup par coup ; le dernier pion posé est en orange.

Avec une telle présentation, on peut construire l'arbre des plateaux où le plateau vide est à la racine et où on passe d'un plateau à un autre en effectuant des coups. La question qu'on se pose alors est de savoir comment parcourir cet arbre pour trouver la meilleure solution.

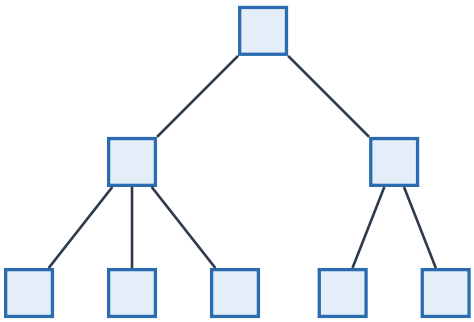


Figure 40.5 — Arbre des plateaux.

Si la taille de l'arbre est raisonnable par rapport à la taille de l'instance du problème considéré, par exemple la dimension du plateau, on peut se permettre de l'énumérer exhaustivement par backtracking (voir le chapitre sur la recherche par force brute). Mais dans le cas classique où la taille de l'arbre est exponentielle, c'est impraticable.

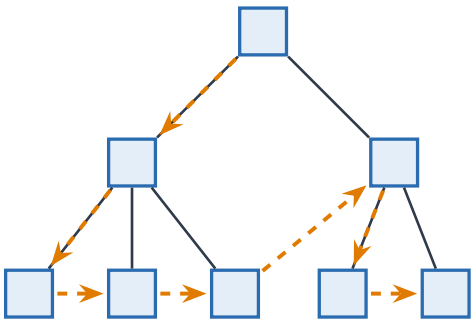


Figure 40.6 — Exploration exhaustive de l'arbre par backtracking.

Une solution peut être de chercher la meilleure branche en choisissant à chaque étape le mouvement qui semble être le meilleur, c'est-à-dire qui maximise notre mesure de qualité. On parle de **choix glouton** (voir le chapitre sur les algorithmes gloutons) mais un tel choix peut très bien se tromper et ne pas se rendre compte qu'un mouvement a priori moins bon pouvait se révéler meilleur ensuite.

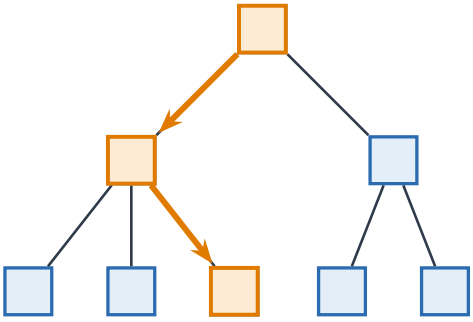


Figure 40.7 — Une seule branche, obtenue par choix gloutons successifs.

On parle de programmation dynamique dans les cas où cet arbre présente une notion naturelle de partage : certains sous-arbres sont complètement dupliqués parce qu'on aboutit au même plateau par différents mouvements.

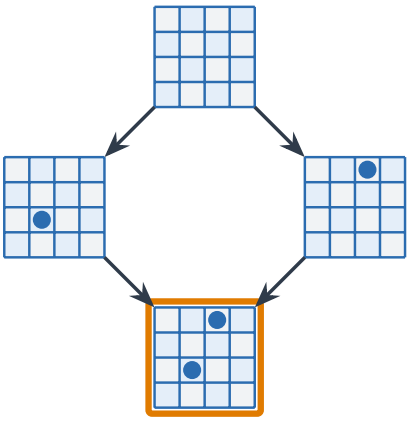


Figure 40.8 — Deux suites de coups différentes mènent au même plateau (en orange).

On peut donc alors le résoudre récursivement par exploration exhaustive mais se souvenir quand on a déjà exploré une position. C'est exactement comme lorsqu'on navigue en ligne : si on a déjà téléchargé une page, quand on y accède à nouveau on utilise le cache mémoire.

De l'arbre des plateaux, on peut déduire un graphe des plateaux où on ne fait apparaître qu'une fois chaque plateau. En faisant cela, on perd la structure arborescente mais on garde l'acyclicité. On obtient un graphe orienté acyclique (un DAG) connexe.

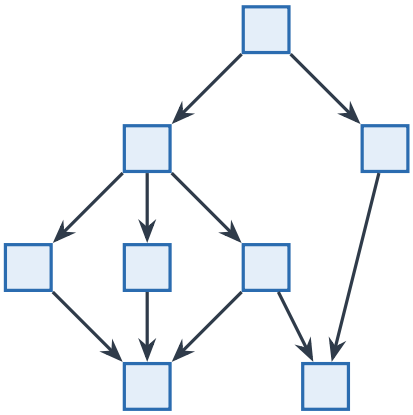


Figure 40.9 — Graphe des plateaux : chaque plateau n'y apparaît qu'une fois.

Ce graphe exprime des dépendances : si $x \to y$, pour avoir la valeur de x, je dois connaître la valeur de y. On trouve facilement un ordre d'énumération respectant cela, on parle de tri topologique. Ici, on l'indique par les numéros croissants.

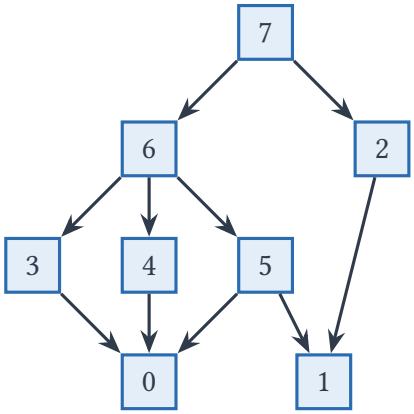


Figure 40.10 — Un tri topologique du graphe des plateaux.

Une fois cet ordre obtenu, on réalise qu'il suffit de calculer la valeur cherchée dans un tableau en suivant ces indices. En faisant ainsi, on a la garantie d'avoir toujours rempli dans le tableau les valeurs utiles dans la suite. C'est la tabulation !

> **Remarque 40.12** Face à un problème qui semble soluble par programmation dynamique on va donc :
>
> 1. Identifier la notion de problème et de sous-problèmes permettant d'établir une équation de Bellman pour un **calcul**.
> 2. Là, on a deux choix :
>   - résoudre l'équation de Bellman naïvement avec un cache de mémoïsation ;
>   - ou tabuler les résultats après avoir déduit un ordre de parcours compatible avec les dépendances de l'équation.
> 3. Puis, on décore le code du calcul pour permettre de construire les objets optimaux recherchés, ce qu'on peut éventuellement faire directement depuis la table ou le cache de calcul.

40.3 Exemples

Pour chacun de ces exemples, on cherchera à mettre en place une solution par mémoïsation puis par tabulation, avec une reconstruction directe ou indirecte.

40.3.1 Sous-séquence contiguë maximale

On considère un tableau t de taille $n \geq 1$ contenant des entiers et on demande de trouver le couple (i, l) tel que la somme

$$t[i] + \ldots + t[i + l]$$

soit la plus grande possible.

Si on considère le tableau :

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$t[i]$	13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

La plus grande somme est $t[7] + t[8] + t[9] + t[10] = 43$.

Notion de sous-problème et récurrence

Ici, on va introduire un sous-problème différent qui est celui de calculer

$$f(i) = \max_{i \leq k < n} \sum_{j=i}^{k} t[j]$$

c'est-à-dire la plus grande somme démarrant par $t[i]$.

La connaissance d'une seule valeur de f ne sera pas suffisante pour déterminer la plus grande somme quelconque. Cependant, il va suffire de calculer le maximum des valeurs prises par f.

Pour le tableau précédent on obtient :

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$t[i]$	13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7
$f(i)$	13	−3	−4	21	1	4	20	43	25	5	12	−5	−4	18	3	7

L'intérêt d'avoir rigidifié la somme en fixant un point de départ est de pouvoir exprimer la relation suivante :

$$f(n-1) = t[n-1] \qquad \forall i < n-1,\ f(i) = t[i] + \begin{cases} f(i+1) & \text{si } f(i+1) > 0 \\ 0 & \text{sinon} \end{cases}$$

Notons qu'on peut écrire aussi $f(i) = t[i] + \max(f(i+1), 0)$.

Preuve Pour $i < n-1$, une somme démarrant par $t[i]$ est soit $t[i]$ seul, soit $t[i] + \sum_{j=i+1}^{k} t[j]$ avec $k \geq i+1$, c'est-à-dire $t[i]$ plus une somme démarrant par $t[i+1]$. Le maximum de ces dernières est $f(i+1)$, donc $f(i) = \max(t[i], t[i] + f(i+1)) = t[i] + \max(0, f(i+1))$.

□

Récurrence naïve et mémoïsation

On écrit directement le programme résolvant cette récurrence :

OCaml

```
let rec f t i =
    let n = Array.length t in
    if i = n - 1
    then t.(n-1)
    else t.(i) + max 0 (f t (i+1))
```

Python

```
def f(t, i):
    if i == len(t) - 1:
        return t[i]
    return t[i] + max(0, f(t,i+1))
```

On peut alors répondre au problème initial avec :

OCaml

```
let maxsubarray t =
    let n = Array.length t in
    let m = ref (f t 0) in
    for i = 1 to n-1 do
        m := max !m (f t i)
    done;
    !m
```

Python

```
def maxsubarray(t):
    m = f(t, 0)
    for i in range(1, len(t)):
        m = max(m, f(t,i))
    return m
```

Ici, l'arbre d'appels de `f` n'est pas exponentiel (chaque appel en fait au plus un autre), mais `maxsubarray` recalcule les mêmes valeurs : on effectue $O(n^2)$ opérations au total.

On adapte directement le code vu précédemment pour ajouter un cache de mémoïsation :

OCaml

```
let cache = Hashtbl.create 42
    (* taille quelconque ici *)

let rec f t i =
    try
        Hashtbl.find cache i
    with Not_found ->
        let v = let n = Array.length t in
            if i = n - 1
            then t.(n-1)
            else t.(i) + max 0 (f t (i+1))
        in Hashtbl.add cache i v; v
```

Python

```
cache = {}

def f(t, i):
    if i in cache:
        return cache[i]

    if i == len(t) - 1:
        v = t[i]
    else:
        v = t[i] + max(0, f(t,i+1))
    cache[i] = v
    return v
```

En plaçant le cache en dehors de la fonction, on permet à `maxsubarray` de ne pas recalculer les valeurs mais on rend la fonction à usage unique : il faudrait vider le cache au début de `maxsubarray`.

40.3.1.2.1 `OCaml` : cache dans une clôture paramétrée

Il est beaucoup plus élégant de cacher le cache dans une clôture, mais en faisant cela, on va se heurter au fait qu'il faut un cache partagé entre les différentes valeurs de i mais propre à un t donné. L'astuce réside dans le fait de *couper* la fonction en insérant la création du cache après le paramètre `t` :

```
let f t =
    let n = Array.length t in
    let cache = Hashtbl.create (n+1) in
        (* ici on a une bonne valeur *)
    let rec f_aux i =
        try
            Hashtbl.find cache i
        with Not_found ->
            let v =
                if i = n - 1
                then t.(n-1)
                else t.(i) + max 0 (f_aux (i+1))
            in Hashtbl.add cache i v; v
    in f_aux
```

> **Remarque 40.13** On finit sur `f_aux`, ce qui fait que `f t` est une fonction `int -> int`, mais on aurait pu écrire `fun i -> f_aux i` pour rendre la fonctionnelle plus explicite.

Tabulation

On tabule `f` directement en remplissant un tableau de droite à gauche vu que $f(i)$ dépend de $f(i+1)$:

OCaml

```
let maxsubarray t =
    let n = Array.length t in
    let f = Array.make n 0 in
    f.(n-1) <- t.(n-1);
    for i = n-2 downto 0 do
        f.(i) <- t.(i) + max 0 f.(i+1)
    done;
    let m = ref f.(0) in
    for i = 1 to n-1 do
        m := max !m f.(i)
    done;
    !m
```

Python

```
def maxsubarray(t):
    n = len(t)
    f = [ 0 ] * n
    f[n-1] = t[n-1]
    for i in reversed(range(0,n-1)):
        f[i] = t[i] + max(0, f[i+1])

    # on peut se servir de max
    # sur un tableau ici
    return max(f)
```

Reconstruction

Lorsqu'on effectue le calcul du maximum dans la fonction précédente, il est direct de garder dans une variable l'indice où il se produit. Cependant, en procédant ainsi, on ne peut pas en déduire facilement la longueur de la somme. Pour ce faire, on va créer un nouveau tableau indiquant à l'indice i la longueur de la plus grande somme commençant par $t[i]$.

En effet, si on note $l(i)$ l'entier tel que $f(i) = \sum_{k=i}^{i+l(i)} t[k]$ obtenu par la récurrence, on a $l(n-1) = 0$ et

$$\forall i < n-1, l(i) = \begin{cases} 1 + l(i+1) \text{ si } & f(i+1) > 0 \\ 0 & \text{sinon} \end{cases}$$

Sur l'exemple précédent on va donc calculer les valeurs suivantes :

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$t[i]$	13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7
$f(i)$	13	−3	−4	21	1	4	20	43	25	5	12	−5	−4	18	3	7
$l(i)$	0	0	8	7	6	5	4	3	2	1	0	0	3	2	1	0

Il n'y a alors plus qu'à lire le couple $(i, l(i))$ correspondant au maximum pour f, ici $(7, 3)$.

On programme cela directement en décorant le programme précédent :

OCaml

```
let maxsubarray t =
    let n = Array.length t in
    let f = Array.make n 0 in
    let l = Array.make n 0 in
    f.(n-1) <- t.(n-1);
    for i = n-2 downto 0 do
        if f.(i+1) > 0
        then begin
            f.(i) <- t.(i) + f.(i+1);
            l.(i) <- 1 + l.(i+1)
        end else f.(i) <- t.(i)
    done;
    let m = ref 0 in
    for i = 1 to n-1 do
```

```
        if f.(!m) < f.(i)
        then m := i
    done;
    !m, l.(!m)
```

Python

```
def maxsubarray(t):
    n = len(t)
    f = [ 0 ] * n
    l = [ 0 ] * n
    f[n-1] = t[n-1]
    for i in reversed(range(0,n-1)):
        if f[i+1] > 0:
            f[i] = t[i] + f[i+1]
            l[i] = 1 + l[i+1]
        else:
            f[i] = t[i]

    m = 0
    for i in range(1, n):
        if f[m] < f[i]:
            m = i

    return m, l[m]
```

40.3.2 Chemins monotones

On considère ici le problème de la plus grande somme en partant du sommet d'un triangle de nombres et descendant soit en bas à gauche soit en bas à droite d'un cran.

```
             75
            95 64
           17 47 82
          18 35 87 10
         20 04 82 47 65
        19 01 23 75 03 34
       88 02 77 73 07 63 67
      99 65 04 28 06 16 70 92
     41 41 26 56 83 40 80 70 33
    41 48 72 33 47 32 37 16 94 29
   53 71 44 65 25 43 91 52 97 51 14
  70 11 33 28 77 73 17 78 39 68 17 57
 91 71 52 38 17 14 91 43 58 50 27 29 48
63 66 04 68 89 53 67 30 73 16 69 87 40 31
04 62 98 27 23 09 70 98 73 93 38 53 60 04 23
```

Remarque 40.14 Il s'agit des problèmes Project Euler 18 et Project Euler 67, dont est extrait le triangle ci-dessus.

Tout d'abord, on va représenter le problème sous la forme d'un tableau de tableaux, ligne par ligne, ainsi :

OCaml

```
let t = [|
   [| 75 |];
   [| 95; 64 |];
   [| 17; 47; 82 |];
   ...
|]
```

Python

```
t = [
  [ 75 ],
  [ 95, 64 ],
  [ 17, 47, 82 ],
```

```
  ...
]
```

Ce tableau a n lignes ayant de 1 à n éléments. Les successeurs de l'élément d'indice (i, j) sont les éléments d'indice $(i+1, j)$ et $(i+1, j+1)$. On notera ici $t(i, j)$ la valeur en (i, j) dans le tableau t.

Notion de sous-problèmes et récurrence

Ici, on considère $f(i, j)$ la plus grande somme d'un chemin partant de la i-ème ligne et j-ème colonne. On cherche donc $f(0, 0)$ pour répondre au problème initial.

Sur la dernière ligne, pas le choix, on ne peut considérer que l'élément lui-même, on a donc $f(n-1, j) = t(n-1, j)$.

Pour les autres éléments, il faut considérer les deux possibilités de premier déplacement, en bas à gauche ou à droite, et prendre le choix qui aboutit à la plus grande somme :

$$\forall i \in [\![0, n-2]\!], \forall j \in [\![0, i]\!], f(i, j) = t(i, j) + \max(f(i+1, j), f(i+1, j+1))$$

Et ainsi, on a les dépendances $\text{dep}(i, j) = \{(i+1, j), (i+1, j+1)\}$.

Résolution naïve et mémoïsation

On implémente directement cette récurrence :

OCaml

```
let rec maxpath t i j =
    if i = Array.length t - 1
    then t.(i).(j)
    else t.(i).(j) + max (maxpath t (i+1) j)
                         (maxpath t (i+1) (j+1))
```

Python

```
def maxpath(t, i, j):
    if i == len(t) - 1:
        return t[i][j]
    return t[i][j] + max(maxpath(t, i+1, j),
                         maxpath(t, i+1, j+1))
```

Chaque appel sur une ligne autre que la dernière en provoque deux : on effectue $2^n - 1$ appels, ce qui est déjà impraticable pour le triangle à 100 lignes du problème 67.

Remarque 40.15 Pour lire le triangle depuis un fichier, on peut utiliser la fonction suivante :

OCaml

```
let lit_triangle f =
    let rec aux () =
        try
            let l = input_line f in
            Array.of_list
                (List.map int_of_string
                    (String.split_on_char ' ' l))
            :: aux ()
        with End_of_file -> []
    in Array.of_list (aux ())
```

Python

```
def lit_triangle(f):
    t = []
    for l in open(f):
        t.append([ int(tok) for tok
                            in l.split() ])
    return t
```

Ici `f` est un canal d'entrée (obtenu par `open_in`) en `OCaml` et un nom de fichier en `Python`. La version `OCaml` suppose que les nombres sont séparés par exactement une espace.

On modifie alors cette fonction assez rapidement pour utiliser la mémoïsation. On prend garde notamment au fait que les clés sont maintenant des couples d'indices (i, j).

OCaml

```
let cache = Hashtbl.create 42

let rec maxpath t i j =
    try
        Hashtbl.find cache (i,j)
    with Not_found ->
        let v = if i = Array.length t - 1
            then t.(i).(j)
            else t.(i).(j) + max (maxpath t (i+1) j)
                                 (maxpath t (i+1) (j+1))
        in Hashtbl.add cache (i,j) v; v
```

Python

```
cache = {}

def maxpath(t, i, j):
    if (i,j) in cache:
        return cache[(i,j)]
    if i == len(t) - 1:
        v = t[i][j]
    else:
        v = t[i][j] + max(maxpath(t, i+1, j),
                          maxpath(t, i+1, j+1))
    cache[(i,j)] = v
    return v
```

Remarque 40.16 `OCaml` - Là encore, on peut cacher le dictionnaire dans une clôture pour créer un cache par tableau `t`.

```
let maxpath t =
    let n = Array.length t in
    let cache = Hashtbl.create (n*n) in
    let rec aux i j =
        try
            Hashtbl.find cache (i,j)
        with Not_found ->
            let v = if i = Array.length t - 1
                then t.(i).(j)
                else t.(i).(j) + max (aux (i+1) j)
                                     (aux (i+1) (j+1))
            in Hashtbl.add cache (i,j) v; v
    in aux
```

Tabulation

On peut représenter le graphe de dépendances des sous-problèmes :

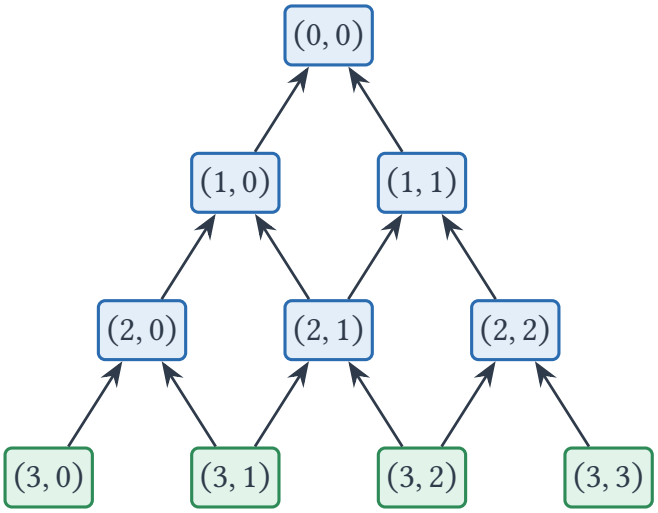


Figure 40.11 — Dépendances entre sous-problèmes (i, j) ; en vert, les cas de base (dernière ligne).

On remarque assez naturellement qu'on peut obtenir un tri topologique en traitant ce graphe du bas vers le haut. Ici, il suffit donc de tabuler dans le même format triangulaire que `t` (ou dans une matrice par simplicité) et de remplir la case (i, j) par `i` décroissant puis `j` croissant (ou décroissant).

On en déduit alors le programme suivant :

OCaml

```
let maxpath t =
    let n = Array.length t in
    let som = Array.make_matrix n n 0 in
    for j = 0 to n-1 do
        som.(n-1).(j) <- t.(n-1).(j)
    done;
    for i = n-2 downto 0 do
        for j = 0 to i do
            som.(i).(j) <- t.(i).(j)
                + max som.(i+1).(j) som.(i+1).(j+1)
        done
    done;
    som.(0).(0)
```

Python

```
def maxpath(t):
    n = len(t)
    # matrice sans partage par compréhension
    som = [ [ 0 ] * n for _ in range(n) ]
    for j in range(n):
        som[n-1][j] = t[n-1][j]
    for i in reversed(range(n-1)):
        for j in range(i+1):
            som[i][j] = t[i][j] \
                + max(som[i+1][j], som[i+1][j+1])
    return som[0][0]
```

Sur le triangle précédent, on obtient 1074.

Reconstruction

Pour reconstruire un chemin de plus grande somme, il suffit de se souvenir des embranchements pris en calculant le maximum. Pour cela, on construit de même un tableau (ou une matrice pour simplifier) `choix` où `choix(i,j)` indique `Gauche` ou `Droite`.

OCaml

```
type choix = Gauche | Droite

let maxpath t =
    let n = Array.length t in
    let som = Array.make_matrix n n 0 in
    let ch = Array.make_matrix n n None in
    for j = 0 to n-1 do
        som.(n-1).(j) <- t.(n-1).(j)
    done;
    for i = n-2 downto 0 do
        for j = 0 to i do
            ch.(i).(j) <-
                Some (if som.(i+1).(j) > som.(i+1).(j+1)
                      then Gauche else Droite);
            som.(i).(j) <- t.(i).(j)
                + max som.(i+1).(j) som.(i+1).(j+1)
        done
    done;
    let rec chemin i j =
        (i,j) :: match ch.(i).(j)
        with None -> [ ]
        | Some Gauche -> chemin (i+1) j
        | Some Droite -> chemin (i+1) (j+1)
    in chemin 0 0
```

Python

```
def maxpath(t):
    n = len(t)
    # matrice sans partage par compréhension
    som = [ [ 0 ] * n for _ in range(n) ]
    ch = [ [ None ] * n for _ in range(n) ]
    for j in range(n):
        som[n-1][j] = t[n-1][j]
    for i in reversed(range(n-1)):
        for j in range(i+1):
            if som[i+1][j] > som[i+1][j+1]:
                ch[i][j] = 'gauche'
            else:
                ch[i][j] = 'droite'
            som[i][j] = t[i][j] \
                + max(som[i+1][j], som[i+1][j+1])

    # on remonte alors dans le tableau pour
    # reconstruire le chemin
    posi, posj = 0, 0
    chemin = []
    while ch[posi][posj] != None:
        chemin.append( (posi, posj) )
        if ch[posi][posj] == 'droite':
            posj = posj+1
        posi = posi+1
    chemin.append( (posi, posj) )
    return chemin
```

Complexité

Il faut explorer chaque case pour pouvoir conclure : comme il y en a $\frac{n(n+1)}{2}$, la complexité temporelle est au moins quadratique, et on atteint ici $O(n^2)$, chaque case étant calculée en temps constant. Comme on a utilisé des tableaux de mêmes dimensions pour la tabulation ou le cache, on est également en $O(n^2)$ en espace.

40.3.3 Produit de matrices

On souhaite calculer un produit $A_0 \ldots A_{n-1}$ de matrices. Pour cela, on aimerait découper le calcul en produits de deux matrices, cela revient à placer des parenthèses.

Le problème est qu'une multiplication matricielle effectue un nombre d'opérations de l'ordre de mnp quand on multiplie une matrice (m, n) et une matrice (n, p). Pour simplifier, on considérera que la constante ici vaut 1 et on cherche le plus petit nombre d'opérations à effectuer.

Notion de sous-problème et récurrence

Tout d'abord, on remarque que si le produit A_iA_{i+1} est licite, alors A_i a autant de colonnes que A_{i+1} a de lignes. Il suffit donc de ne donner que le tableau des nombres de lignes et le nombre de colonnes de A_{n-1}.

On considère donc le tableau `t` de `n+1` valeurs où la matrice A_i a `t[i]` lignes et `t[i+1]` colonnes.

Exemple 40.17

Si on considère $A_0 \ldots A_5$ de dimensions respectives $(30, 35), (35, 15), (15, 5), (5, 10), (10, 20)$ et $(20, 25)$, on aura ainsi le tableau :

i	0	1	2	3	4	5	6
`t[i]`	30	35	15	5	10	20	25

Effectuer un produit et placer des parenthèses va naturellement induire deux sous-problèmes portant sur des matrices rangées à la suite. C'est pour cela qu'on introduit le problème du calcul de $f(i, j)$, le plus petit nombre d'opérations pour multiplier les matrices $A_iA_{i+1} \ldots A_j$.

Si $i = j$, on n'a pas de multiplication à faire et on a directement $f(i, i) = 0$. Sinon, on va forcément faire en dernier un produit $(A_i \ldots A_k)(A_{k+1} \ldots A_j)$ pour $i \leq k < j$, d'une matrice $(t[i], t[k+1])$ par une matrice $(t[k+1], t[j+1])$. Les deux facteurs se calculent indépendamment, chacun de manière optimale. Il suffit alors d'effectuer le moins coûteux :

$$\forall i < j, f(i, j) = \min_{i \leq k < j} (t[i]t[k+1]t[j+1] + f(i, k) + f(k+1, j))$$

Récurrence naïve et mémoïsation

OCaml

```
let rec matmult t i j =
    if i = j
    then 0
    else let m = ref (t.(i) * t.(i+1) * t.(j+1)
            + matmult t (i+1) j) in
        for k = i+1 to j-1 do
            m := min !m
                (t.(i) * t.(k+1) * t.(j+1)
                    + matmult t i k + matmult t (k+1) j)
        done;
        !m
```

Python

```
def matmult(t, i, j):
    if i == j:
        return 0
    m = (t[i] * t[i+1] * t[j+1]
         + matmult(t, i+1, j))
    for k in range(i+1, j):
        m = min(m,
                t[i] * t[k+1] * t[j+1]
                + matmult(t, i, k)
                + matmult(t, k+1, j))
    return m
```

Ici, la valeur initiale de `m` correspond au cas $k = i$, pour lequel $f(i, i) = 0$.

OCaml

```
let matmult t =
    let n = Array.length t in
    let cache = Hashtbl.create (n*n) in
    let rec aux i j =
        try
            Hashtbl.find cache (i,j)
        with Not_found ->
            let v =  if i = j
                then 0
                else
                    let m = ref (t.(i) * t.(i+1) * t.(j+1)
                        + aux (i+1) j) in
                    for k = i+1 to j-1 do
                        m := min !m
                            (t.(i) * t.(k+1) * t.(j+1)
                                + aux i k
                                + aux (k+1) j)
                    done;
                    !m
            in Hashtbl.add cache (i,j) v; v
    in aux
```

Python

```
cache = {}

def matmult(t, i, j):
    if (i,j) in cache:
        return cache[(i,j)]

    if i == j:
        v = 0
    else:
        m = (t[i] * t[i+1] * t[j+1]
             + matmult(t, i+1, j))
        for k in range(i+1, j):
            m = min(m,
                    t[i] * t[k+1] * t[j+1]
                    + matmult(t, i, k)
                    + matmult(t, k+1, j))
        v = m
    cache[(i,j)] = v
    return v
```

Tabulation

Ici, il est plus simple de représenter le graphe de dépendance sur une grille où chaque case représente un couple (i, j). Ainsi, dans l'exemple considéré, les cases $(0, 2), (1, 2), (2, 3), (2, 4)$ et $(2, 5)$ dépendent de $(2, 2)$. Ces relations de dépendance sont indiquées sur le schéma suivant en orange :

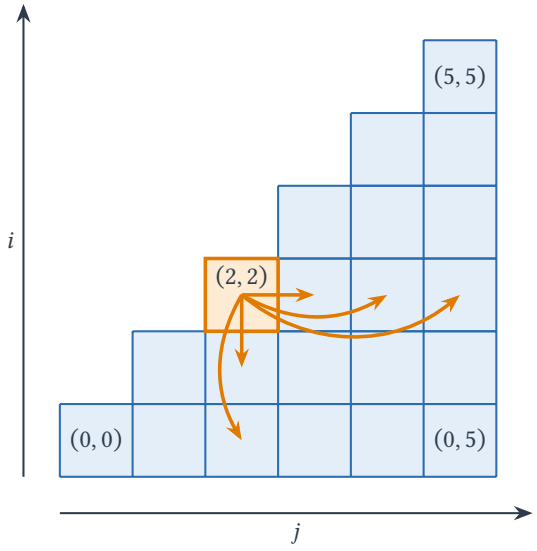


Figure 40.12 — Les cases qui dépendent de $(2, 2)$ (flèches orange).

Pour trouver un tri topologique, il faut renverser ces dépendances : pour calculer (i, j) on va avoir besoin de toutes les cases au dessus et à gauche jusqu'à tomber sur la diagonale. On peut donc remplir de haut en bas et de gauche à droite, ce qui assure qu'à tout moment, lorsqu'on remplit une case, on a déjà calculé ses dépendances :

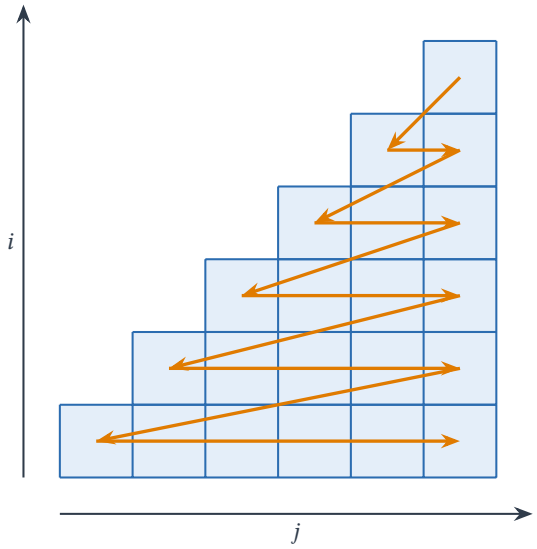


Figure 40.13 — Ordre de remplissage : de haut en bas, puis de gauche à droite.

Cela correspond au programme suivant :

OCaml

```
let matmult t =
    let n = Array.length t in
    let cout = Array.make_matrix (n-1) (n-1) 0 in
    for i = n-2 downto 0 do
        for j = i+1 to n-2 do
            let m = ref (t.(i) * t.(i+1) * t.(j+1)
                        + cout.(i+1).(j)) in
            for k = i+1 to j-1 do
                m := min !m
                    (t.(i) * t.(k+1) * t.(j+1)
                        + cout.(i).(k)
                        + cout.(k+1).(j))
```

```
            done;
            cout.(i).(j) <- !m
        done
    done;
    cout.(0).(n-2)
```

Python

```
def matmult(t):
    n = len(t)
    cout = [ [ 0 ] * (n-1) for _ in range(n-1) ]
    for i in reversed(range(0,n-1)):
        for j in range(i+1,n-1):
            m = (t[i] * t[i+1] * t[j+1]
                 + cout[i+1][j])
            for k in range(i+1, j):
                m = min(m,
                        t[i] * t[k+1] * t[j+1]
                        + cout[i][k]
                        + cout[k+1][j])
            cout[i][j] = m
    return cout[0][n-2]
```

Sur l'exemple, on obtient 15125 opérations. On remplit $O(n^2)$ cases, chacune en $O(n)$: la complexité temporelle est en $O(n^3)$ et la complexité spatiale en $O(n^2)$.

`OCaml` : reconstruction

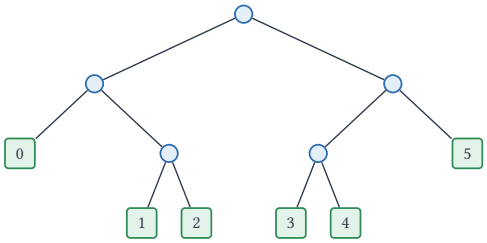
Pour reconstruire un parenthésage on va construire un arbre en plaçant dans un tableau l'arbre du calcul de $A_i \ldots A_j$ à la case (i, j).

```
type arbre = Feuille of int | Noeud of arbre * arbre

let matmult t =
    let n = Array.length t in
    let cout = Array.make_matrix (n-1) (n-1) 0 in
    let arbres = Array.make_matrix (n-1) (n-1) (Feuille (-1)) in
    for i = n-2 downto 0 do
        arbres.(i).(i) <- Feuille i;
        for j = i+1 to n-2 do
            let m = ref (t.(i) * t.(i+1) * t.(j+1)
                         + cout.(i+1).(j)) in
            let a = ref (Noeud(arbres.(i).(i),
                               arbres.(i+1).(j))) in
            for k = i+1 to j-1 do
                let v = t.(i) * t.(k+1) * t.(j+1)
                        + cout.(i).(k)
                        + cout.(k+1).(j)
                in
                if v < !m
                then begin
                    m := v;
                    a := Noeud(arbres.(i).(k),
                               arbres.(k+1).(j))
                end
            done;
            cout.(i).(j) <- !m;
            arbres.(i).(j) <- !a
        done
    done;
    arbres.(0).(n-2)
```

Exemple 40.18

Dans l'exemple précédent on va renvoyer l'arbre



qui correspond au produit $(A_0(A_1A_2))((A_3A_4)A_5)$.

40.3.4 Plus longue sous-séquence croissante

On se donne un tableau t de $n \geq 1$ nombres entiers et on demande de déterminer un p-uplet $(i_1, ..., i_p)$, où $i_1 < i_2 < ... < i_p$ et $t(i_1) < t(i_2) < ... < t(i_p)$, pour lequel p est maximal.

Notion de sous-problème et récurrence

On reprend ici directement ce qui avait été fait pour le premier exemple : on considère $f(i)$ la longueur du plus grand p-uplet $(i, i_2, ..., i_p)$ où $i < i_2 < ... < i_p$ et $t(i) < t(i_2) < ... < t(i_p)$. Autrement dit, on oblige la sous-séquence à commencer par $t(i)$.

On a donc $f(n-1) = 1$ car elle ne peut contenir qu'une valeur. Ensuite, on considère

$$\Phi(i) = \{j \mid j > i \wedge t[j] > t[i]\}$$

et on a

$$\forall i < n-1, f(i) = \begin{cases} 1 & \text{si } \Phi(i) = \emptyset \\ 1 + \max_{j \in \Phi(i)} f(j) & \text{sinon.} \end{cases}$$

La longueur cherchée est alors $\max_{0 \leq i < n} f(i)$.

Récurrence naïve et mémoïsation

OCaml

```
let rec croissante t i =
    let n = Array.length t in
    if i = n-1
    then 1
    else let m = ref 1 in
        for j = i+1 to n-1 do
            if t.(j) > t.(i)
            then m := max !m (1 + croissante t j)
        done;
        !m
```

Python

```
def croissante(t, i):
    n = len(t)
    if i == n-1:
        return 1
    m = 1
    for j in range(i+1,n):
        if t[j] > t[i]:
            m = max(m,
                1 + croissante(t, j))
    return m
```

OCaml

```
let croissante t =
    let n = Array.length t in
    let cache = Hashtbl.create n in
```

```
    let rec aux i =
        try
            Hashtbl.find cache i
        with Not_found ->
            let v = if i = n-1
                    then 1
                    else let m = ref 1 in
                        for j = i+1 to n-1 do
                            if t.(j) > t.(i)
                            then m := max !m (1 + aux j)
                        done;
                        !m
            in Hashtbl.add cache i v; v
    in aux
```

Python

```
cache = {}

def croissante(t, i):
    n = len(t)
    if i in cache:
        return cache[i]

    if i == n-1:
        v = 1
    else:
        m = 1
        for j in range(i+1,n):
            if t[j] > t[i]:
                m = max(m,
                    1 + croissante(t, j))
        v = m
    cache[i] = v
    return v
```

Tabulation

On reprend ici aussi le même principe que pour la tabulation de la sous-séquence maximale, c'est-à-dire qu'on va remplir de droite à gauche. La fonction renvoie le tableau des $f(i)$, dont il reste à prendre le maximum.

OCaml

```
let croissante t =
    let n = Array.length t in
    let f = Array.make n 1 in
    for i = n-2 downto 0 do
        for j = i+1 to n-1 do
            if t.(j) > t.(i)
            then f.(i) <- max f.(i) (1 + f.(j))
        done
    done;
    f
```

Python

```
def croissante(t):
    n = len(t)
    f = [ 1 ] * n
    for i in reversed(range(n-1)):
        for j in range(i+1, n):
            if t[j] > t[i]:
                f[i] = max(f[i], 1 + f[j])
    return f
```

La complexité temporelle est en $O(n^2)$.

`OCaml` : reconstruction

On va conserver dans un tableau le suivant potentiel d'un indice dans une plus grande sous-séquence croissante commençant par lui.

```
let croissante t =
    let n = Array.length t in
    let f = Array.make n 1 in
    let suivant = Array.make n None in
    for i = n-2 downto 0 do
        for j = i+1 to n-1 do
            if t.(j) > t.(i) && f.(i) < 1 + f.(j)
            then begin
                f.(i) <- 1 + f.(j);
                suivant.(i) <- Some j
            end
        done
    done;
    f, suivant

let sous_sequence_croissante t =
    let f, suivant = croissante t in
    let n = Array.length t in
    let mi = ref 0 in
    for i = 1 to n-1 do
        if f.(i) > f.(!mi)
        then mi := i
    done;
    let seq = ref [ !mi ] in
    while suivant.(!mi) <> None do
        mi := Option.get suivant.(!mi);
        seq := !mi :: !seq
    done;
    List.rev !seq
```

40.3.5 Distance d'édition

Exercice

Étant donné deux chaînes de caractères A et B de longueurs respectives n et m, on veut transformer la chaîne A en la chaîne B en effectuant un minimum d'opérations parmi celles-ci :

- supprimer un caractère n'importe où ;
- insérer un nouveau caractère n'importe où ;
- changer un caractère en n'importe quel autre.

Ce nombre d'opérations est appelé la distance d'édition de A à B. Écrire un algorithme permettant de la calculer.

Par exemple : chien → chen → chan → chat

Corrigé

On note $u = u_1 \ldots u_n$ et $v = v_1 \ldots v_m$ les deux mots et $d(u, v)$ leur distance d'édition, aussi appelée distance de Levenshtein. Comme chaque opération a une opération inverse, on a $d(u, v) = d(v, u)$, et $d(u, v) \geq ||u| - |v||$ car chaque opération modifie la longueur d'au plus 1.

Alignements. Pour obtenir des sous-problèmes, on raisonne sur les préfixes. Il est plus commode pour cela de représenter une transformation par un alignement, qui traite les lettres de la gauche vers la droite. Soit Σ l'alphabet et - $\notin \Sigma$. Un **alignement** de u et v est un couple de mots (x, y) sur $\Sigma \cup \{\text{-}\}$ de même longueur ℓ, tels qu'en effaçant les - de x on obtient u, en effaçant ceux de y on obtient v, et sans position k telle que $x_k = y_k = \text{-}$. Le **coût** de l'alignement est le nombre de positions k telles que $x_k \neq y_k$.

Une colonne $(a, \text{-})$ correspond à la suppression de a, une colonne $(\text{-}, b)$ à l'insertion de b et une colonne (a, b) avec $a \neq b$ à une substitution. Un alignement de coût c donne donc une transformation de u en v en c opérations. On admet la réciproque : la distance d'édition est égale au coût minimal d'un alignement de u et v (admis).

Par exemple, l'alignement suivant de `chien` et `chat` est de coût 3 :

```
c  h  i  e  n
c  h  a  t  -
```

Récurrence. Pour $i \in [\![0, n]\!]$ et $j \in [\![0, m]\!]$, on note $D(i, j)$ la distance d'édition entre $u_1 \ldots u_i$ et $v_1 \ldots v_j$, qui est donc le coût minimal d'un alignement de ces deux préfixes. On a $D(i, 0) = i$, $D(0, j) = j$ et, pour $i, j \geq 1$, en notant $\delta = 0$ si $u_i = v_j$ et $\delta = 1$ sinon :

$$D(i, j) = \min(D(i-1, j) + 1, D(i, j-1) + 1, D(i-1, j-1) + \delta)$$

En effet, le seul alignement de $u_1 \ldots u_i$ et du mot vide est formé de i colonnes $(u_k, \text{-})$, de coût i, et de même pour $D(0, j)$. Pour $i, j \geq 1$, la dernière colonne d'un alignement de $u_1 \ldots u_i$ et $v_1 \ldots v_j$ est de l'une des trois formes $(u_i, \text{-})$, $(\text{-}, v_j)$ ou (u_i, v_j). En la retirant, on obtient un alignement respectivement de $(u_1 \ldots u_{i-1}, v_1 \ldots v_j)$, de $(u_1 \ldots u_i, v_1 \ldots v_{j-1})$ ou de $(u_1 \ldots u_{i-1}, v_1 \ldots v_{j-1})$, dont le coût est inférieur de respectivement 1, 1 et δ. Le coût de tout alignement est donc supérieur ou égal au membre de droite. Réciproquement, en ajoutant l'une de ces trois colonnes à un alignement optimal du sous-problème correspondant, on obtient un alignement dont le coût est le terme correspondant du minimum.

Voici le tableau des $D(i, j)$ pour $u =$ `chien` (lignes) et $v =$ `chat` (colonnes) :

	ε	c	h	a	t
ε	0	1	2	3	4
c	1	0	1	2	3
h	2	1	0	1	2
i	3	2	1	1	2
e	4	3	2	2	2
n	5	4	3	3	3

On lit $d($`chien`, `chat`$) = 3$ dans la dernière case.

Tabulation. Chaque valeur dépend de la case au-dessus, de celle à gauche et de celle en diagonale : on remplit le tableau ligne par ligne, de gauche à droite.

OCaml

```
let distance_edition u v =
  let n = String.length u and m = String.length v in
  (* d.(i).(j) : distance d'édition de u[0..i-1] à v[0..j-1] *)
  let d = Array.make_matrix (n + 1) (m + 1) 0 in
  for i = 0 to n do d.(i).(0) <- i done;
  for j = 0 to m do d.(0).(j) <- j done;
  for i = 1 to n do
    for j = 1 to m do
      let c = if u.[i-1] = v.[j-1] then 0 else 1 in
      d.(i).(j) <- min (min (d.(i-1).(j) + 1) (d.(i).(j-1) + 1))
                       (d.(i-1).(j-1) + c)
    done
  done;
  d.(n).(m)
```

C

```c
#include <stdlib.h>
#include <string.h>

int min3(int a, int b, int c)
{
    int m = a < b ? a : b;
    return m < c ? m : c;
}

int distance_edition(const char *u, const char *v)
{
    int n = strlen(u), m = strlen(v);
    /* d[i * (m+1) + j] : distance d'édition de u[0..i-1] à v[0..j-1] */
    int *d = malloc(sizeof(int) * (n + 1) * (m + 1));
    for (int i = 0; i <= n; i++)
        for (int j = 0; j <= m; j++)
        {
            if (i == 0)
                d[j] = j;
            else if (j == 0)
                d[i * (m + 1)] = i;
            else
            {
                int c = u[i - 1] == v[j - 1] ? 0 : 1;
                d[i * (m + 1) + j] = min3(d[(i - 1) * (m + 1) + j] + 1,
                                          d[i * (m + 1) + j - 1] + 1,
                                          d[(i - 1) * (m + 1) + j - 1] + c);
            }
        }
    int res = d[n * (m + 1) + m];
    free(d);
    return res;
}
```

Python

```python
def distance_edition(u, v):
    n, m = len(u), len(v)
    # d[i][j] : distance d'édition de u[0..i-1] à v[0..j-1]
    d = [[0] * (m + 1) for _ in range(n + 1)]
    for i in range(n + 1):
        d[i][0] = i
    for j in range(m + 1):
        d[0][j] = j
    for i in range(1, n + 1):
        for j in range(1, m + 1):
            c = 0 if u[i-1] == v[j-1] else 1
            d[i][j] = min(d[i-1][j] + 1, d[i][j-1] + 1, d[i-1][j-1] + c)
    return d[n][m]
```

La complexité temporelle et spatiale est en $O(nm)$. Pour obtenir la suite d'opérations, on remonte le tableau depuis (n, m) en cherchant à chaque étape un terme du minimum qui réalise $D(i, j)$. Si on ne veut que la distance, il suffit de garder deux lignes du tableau, ce qui donne une complexité spatiale en $O(\min(n, m))$ quitte à échanger u et v.

40.3.6 Plus longue sous-séquence palindromique

Exercice

On se donne une chaîne de caractères et on cherche la plus longue chaîne extraite qui soit un palindrome.

Par exemple, dans `abracadabra` il y a `aradara` comme plus long palindrome extrait.

Déterminer une plus longue sous-séquence palindromique d'une chaîne de caractères.

40.4 Travaux pratiques

- Programmation dynamique en Python : somme de sous-ensembles et distance d'édition

✍ Pour d'autres exemples rédigés (ordonnancement de tâches pondérées, plus longue sous-suite commune), voir aussi la page Algorithmique exacte.

TRAVAUX PRATIQUES 41

Programmation dynamique - TP 2 (Python)

41.1 Programmation dynamique - TP 2

Ce TP applique la méthode du chapitre sur la programmation dynamique : récurrence naïve, mémoïsation, tabulation puis reconstruction.

41.1.1 Subset-Sum

On s'intéresse ici au problème SUBSET-SUM : étant donné un ensemble d'entiers $v_0, ..., v_{n-1} \in \mathbb{N}^*$ et un entier $T \in \mathbb{N}$ cible, existe-t-il $\{i_1, ..., i_k\} \subset [\![0, n-1]\!]$ tel que $T = v_{i_1} + ... + v_{i_k}$? Autrement dit, existe-t-il un sous-ensemble des valeurs de somme T ?

On va considérer les valeurs dans un tableau `values` et pour représenter un sous-ensemble, un tableau `subset` de même taille de booléens (`True` ou `False`) qui représentera l'ensemble des `values[i]` pour lesquels `subset[i]` vaut `True`.

Question 1

Écrire une fonction `get_subset` telle que `get_subset(values, subset)` renvoie la liste des éléments du sous-ensemble.

Corrigé

```
def get_subset(values, subset):
    s = []
    for i in range(len(values)):
        if subset[i]:
            s.append(values[i])
    return s
```

Tests :

```
assert(get_subset([1,2,3,4,5],
    [True,False,True,True,False]) == [1,3,4])
assert(get_subset([1,2], [False,False]) == [])
assert(get_subset([1,2], [True,True]) == [1,2])
"OK"
```

Résolution par backtracking

On va résoudre récursivement ce problème en énumérant tous les sous-ensembles.

Recherche (i, S, T) :

- si $T = 0$, on renvoie le sous-ensemble S ;
- si $i \geq n$, on échoue ;
- si $v_i \leq T$, on lance Recherche $(i+1, S \cup \{v_i\}, T - v_i)$:
  - si cela a réussi, on renvoie ce sous-ensemble ;

- on renvoie le résultat de Recherche $(i+1, S, T)$.

Il suffit donc de lancer Recherche $(0, \emptyset, T)$ pour essayer d'obtenir une décomposition.

Question 2

Écrire une fonction `subsetsum_bt` telle que `subsetsum_bt(values, i, subset, target)` renvoie un sous-ensemble de `values` permettant d'obtenir `target` s'il en existe un et renvoie `None` sinon.

Corrigé

```
def subsetsum_bt(values, i, subset, target):
    if target == 0:
        return get_subset(values, subset)
    if i >= len(values):
        return None
    if values[i] <= target and not subset[i]:
        subset[i] = True
        v = subsetsum_bt(values, i+1, subset, target-values[i])
        if v != None:
            return v
        subset[i] = False
    return subsetsum_bt(values, i+1, subset, target)
```

On peut tester cette fonction ainsi avec 0 et $\emptyset$ (un tableau de `False`).

```
subsetsum_bt([1,3,5,7],0,[False,False,False,False],10)
```

```
subsetsum_bt([1,3,5,7],0,[False,False,False,False],15)
```

```
subsetsum_bt([1,3,5,7],0,[False,False,False,False],2)
```

On obtient respectivement `[3, 7]`, `[3, 5, 7]` et `None`.

Tests :

```
assert(subsetsum_bt([1,3,5],0,[False,False,False],4) == [1,3])
assert(subsetsum_bt([1,3,5,7],0,[False,False,False,False],10) == [3,7])
"OK"
```

On remarque que cet algorithme va énumérer tous les sous-ensembles dans le pire cas, donc il est en $O(2^n)$ pour n valeurs. On ne connaît pas d'algorithme de complexité polynomiale pour ce problème : il s'agit d'un problème dit NP-complet, et en trouver un permettrait de résoudre le fameux problème P = NP.

On va essayer d'utiliser la programmation dynamique ici.

Problème de décision

On va se restreindre à une version plus simple ici où se pose la question de savoir si c'est possible ou non d'atteindre T mais sans chercher explicitement le sous-ensemble qui le réalise.

Question 3

Écrire une fonction `subsetsum_rec_aux` telle que `subsetsum_rec_aux(values, target, i)` renvoie un booléen indiquant si le nombre `target` peut être atteint par un sous-ensemble de valeurs dans `values` d'indice $\geq i$.

Corrigé

```
def subsetsum_rec_aux(values, target, i):
    if target == 0:
        return True
    if i >= len(values):
```

```
        return False
    b = subsetsum_rec_aux(values, target, i+1)
    if values[i] <= target:
        b = b or subsetsum_rec_aux(values, target-values[i], i+1)
    return b
```

Tests :

```
assert(subsetsum_rec_aux([1,3,4], 0, 0))
assert(subsetsum_rec_aux([1,3,4], 4, 0))
assert(subsetsum_rec_aux([1,3,4], 1, 0))
assert(subsetsum_rec_aux([1,3,4], 8, 0))
assert(not subsetsum_rec_aux([1,3,4], 2, 0))
assert(not subsetsum_rec_aux([1,3,4], 9, 0))
"OK"
```

Question 4

Écrire une fonction `subsetsum_rec` telle que `subsetsum_rec(values, target)` renvoie le booléen indiquant si `target` peut être écrit comme la somme d'un sous-ensemble des valeurs dans `values` en utilisant `subsetsum_rec_aux`.

Corrigé

```
def subsetsum_rec(values, target):
    return subsetsum_rec_aux(values, target, 0)
```

Tests :

```
assert(subsetsum_rec([1,3,4], 0))
assert(subsetsum_rec([1,3,4], 4))
assert(subsetsum_rec([1,3,4], 1))
assert(subsetsum_rec([1,3,4], 8))
assert(not subsetsum_rec([1,3,4], 2))
assert(not subsetsum_rec([1,3,4], 9))
"OK"
```

Question 5

Écrire une fonction `summable` qui renvoie la liste croissante des nombres qu'on peut atteindre pour un ensemble de valeurs donné.

Corrigé

```
def summable(values):
    s = sum(values)
    return [ k for k in range(s+1) if subsetsum_rec(values, k) ]
```

Tests :

```
assert(summable([1,3,4]) == [0,1,3,4,5,7,8])
assert(summable([1,2,6,18]) == [0, 1, 2, 3, 6, 7, 8,
    9, 18, 19, 20, 21, 24, 25, 26, 27])
"OK"
```

Mémoïsation

Question 6

Écrire une fonction `subsetsum_memo_aux` telle que `subsetsum_memo_aux(values, target, i, cache)` renvoie un booléen indiquant si le nombre `target` peut être atteint par un sous-ensemble de valeurs dans `values` d'indice $\geq i$ en utilisant le dictionnaire `cache` pour mémoriser les valeurs des couples `(target, i)`.

Corrigé

```
def subsetsum_memo_aux(values, target, i, cache):
    if (target, i) in cache:
        return cache[(target, i)]
    if target == 0:
        b = True
    elif i >= len(values):
        b = False
    else:
        b = subsetsum_memo_aux(values, target, i+1, cache)
        if values[i] <= target:
            b = b or subsetsum_memo_aux(values, target-values[i], i+1, cache)
    cache[(target, i)] = b
    return b
```

Tests :

```
assert(subsetsum_memo_aux([1,3,4], 0, 0, {}))
assert(subsetsum_memo_aux([1,3,4], 4, 0, {}))
assert(subsetsum_memo_aux([1,3,4], 1, 0, {}))
assert(subsetsum_memo_aux([1,3,4], 8, 0, {}))
assert(not subsetsum_memo_aux([1,3,4], 2, 0, {}))
assert(not subsetsum_memo_aux([1,3,4], 9, 0, {}))
assert(not subsetsum_memo_aux([ 2*i for i in range(100) ], 265, 0, {}))
"OK"
```

Question 7

Écrire une fonction `subsetsum_memo` telle que `subsetsum_memo(values, target)` renvoie le booléen indiquant si `target` peut être écrit comme la somme d'un sous-ensemble des valeurs dans `values` en utilisant `subsetsum_memo_aux`.

Corrigé

```
def subsetsum_memo(values, target):
    return subsetsum_memo_aux(values, target, 0, {})
```

Tests :

```
assert(subsetsum_memo([1,3,4], 0))
assert(subsetsum_memo([1,3,4], 4))
assert(subsetsum_memo([1,3,4], 1))
assert(subsetsum_memo([1,3,4], 8))
assert(not subsetsum_memo([1,3,4], 2))
assert(not subsetsum_memo([1,3,4], 9))
assert(not subsetsum_memo([ 2*i for i in range(100) ], 265))
"OK"
```

Tabulation

Pour résoudre ce problème par tabulation, on va devoir utiliser des matrices.

Question 8

Écrire une fonction `matrix` telle que `matrix(n, m, v)` renvoie une matrice, i.e. un tableau de tableaux, de n tableaux de m valeurs v.

Par exemple, `matrix(3, 2, 0)` doit renvoyer `[[0,0],[0,0],[0,0]]`.

Corrigé

```
def matrix(n,m,v):
    mat = []
    for i in range(n):
        mat.append([v] * m)
    return mat
```

Tests :

```
assert(matrix(2,3,0) == [[0,0,0],[0,0,0]])
assert(matrix(3,2,0) == [[0,0],[0,0],[0,0]])
assert(matrix(1,1,'a') == [['a']])
m = matrix(2,2,0)
m[0][0] = 1
assert(m == [[1,0],[0,0]])
"OK"
```

Question 9

Écrire une fonction `subsetsum_tab` telle que `subsetsum_tab(values, target)` renvoie le booléen indiquant si `target` peut être écrit comme la somme d'un sous-ensemble des valeurs dans `values` en utilisant une matrice de booléens pour tabuler.

Corrigé

Ici `b[i][t]` indique si `t` peut être atteint avec les valeurs d'indice $\geq i$.

```
def subsetsum_tab(values, target):
    n = len(values)
    b = matrix(n+1, target+1, False)
    b[n][0] = True
    for i in reversed(range(n)):
        b[i][0] = True
        for t in range(1, target+1):
            b[i][t] = b[i+1][t]
            if t >= values[i]:
                b[i][t] = b[i][t] or b[i+1][t-values[i]]
    return b[0][target]
```

Tests :

```
assert(subsetsum_tab([1,3,4], 0))
assert(subsetsum_tab([1,3,4], 4))
assert(subsetsum_tab([1,3,4], 1))
assert(subsetsum_tab([1,3,4], 8))
assert(not subsetsum_tab([1,3,4], 2))
assert(not subsetsum_tab([1,3,4], 9))
assert(not subsetsum_tab([ 2*i for i in range(100) ], 265))
"OK"
```

Si on a n valeurs et une cible T, l'algorithme tabulé utilise $O(nT)$ espace mémoire pour la table et la même complexité en temps pour la remplir. On a donc un algorithme qui est polynomial en n pour résoudre SUBSET-SUM, P = NP et je vais aller empocher le million de dollars de l'institut Clay.

Question 10

So long 👋

ou pas ?

Corrigé

Ce n'est **pas** une complexité polynomiale en la taille de l'entrée : l'entier T est représenté sur environ $\log_2 T$ bits, et T est donc exponentiel en sa taille. Avec des valeurs et une cible de l'ordre de 2^n, la table a $n2^n$ cases. On dit que l'algorithme est **pseudo-polynomial** : il n'est efficace que lorsque T reste petit.

Questions annexes

Question 11

À l'aide de ces fonctions, déterminer le plus petit nombre qui ne soit pas la somme d'un sous-ensemble de l'ensemble de valeurs $\{1, 2, 3, 5, 7, ..., 97\}$ constitué des nombres premiers plus petits que 100 et de 1.

Note : comme l'ensemble de valeurs ne change pas, il est sûrement plus efficace de conserver un unique cache ou de reprendre la tabulation pour conserver la table.

Question 12

Reprendre la fonction `subsetsum_tab` pour qu'elle renvoie un sous-ensemble dans le cas où la cible peut être écrite comme une somme, et `None` sinon.

Note : on pourra utiliser une matrice indiquant à l'indice (i, t) si on utilise la i-ème valeur dans la décomposition de t.

Corrigé

On reconstruit en partant de $(0, T)$: si `dec[i][v]` est vrai, on prend `values[i]` et on continue avec `v - values[i]`. On maintient ainsi l'invariant « `b[i][v]` est vrai », ce qui garantit qu'on termine avec `v` nul.

```
def subsetsum_tab(values, target):
    n = len(values)
    b = matrix(n+1, target+1, False)
    dec = matrix(n+1, target+1, False)
    b[n][0] = True
    for i in reversed(range(n)):
        b[i][0] = True
        for t in range(1, target+1):
            b[i][t] = b[i+1][t]
            if t >= values[i]:
                b[i][t] = b[i][t] or b[i+1][t-values[i]]
                if b[i+1][t-values[i]]:
                    dec[i][t] = True
    if not b[0][target]:
        return None
    v = target
    subset = []
    for i in range(n):
        if dec[i][v]:
            subset.append(values[i])
            v = v-values[i]
    return subset
```

Question 13

Si la liste de valeurs est en ordre croissant, que peut-on dire du cardinal du sous-ensemble renvoyé ?

41.1.2 Distance d'édition

Étant donné deux chaînes de caractères A et B de longueurs respectives n et m, on veut transformer la chaîne A en la chaîne B en effectuant un minimum d'opérations parmi celles-ci :

- supprimer un caractère n'importe où ;
- insérer un nouveau caractère n'importe où ;
- changer un caractère en n'importe quel autre.

Ce nombre d'opérations est appelé la distance d'édition de A à B.

Par exemple : chien → chen → chan → chat

Question 14

Appliquer la méthodologie de la programmation dynamique pour calculer cette distance efficacement.

Question 15

En déduire une fonction renvoyant une suite de chaînes de longueur minimale pour passer d'une chaîne à une autre.

Exemple : pour `chien` et `chat` on obtiendra `['chien', 'chen', 'chan', 'chat']`.

CHAPITRE 42

Algorithmique des textes

DANS CE CHAPITRE

42.1 Recherche dans un texte
42.2 Compression
42.3 Problèmes supplémentaires

Source image : justgrims, https://www.flickr.com/photos/notbrucelee/8016192302

> **Note** **Roadmap** :
>
> - Des exercices.
> - Les extensions à la fin.

Sources

- *Algorithms* Robert Sedgewick, Kevin Wayne
- *Éléments d'algorithmique* D. Beauquier, J. Berstel, Ph. Chrétienne
- *125 Problems in Text Algorithms with Solutions* Maxime Crochemore, Thierry Lecroq, Wojciech Rytter

42.1 Recherche dans un texte

42.1.1 Principe de la recherche

On s'intéresse ici au problème suivant :

Problème 42.1 — : RechercheTexte

- Entrées :
  - une chaîne de caractères s sur l'alphabet Σ
  - une autre chaîne de caractères m sur ce même alphabet, appelée *motif*, de longueur plus petite que celle de s
- Sortie : un résultat partiel correspondant à l'indice de la première occurrence du motif dans la chaîne s'il est présent.

La différence fondamentale entre ce problème et celui de la recherche d'un sous-tableau dans un tableau est le fait qu'on considère un alphabet fini et dont le nombre d'éléments est le plus souvent négligeable par rapport à la taille des chaînes de caractères. Cela permet d'effectuer des optimisations qui ne sont pas sans rappeler les tris linéaires comme le tri par comptage.

On parle alors d'algorithmique du texte pour désigner des algorithmes tirant partie de cette contrainte sur les données. La plupart des algorithmes que l'on présente peuvent ainsi s'adapter aisément au cas de tableaux dont les éléments sont pris dans un ensemble fini de petit cardinal.

Avant d'entamer ce chapitre, remarquons qu'il existe, outre l'alphabet usuel, trois alphabets très importants :

- celui des caractères ASCII usuels
- celui contenant les deux éléments 0 et 1, ce qui permet de travailler sur des recherche en binaire.
- et enfin, très important pour la biologie, l'alphabet à quatre lettres A, T, G et C correspondant aux bases d'un brin d'ADN et qui ouvre la porte à beaucoup d'applications en bio-informatique.

> **Note** Il y aura sûrement des applications bio-info dans la partie programmation dynamique, faire le lien ici.

42.1.2 Algorithme naïf en force brute

Une solution naïve consiste à parcourir chaque position de s afin de tester si le motif est présent à partir de cette position.

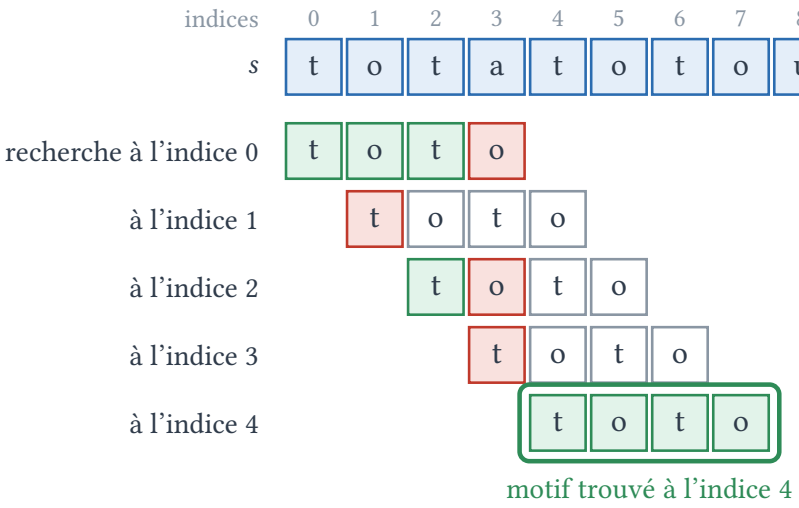


Figure 42.1 — Recherche naïve de toto : à chaque indice, comparaisons réussies (vert) jusqu'au premier échec (rouge).

Cela donne l'implémentation assez directe suivante :

OCaml

```
exception Trouve of int
exception PasDeMotif

(* renvoie un booléen indiquant si m est présent
 * dans s à l'indice i *)
let cherche_motif (m:string) (s:string) (i:int) : bool =
    let p = String.length m in
    try
        for j = 0 to p-1 do
            if s.[i+j] <> m.[j]
            then raise PasDeMotif
        done;
        true
    with PasDeMotif -> false

let recherche_naive m s =
    let n = String.length s in
    let p = String.length m in
    try
        for i = 0 to n-p do
            if cherche_motif m s i
            then raise (Trouve i)
        done;
        None
    with Trouve i -> Some i
```

C

```c
/* recherche_naive(m,s) recherche le motif m dans la chaine
 * s et renvoie l'indice de la première occurrence s'il est présent
 * ou -1 sinon */
int recherche_naive(const char *m, const char *s)
{
    int n = strlen(s);
    int p = strlen(m);

    for (int i = 0; i <= n-p; i++)
    {
        int j;
        for (j = 0; j < p; j++)
        {
            if (s[i+j] != m[j])
                break;
        }
        if (j == p)
            return i;
    }

    return -1;
}
```

Python

```python
def recherche_naive(motif, chaine):
    '''Recherche le motif dans la chaine et renvoie l'indice de
    la première occurrence du motif s'il est présent ou None sinon.'''
    n, p = len(chaine), len(motif)
    for i in range(n-p+1):
        present = True
        for j in range(p):
            if chaine[i+j] != motif[j]:
                present = False
                break
        if present:
            return i
    return None
```



Remarque 42.2 Ces implémentations se comportent correctement dans les deux cas limites exclus par l'énoncé du problème : si $p > n$, la boucle sur i est vide et le motif est déclaré absent ; si $p = 0$, le motif vide est trouvé dès l'indice 0, ce qui est cohérent puisque le mot vide est préfixe de tout mot. Toutes les recherches de ce chapitre suivent cette même convention.

La complexité temporelle en pire cas de cet algorithme correspond au maximum de comparaisons. On peut naturellement en déduire par majoration une borne en $O(np)$ mais on peut remarquer qu'il est assez difficile d'obtenir un exemple concret, ce qui fait penser que ce pire cas est *rare*.

Remarque 42.3 Considérons la chaîne $s = aa...a = a^n$ qui contient n fois la lettre a et le motif $m = a^{p-1}b$ qui contient $p-1$ a et finit par un b. Dans l'algorithme, on va donc à chaque étape de la première boucle effectuer p itérations dans la seconde avant de se rendre compte que le motif n'est pas présent en comparant b et a. On a donc exactement $(n-p+1)p$ comparaisons, ce qui est un $\Theta(np)$ dès que $p \leq n/2$ par exemple, et on retombe ainsi sur la complexité $O(np)$ pour ces exemples.

Ce qui va se passer dans une application usuelle de cet algorithme, c'est qu'au bout d'une ou deux comparaisons, on pourra invalider la position et passer à la suivante. On va alors avoir une complexité en $O(n+p)$ en considérant en plus la validation du motif dans le cas où il est présent. Ici $p \leq n$ donc $O(n+p) = O(n)$ mais c'est important de garder en tête cette complexité en $O(n+p)$ qu'on retrouvera car elle s'appliquera à des algorithmes où on effectue un prétraitement sur le motif pour l'appliquer ensuite sur plusieurs chaînes.

42.1.3 Algorithme de Boyer-Moore

Dans un premier temps, on va présenter la variante usuelle de cet algorithme appelée algorithme de Boyer-Moore-Horspool. On présentera ensuite l'algorithme de Boyer-Moore en tant que tel.

Principe de Boyer-Moore-Horspool

Le principe de l'algorithme de Boyer-Moore-Horspool est d'effectuer une recherche du motif comme précédemment mais en partant de la fin. On va alors tenter de trouver des suffixes de plus en plus grands du motif. Si on trouve ainsi le motif, on renvoie la position. Sinon, c'est qu'on a lu dans s un mot de la forme xm' où m' est un suffixe strict de m mais xm' n'en est pas un. Si x n'est pas présent dans m, alors on peut relancer la recherche juste après x dans s. Si x est présent dans m, on peut relancer la recherche en alignant ce caractère avec sa position la plus à droite dans m.

> **Remarque 42.4** Il faut tenir compte différemment du dernier caractère du motif, car il n'est pas utile de le réaligner. On considère alors, quand elle existe, l'occurrence précédente de ce caractère.

On obtient ainsi une stratégie de saut qui en cas d'échec relance la recherche plus loin.

Voici un premier exemple où on effectue une recherche de *abaa* dans le mot *aaababbbaa*. Cette stratégie a permis d'éviter une recherche inutile à partir de l'indice 1.

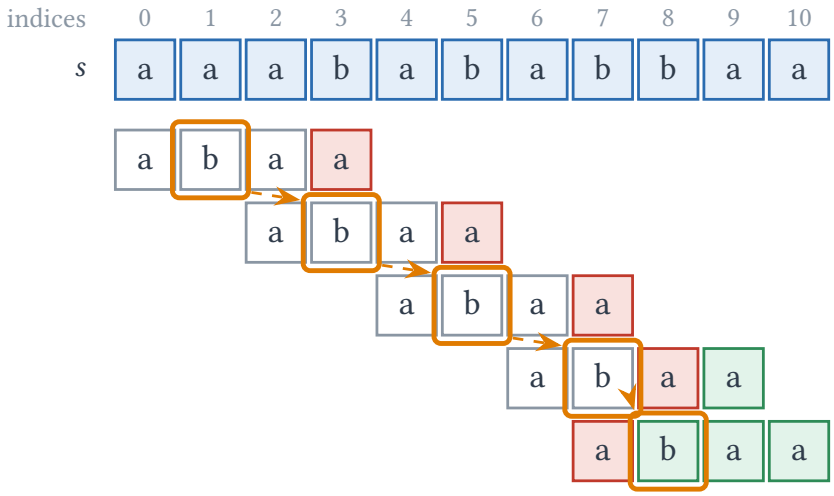


Figure 42.2 — Recherche de abaa : après chaque échec (rouge), on réaligne le b du motif (orange) sur le caractère lu.

Implémentation par table de saut

Pour réaliser ces sauts, on construit une table `droite` indexée par Σ et telle que `droite[c]` indique l'indice de l'occurrence la plus à droite dans le motif m du caractère c, en ignorant le dernier caractère du motif.

Ainsi, dans l'exemple précédent du motif `abaa`, on obtient la table suivante :

c	"a"	"b"	"c"	...
droite[c]	2	1	$\emptyset$	...

On a indiqué ici $\emptyset$ quand un caractère de Σ n'est pas présent dans le motif, car il peut être présent dans s.

Cette table contient donc de l'ordre de $|\Sigma|$ éléments. On peut la réaliser par un tableau direct de taille $|\Sigma|$ étant donné un ordre d'énumération. On peut aussi la réaliser par un dictionnaire, ce qui est plus économe en espace si le motif contient peu de lettres différentes. On a choisi ici, pour des raisons pédagogiques, de considérer la numérotation ASCII naturelle associée aux caractères pour indexer cette table.

OCaml

```
let taille_alphabet = 256

let calcule_droite (motif:string) : int option array =
    (* Calcule le tableau droite associé au motif *)
    let droite = Array.make taille_alphabet None in
    let p = String.length motif in
    for i = 0 to p-2 do
```

```
        let j = p-2-i in
        let c = motif.[j] in
        if droite.(Char.code c) = None
        then droite.(Char.code c) <- Some j
    done;
    droite
```

C

```c
int taille_alphabet = 256; // on pourrait passer par un define

/* Calcule le tableau droite associé au motif
 * le tableau renvoyé a été alloué, il devra être libéré après utilisation */
int *calcule_droite(char *motif)
{
    int *droite = malloc(sizeof(int) * taille_alphabet);
    int p = strlen(motif);

    memset(droite, -1, sizeof(int) * taille_alphabet);

    for (int i = 0; i <= p-2; i++)
    {
        int j = p-2-i;
        unsigned char c = motif[j];
        if (droite[c] < 0)
            droite[c] = j;
    }

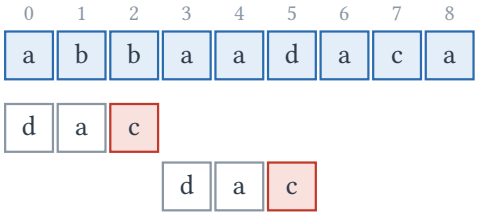
    return droite;
}
```

Python

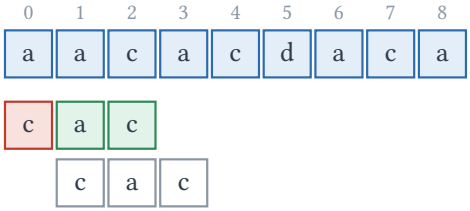
```python
taille_alphabet = 256
def calcule_droite(motif):
    '''Calcule le tableau droite associé au motif'''
    droite = [ None ] * taille_alphabet
    p = len(motif)
    for i in range(p-1):
        j = p-2-i # indice en partant de la fin
        c = motif[j]
        if droite[ord(c)] is None:
            droite[ord(c)] = j
    return droite
```

Afin d'implémenter l'algorithme lui-même, il est nécessaire de faire des calculs élémentaires mais précis pour déterminer le saut à effectuer. Si à la position $i + j$ on a un échec après avoir lu le caractère `c` où `droite[c]` contient la valeur k :

- Si $k = \emptyset$, c'est que le motif ne pourra jamais être trouvé tant que ce caractère `c` sera dans la zone comparée. On relance donc la recherche juste après, à l'indice $i + j + 1$.



- Si $k \geq j$, cela signifie que `c` est présent plus à droite dans le motif, donc aligner cette occurrence ne permettrait pas d'avancer la recherche. Rien ne nous permet de savoir si `c` est présent ou non ailleurs dans le motif, on relance alors prudemment la recherche en $i + 1$.



- Sinon, on veut aligner ce c avec le caractère correspondant du motif, si on relance à l'indice i', on souhaite ainsi avoir $i' + k = i + j$ donc $i' = i + j - k$.

OCaml

```
exception Difference
exception Trouve of int

let recherche_BMH (motif:string) (droite:int option array)
    (chaine:string) : int option =
    let n = String.length chaine in
    let p = String.length motif in
    let i = ref 0 in
    try
        while !i <= n-p do
            try
                for j = p-1 downto 0 do
                    if chaine.[!i+j] <> motif.[j]
                    then begin
                        let dec =  match droite.(Char.code chaine.[!i+j]) with
                                | None -> j+1
                                | Some k when k < j -> j-k
                                | _ -> 1 in
                        i := !i + dec;
                        raise Difference
                    end
                done;
                raise (Trouve !i)
            with Difference -> ()
        done;
        None
    with Trouve k -> Some k
```

C

```
/* cherche motif dans chaine en utilisant la table de saut précalculée
 * droite. Renvoie l'indice de la première occurrence ou -1 s'il n'est pas
 * présent */
int recherche_BMH(char *motif, int *droite, char *chaine)
{
    int n = strlen(chaine);
    int p = strlen(motif);

    for(int i = 0; i <= n-p; )
    {
        bool present = true;
        for (int j = p-1; j >= 0; j--)
        {
            if (chaine[i+j] != motif[j])
            {
                int k = droite[(unsigned char) chaine[i+j]];
                present = false;
                if (k < 0)
                    i = i + j + 1;
                else if (k < j)
                    i = i + j - k;
                else
                    i = i + 1;
                break;
```

```
            }
        }
        if (present)
            return i;
    }

    return -1;
}
```

Python

```python
def recherche_BMH(motif, droite, chaine):
    '''Cherche motif dans chaine en utilisant la table de saut
    précalculée droite'''
    n, p = len(chaine), len(motif)
    i = 0
    while i <= n-p:
        present = True
        for j in reversed(range(p)):
            if chaine[i+j] != motif[j]:
                present = False
                k = droite[ord(chaine[i+j])]
                if k is None:
                    i = i + j + 1
                elif k < j:
                    i = i + j - k
                else:
                    i = i + 1
                break
        if present:
            return i
    return None
```

42.1.3.2.1 Correction

Tout d'abord, remarquons que la terminaison ne pose pas de questions dans la mesure où le nouvel indice auquel on relance la recherche est toujours strictement plus grand que le précédent.

Au sujet de la correction, il suffit de s'assurer que les indices écartés correspondent nécessairement à des recherches infructueuses. Seuls les sauts d'au moins deux indices nécessitent une preuve. Notons $m = m_0 \ldots m_{p-1}$ et supposons qu'à l'indice i, on ait un échec en comparant le caractère $c = s_{i+j}$ avec m_j et que l'on relance la recherche à l'indice $i + j - k$ où k est la valeur de `droite[c]` si $k < j$, et $k = -1$ si `droite[c]` $= \emptyset$ (ce qui correspond bien au saut en $i + j + 1$).

Soit i' tel que $i < i' < i + j - k$. Une occurrence du motif à l'indice i' placerait le caractère $c = s_{i+j}$ en face du caractère $m_{j'}$ du motif où $j' = i + j - i'$ vérifie $k < j' < j \leq p - 1$. Or, par définition de `droite`, c n'apparaît pas parmi $m_{k+1}, \ldots, m_{p-2}$, donc $m_{j'} \neq c$ et la recherche à l'indice i' échoue nécessairement.

42.1.3.2.2 Complexité

Tout d'abord, on remarque que la table de saut se construit en $O(\max(|\, m \,|, |\, \Sigma \,|))$ pour un motif m sur un alphabet Σ.

Sans chercher à rentrer dans les détails, on peut raisonnablement penser **si l'alphabet contient assez de caractères** que les motifs auront peu de répétitions et qu'ainsi, les sauts seront presque toujours maximaux, ce qui permet d'obtenir de l'ordre de $\frac{n}{p}$ comparaisons où n est la longueur de la chaîne et p la longueur du motif.

Cependant, en pire cas, cet algorithme n'est pas meilleur que le précédent. Pour s'en convaincre, on va considérer un exemple proche de celui introduit pour l'algorithme naïf. Si on cherche ba^{p-1} dans a^n à l'indice i, il est nécessaire d'attendre de comparer au caractère b pour constater un échec et devoir relancer l'algorithme à l'indice $i + 1$. On va donc faire ici aussi $(n - p + 1)p$ comparaisons, soit $\Theta(np)$ dès que $p \leq n/2$.

La complexité temporelle en pire cas de Boyer-Moore-Horspool est donc de $O(np)$, même si, en pratique, elle est sous-linéaire.

> **Remarque 42.5** Si l'alphabet contient peu de caractères, ce qui est le cas en particulier du binaire, il y a de grandes chances qu'on soit dans ce cas pire cas. Ainsi, Boyer-Moore-Horspool n'est pas adapté pour ce type de texte.

Principe de Boyer-Moore

Considérons le cas suivant de l'algorithme précédent : on cherche abbcabc dans cbacbbcabc.

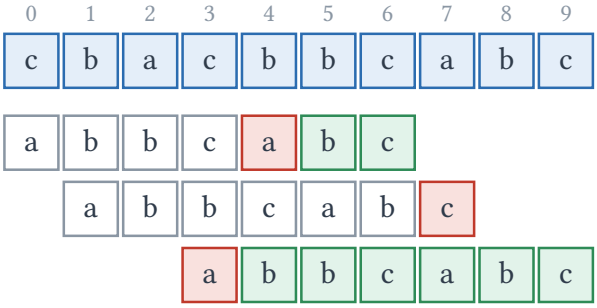


Figure 42.3 — Horspool pour abbcabc dans cbacbbcabc : des sauts très courts.

On remarque qu'en raison du fonctionnement de cet algorithme, on est forcé de faire de tous petits sauts et on est ramené à l'algorithme naïf. Cependant, après la première étape, on sait qu'on a lu dans la chaîne le suffixe `bc` du motif, précédé d'un caractère `b` en sorte que `bbc` ne soit pas un suffixe du motif (dans le motif, `bc` est précédé de `a`).

Il y a un autre endroit dans le motif où on peut trouver `*bc` avec `*` un autre caractère que `a`. On pourrait donc relancer la recherche en alignant cette occurrence de `bc` avec celle qu'on vient de lire. Cela revient à sauter directement à la dernière étape dans cet exemple :

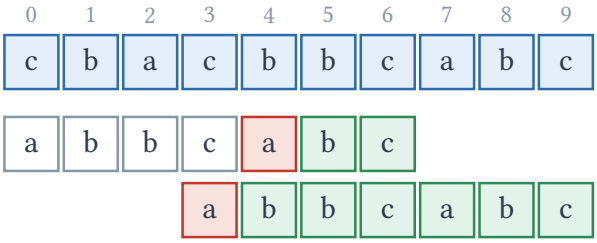


Figure 42.4 — Saut obtenu en alignant l'autre occurrence de bc dans le motif.

Pour pouvoir réaliser ce décalage, il est nécessaire de calculer une nouvelle table en parcourant le motif pour identifier de telles apparitions de suffixes.

On peut aller plus loin en considérant également le plus long préfixe du motif qui soit un suffixe du suffixe considéré. Par exemple, pour le motif `bcabc` on remarque que `bc` étant un préfixe, on peut effectuer un saut comme dans l'exemple suivant :

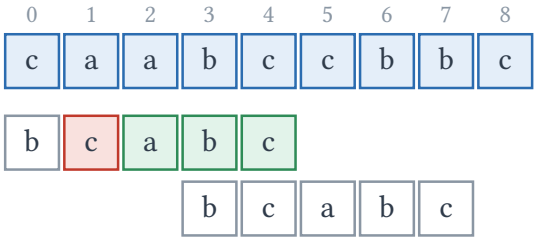


Figure 42.5 — Saut alignant le préfixe bc du motif bcabc sur le suffixe bc lu.

42.1.3.3.1 Table des bons suffixes

> **Note** Tout cela sera redéfini proprement plus tard dans le chapitre sur les langages. Je laisse cette partie en attendant pour que la présentation soit complète.

Il est nécessaire d'introduire des définitions précises pour formaliser la stratégie qu'on vient de présenter. Dans le contexte des langages, on parle plus souvent de mot que de chaîne de caractères, qui sont un type de données permettant de les représenter. Un mot sur l'alphabet Σ est donc une suite finie $a_1 \dots a_n$ de lettres dans l'alphabet. On note μ l'unique mot vide, c'est-à-dire ne contenant aucune lettre. L'ensemble des mots sur Σ est noté Σ^*. Si u et v sont des mots, uv est le mot obtenu par concaténation.

> **Remarque 42.6** Σ^* muni de cette loi de composition a une structure proche de l'ensemble des entiers naturels $\mathbb{N}$ muni de l'addition :
>
> - on a : $\forall u, v, w \in \Sigma^*, u(vw) = (uv)w = uvw$, on dit que la loi est associative ;
> - elle possède un élément neutre μ : $\forall u \in \Sigma^*, \mu u = u\mu = u$.
>
> On dit alors que Σ^* est un **monoïde**. Cette structure très simple est cruciale en informatique.

Définition 42.7

Soit $u, v \in \Sigma^*$, on dit que v est :

- un **suffixe** de u s'il existe $w \in \Sigma^*$ tel que $u = wv$
- un **préfixe** de u s'il existe $w \in \Sigma^*$ tel que $u = vw$

Lorsque $w \neq \mu$, on parle de suffixe ou de préfixe **propre**.

On dit que v est un **bord** de u lorsque v est suffixe et préfixe propre de u.

Exemple 42.8

Soit $u = abacaba$. $abac$ est un préfixe de u, $caba$ un suffixe et aba un bord.

Définition 42.9

Soit $x = x_1 \dots x_n$ et u, v deux suffixes **distincts** de x. On dit que u et v sont des suffixes **disjoints** quand on est dans l'un des cas suivants :

- $u = x$
- $v = x$
- $u \neq x, v \neq x$ et $x_{|x|-|u|} \neq x_{|x|-|v|}$.

Des suffixes disjoints sont donc des suffixes précédés par des lettres différentes dans x. On définit de même la notion de préfixes disjoints.

On considère un motif $x = x_0 \dots x_{n-1}$ et on va reprendre, en la précisant, la description précédente. Ce faisant, on va construire une table `bonsuffixe` appelée la table des **bons suffixes** du motif x et telle que, pour $i \in [\![0, n-1]\!]$, `bonsuffixe[i]` donne le nombre de positions dont on doit décaler le motif vers la droite pour relancer la recherche après la lecture du suffixe $x_{i+1} \dots x_{n-1}$.

Supposons qu'on vient de lire avec succès un suffixe propre u. Ainsi $x = x_0 \dots x_i u$ et on vient de lire dans la chaîne où on effectue la recherche au avec $a \neq x_i$.

- Soit il existe un autre suffixe buv de x où $b \neq x_i$ et alors on appelle bon suffixe pour u un tel suffixe de longueur minimale et on pose alors `bonsuffixe[i]`$= |v|$
- Sinon, on cherche v de longueur minimale tel que x soit un suffixe de uv et on pose également `bonsuffixe[i]` $= |v|$.

On remarque que si x est suffixe de uv et qu'on a également buv' suffixe de x, alors $|uv| = |u| + |v| \geq |x| \geq |buv'| \geq |u| + |v'|$ donc $|v| \geq |v'|$ ce qui permet de considérer le plus petit v sur l'ensemble des cas.

42.1.3.3.2 Table des suffixes

Afin de calculer efficacement `bonsuffixe` on va commencer par calculer la table des suffixes du motif, il s'agit de la table `suffixe` où `suffixe[i]` contient la longueur du plus long suffixe de x de la forme $x_j...x_i$. Ainsi, si on note S_i les suffixes de cette forme, on a :

$$\texttt{suffixe}\,[i] = \begin{cases} 0 & \text{si } S_i = \emptyset \\ \max\{|\,s\,|\,|\,s \in S_i\} & \text{sinon} \end{cases}$$

Nécessairement, `suffixe[n-1]=n` car x convient.

Exemple 42.10

Pour $x = bcabc$ on a :

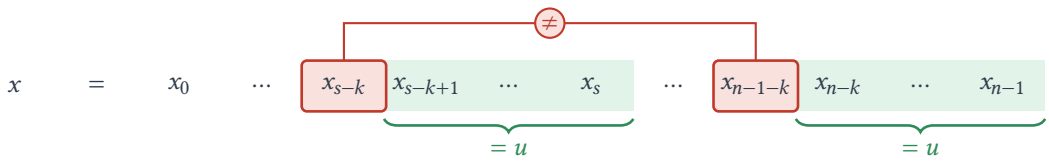
i	0	1	2	3	4
suffixe[i]	0	2	0	0	5

et pour $x = abbabba$:

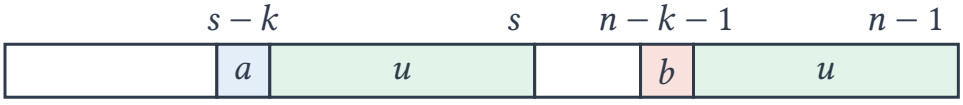
i	0	1	2	3	4	5	6
suffixe[i]	1	0	0	4	0	0	7

Il est possible de construire `suffixe` avec un simple parcours linéaire en tirant partie de l'information déjà calculée. Pour cela, on va remplir `suffixe` de droite à gauche.

À tout moment, on va conserver le meilleur suffixe rencontré, c'est-à-dire celui pour lequel on est allé le plus loin à gauche avant d'avoir un échec de comparaison. On note s la position la plus à droite de ce suffixe et k sa longueur, il s'agit donc de $u = x_{s-k+1}...x_s$ et il y a eu un échec de comparaison en x_{s-k}. Par définition de `suffixe` on a `suffixe[s]`$= k$. Le mot x s'écrit alors :



Ce qu'on peut représenter schématiquement ainsi :



Maintenant, on considère la position $s - i$ où $s > s - i > s - k$, c'est-à-dire $0 < i < k$, cela signifie qu'on cherche un suffixe depuis une position interne au mot u de gauche. Le point clé permettant d'obtenir un algorithme linéaire est de remarquer que la situation est la même que dans le mot u de droite, à la position $n - 1 - i$. Or, comme on procède de droite à gauche, on a déjà calculé la valeur correspondante `suffixe[n-1-i]`. Là, on a deux cas :

- soit quand on a cherché le plus grand suffixe finissant à la position $n - 1 - i$, on s'est heurté à une erreur de comparaison à la position $n - 1 - k$, celle de b. Dans ce cas, on a `suffixe[n-1-i]`$= k - i$ et on peut regarder, en partant de la position $s - k$, si on peut prolonger le suffixe finissant à la position $s - i$.



Pour effectuer ce prolongement, il suffit de comparer, caractère par caractère, vers la gauche en partant de la position $s-k$. On aboutira alors à un nouveau suffixe finissant en $s-i$, de longueur l, qui devient le suffixe allé le plus loin à gauche.



Remarquons qu'il est possible que $s-i-l = s-k$. C'est-à-dire que même si a ne permet pas de prolonger le suffixe déduit de la position $n-1-i$, on considère tout de même que la nouvelle position de référence est $s-i$. On en déduit également la valeur `suffixe[s-i]`$=l$.

- soit `suffixe[n-1-i]`$= t \neq k-i$ et alors
  - soit $t < k-i$, on a alors pour ce suffixe un échec à l'intérieur du u de droite, à la position $n-1-i-t$, et ce même échec se produit à l'intérieur du u de gauche, à la position $s-i-t$: `suffixe[s-i]`$= t$.
  - soit $t > k-i$, donc le suffixe finissant en $n-1-i$ se prolonge au-delà de u et le caractère b en position $n-1-k$ est égal à $x_{n-1-(k-i)}$. Pour prolonger le suffixe finissant en $s-i$ au-delà de la longueur $k-i$, il faudrait donc avoir $x_{s-k} = x_{n-1-(k-i)} = b$, or, c'est impossible car $x_{s-k} = a \neq b$. Ainsi, le suffixe est limité par u : `suffixe[s-i]`$= k-i$.

Il reste à traiter le cas où $s-i \leq s-k$, ce qui revient à considérer qu'on a dépassé le précédent suffixe pouvant apporter une information. On procède donc naïvement pour trouver le plus grand suffixe depuis cette position.

On en déduit l'implémentation suivante :

OCaml

```
let calcule_suffixe (x:string) : int array =
    (* Prend en entrée un mot x et renvoie son tableau de suffixe *)
    let n = String.length x in
    let suffixe = Array.make n (-1) in
    if n > 0 then suffixe.(n-1) <- n;
    let plus_a_gauche = ref (n-1) in
    let depart = ref (-1) in

    for j = n-2 downto 0 do
        if !plus_a_gauche < j
            && suffixe.(n-1- !depart+j) <> j - !plus_a_gauche
        then suffixe.(j) <- min suffixe.(n-1- !depart+j) (j - !plus_a_gauche)
        else begin
            plus_a_gauche := min j !plus_a_gauche;
            depart := j;
            while !plus_a_gauche >= 0
                && x.[!plus_a_gauche] = x.[n-1-j + !plus_a_gauche] do
                plus_a_gauche := !plus_a_gauche - 1
            done;
            suffixe.(j) <- !depart - !plus_a_gauche
        end
    done;
    suffixe
```

C

```c
/* Calcule le tableau suffixe associé au mot x
 * le tableau renvoyé a été alloué, il devra être libéré après utilisation */
int *calcule_suffixe(char *x)
{
    int n = strlen(x);
    int *suffixe = malloc(sizeof(int) * n);
    int plus_a_gauche = n-1;
    int depart = -1;

    memset(suffixe, -1, sizeof(int) * n);
    if (n > 0)
        suffixe[n-1] = n;

    for (int j = n-2; j >= 0; j--)
    {
        if (plus_a_gauche < j
            && suffixe[n-1-depart+j] != j-plus_a_gauche)
            suffixe[j] = MIN(suffixe[n-1-depart+j], j-plus_a_gauche);
        else {
            plus_a_gauche = MIN(plus_a_gauche, j);
            depart = j;
            while (plus_a_gauche >=0
                    && x[plus_a_gauche] == x[n-1-j+plus_a_gauche])
                plus_a_gauche--;
            suffixe[j] = depart - plus_a_gauche;
        }
    }

    return suffixe;
}
```

Python

```python
def calcule_suffixe(x):
    '''Prend en entrée un mot x et renvoie son tableau de suffixe'''
    n = len(x)
    suffixe = [ None ] * n
    if n > 0:
        suffixe[n-1] = n
    plus_a_gauche = n-1
    depart = None

    for j in reversed(range(0,n-1)):
        if plus_a_gauche < j and suffixe[n-1-depart+j] != j-plus_a_gauche:
            suffixe[j] = min(suffixe[n-1-depart+j],j-plus_a_gauche)
        else:
            plus_a_gauche = min(plus_a_gauche, j)
            depart = j
            while plus_a_gauche >= 0 \
                    and x[plus_a_gauche] == x[n-1-j+plus_a_gauche]:
                plus_a_gauche = plus_a_gauche - 1
            suffixe[j] = depart - plus_a_gauche

    return suffixe
```

Remarque 42.11 Cette implémentation est optimisée par rapport à la description précédente en calculant directement sans introduire i ou k, et en fusionnant deux cas qui reviennent à dupliquer du code.

On donne ici le code maladroit qui correspond à la traduction exacte de la description précédente :

OCaml

```ocaml
let calcule_suffixe_rep (x:string) : int array =
    (* Prend en entrée un mot x et renvoie son tableau de suffixe *)
    let n = String.length x in
    let suffixe = Array.make n 0 in
    if n > 0 then suffixe.(n-1) <- n;
    let plus_a_gauche = ref (n-1) in
```

```
    let depart = ref (-1) in

    for j = n-2 downto 0 do
        if !plus_a_gauche < j
        then begin
            (* on a j = depart - i avec *)
            let i = !depart - j in
            (* et plus_a_gauche = depart - k avec *)
            let k = !depart - !plus_a_gauche in
            if suffixe.(n-1-i) <> k-i
            then suffixe.(j) <- min suffixe.(n-1-i) (k-i)
            else begin
                depart := j;
                while !plus_a_gauche >= 0
                    && x.[!plus_a_gauche] = x.[n-1-j + !plus_a_gauche] do
                    plus_a_gauche := !plus_a_gauche - 1
                done;
                suffixe.(j) <- !depart - !plus_a_gauche
            end
        end else begin
            plus_a_gauche := j;
            depart := j;
            while !plus_a_gauche >= 0
                && x.[!plus_a_gauche] = x.[n-1-j + !plus_a_gauche] do
                plus_a_gauche := !plus_a_gauche - 1
            done;
            suffixe.(j) <- !depart - !plus_a_gauche
        end
    done;
    suffixe
```

C

```
/* Calcule le tableau suffixe associé au mot x
 * le tableau renvoyé a été alloué, il devra être libéré après utilisation */
int *calcule_suffixe_rep(char *x)
{
    int n = strlen(x);
    int *suffixe = malloc(sizeof(int) * n);
    int plus_a_gauche = n-1;
    int depart = -1;

    memset(suffixe, -1, sizeof(int) * n);
    if (n > 0)
        suffixe[n-1] = n;

    for (int j = n-2; j >= 0; j--)
    {
        if (plus_a_gauche < j)
        {
            // on a j = depart - i avec
            int i = depart - j;
            // et plus_a_gauche = depart -k avec
            int k = depart - plus_a_gauche;

            if (suffixe[n-1-i] != k-i)
                suffixe[j] = MIN(suffixe[n-1-i], k-i);
            else {
                depart = j;
                while (plus_a_gauche >=0
                        && x[plus_a_gauche] == x[n-1-j+plus_a_gauche])
                    plus_a_gauche--;
                suffixe[j] = depart - plus_a_gauche;
            }
        } else {
            plus_a_gauche = depart = j;
            while (plus_a_gauche >=0
                    && x[plus_a_gauche] == x[n-1-j+plus_a_gauche])
```

```
                plus_a_gauche--;
            suffixe[j] = depart - plus_a_gauche;
        }
    }

    return suffixe;
}
```

Python

```
def calcule_suffixe_rep(x):
    '''Prend en entrée un mot x et renvoie son tableau de suffixe'''
    n = len(x)
    suffixe = [ None ] * n
    if n > 0:
        suffixe[n-1] = n
    plus_a_gauche = n-1
    depart = None

    for j in reversed(range(0,n-1)):
        if plus_a_gauche < j:
            # on a j = depart - i avec
            i = depart - j
            # et plus_a_gauche = depart - k avec
            k = depart - plus_a_gauche
            if suffixe[n-1-i] != k-i:
                suffixe[j] = min(suffixe[n-1-i],k-i)
            else:
                depart = j
                while plus_a_gauche >= 0 \
                        and x[plus_a_gauche] == x[n-1-j+plus_a_gauche]:
                    plus_a_gauche = plus_a_gauche - 1
                suffixe[j] = depart - plus_a_gauche
        else:
            plus_a_gauche = depart = j
            while plus_a_gauche >= 0 \
                    and x[plus_a_gauche] == x[n-1-j+plus_a_gauche]:
                plus_a_gauche = plus_a_gauche - 1
            suffixe[j] = depart - plus_a_gauche

    return suffixe
```

On remarque que dans ce code, `plus_a_gauche` ne peut que diminuer, on effectue donc au plus n itérations dans la boucle `while` pour tout l'algorithme. Donc, en considérant la boucle `for`, on effectue au plus $2n$ comparaisons de caractères : au plus une pour chaque itération de la boucle `for` pour voir si on entre dans le `while`, puis en tout au plus n avant de sortir du `while`.

L'algorithme qu'on a obtenu est bien linéaire en $|\, x\, |$.

42.1.3.3.3 Obtention de `bonsuffixe` à partir de `suffixe`

On reprend maintenant le calcul de `bonsuffixe[i]` dans le mot $x = x_0 \ldots x_{n-1}$.

On cherche à obtenir des suffixes de la forme buv de x où $b \neq x_i$ et $u = x_{i+1} \ldots x_{n-1}$ est un suffixe de x, de longueur $n-1-i$. Mais si, pour $k < n-1$, `suffixe[k]`$= n-1-i$, cela signifie que le facteur de x de longueur $n-1-i$ finissant en position k est exactement u et qu'il est soit préfixe, soit précédé d'une lettre différente de x_i, sinon $n-1-i$ ne serait pas maximal. Le mot v correspondant est alors $x_{k+1} \ldots x_{n-1}$, de longueur $n-1-k$.

On a donc, quand cet ensemble n'est pas vide,

$$\begin{aligned} \texttt{bonsuffixe}\,[i] &= \min\{n-1-k \mid k < n-1 \ \text{ et } \ \texttt{suffixe}\,[k] = n-1-i\} \\ &= n-1-\max\{k \mid k < n-1 \ \text{ et } \ \texttt{suffixe}\,[k] = n-1-i\} \end{aligned}$$

On remarque qu'on peut ainsi faire croître k de 0 à $n-2$ et poser :

$$\texttt{bonsuffixe}\,[n-1-\texttt{suffixe}\,[k]] = n-1-k$$

On aura alors naturellement, à la fin de la boucle, la valeur minimale placée en dernier.

Reste à considérer les valeurs non remplies ainsi dans le tableau `bonsuffixe`. Elles correspondent aux positions i telles qu'il n'existe pas de suffixe de la forme buv. On doit donc chercher un mot uv de longueur minimale dont x est suffixe. Mais u étant un suffixe de x, cela revient à considérer les bords de x. La table `suffixe` permet également de détecter les bords : si $x_0 ... x_k$ est un bord c'est que `suffixe[k]`$= k+1$.

Soit $k < n-1$ maximal vérifiant cette condition. Pour tout $u = x_{i+1} ... x_{n-1}$ suffixe de x, pour qu'il ait $x_0 ... x_k$ comme suffixe, il faut qu'il soit au moins aussi long que ce bord, donc que $n-1-i \geq k+1 \Leftrightarrow i < n-1-k$. Dans ce cas, x est alors suffixe de uv où $v = x_{k+1} ... x_{n-1}$ donc $|\, v \,| = n-1-k$. Les k plus petits ne pourront alors que faire augmenter $|\, v \,|$, on peut ainsi poser `bonsuffixe[i]`$= n-1-k$.

On en déduit un remplissage en parcourant les k dans l'ordre décroissant de $n-2$ à 0, tout en maintenant l'indice i de la prochaine valeur à remplir dans `bonsuffixe`. Dès qu'on détecte un bord, on place $n-1-k$ jusqu'à ce que $i \geq n-1-k$.

En sortie de boucle, il est possible que $i < n$ donc qu'il reste des valeurs à remplir. On remarque dans ce cas là que pour que x soit un suffixe de uv il faut que $v = x$. On a donc pour ces valeurs restantes `bonsuffixe[i]` $= n$.

Comme ce second cas est toujours plus long que le premier quand les deux se produisent en i, on implémente successivement les remplissages de sorte à obtenir la valeur minimum. On en déduit le programme suivant :

OCaml

```
let calcule_bonsuffixe (x:string) : int array =
    (* Prend en entrée un mot x et renvoie son tableau de suffixe *)
    let n = String.length x in
    let suffixe = calcule_suffixe x in
    let bonsuffixe = Array.make n n in

    let suivant = ref 0 in
    for k = n-2 downto 0 do
        if suffixe.(k) = k+1 (* c'est un bord *)
        then begin
            for i = !suivant to n-2-k do
                bonsuffixe.(i) <- n-1-k
            done;
            suivant := n-1-k
        end
    done;

    for k = 0 to n-2 do
        bonsuffixe.(n-1-suffixe.(k)) <- n-1-k
    done;

    bonsuffixe
```

C

```
/* Calcule le tableau suffixe associé au mot x
 * le tableau renvoyé a été alloué, il devra être libéré après utilisation */
int *calcule_bonsuffixe(char *x)
{
    int n = strlen(x);
    int *suffixe = calcule_suffixe(x);
    int *bonsuffixe = malloc(sizeof(int) * n);
    int suivant = 0;

    for (int i = 0; i < n; i++)
        bonsuffixe[i] = n;

    for (int k = n-2; k >= 0; k--)
    {
        if (suffixe[k] == k+1) // bord
        {
```

```
            for (int i = suivant; i < n-1-k; i++)
                bonsuffixe[i] = n-1-k;
            suivant = n-1-k;
        }
    }

    for (int k = 0; k < n-1; k++)
        bonsuffixe[n-1-suffixe[k]] = n-1-k;

    free(suffixe);

    return bonsuffixe;
}
```

Python

```
def calcule_bonsuffixe(x):
    '''Prend en entrée un mot x et renvoie son tableau de suffixe'''
    n = len(x)
    suffixe = calcule_suffixe(x)
    bonsuffixe = [ n ] * n

    suivant = 0
    for k in reversed(range(0,n-1)):
        if suffixe[k] == k+1: # c'est un bord
            for i in range(suivant,n-1-k):
                bonsuffixe[i] = n-1-k
            suivant = n-1-k

    for k in range(0,n-1):
        bonsuffixe[n-1-suffixe[k]] = n-1-k

    return bonsuffixe
```

Il est facile de constater que cet algorithme est de complexité temporelle linéaire en $|x|$.

42.1.3.3.4 Algorithme de Boyer-Moore

On incorpore naturellement la table précédente à l'algorithme de Boyer-Moore en choisissant le meilleur décalage entre cette table et la stratégie précédente.

OCaml

```
let recherche_BM (motif:string)
    (droite:int option array) (bonsuffixe:int array)
    (chaine:string) : int option =
    let n = String.length chaine in
    let p = String.length motif in
    let i = ref 0 in
    try
        while !i <= n-p do
            try
                for j = p-1 downto 0 do
                    if chaine.[!i+j] <> motif.[j]
                    then begin
                        let dec =  match droite.(Char.code chaine.[!i+j]) with
                                | None -> j+1
                                | Some k when k < j -> j-k
                                | _ -> 1 in
                        i := !i + max dec bonsuffixe.(j);
                        raise Difference
                    end
                done;
                raise (Trouve !i)
            with Difference -> ()
        done;
        None
    with Trouve k -> Some k
```

C

```c
/* cherche motif dans chaine en utilisant les tables de saut précalculées
 * droite et bonsuffixe. Renvoie l'indice de la première occurrence ou -1 s'il n'est pas
 * présent */
int recherche_BM(char *motif, int *droite, int *bonsuffixe, char *chaine)
{
    int n = strlen(chaine);
    int p = strlen(motif);

    for(int i = 0; i <= n-p; )
    {
        bool present = true;
        for (int j = p-1; j >= 0; j--)
        {
            if (chaine[i+j] != motif[j])
            {
                int k = droite[(unsigned char) chaine[i+j]];
                int dec = 1;
                present = false;
                if (k < 0)
                    dec = j + 1;
                else if (k < j)
                    dec = j - k;
                i = i + MAX(dec, bonsuffixe[j]);
                break;
            }
        }
        if (present)
            return i;
    }

    return -1;
}
```

Python

```python
def recherche_BM(motif, droite, bonsuffixe, chaine):
    '''Cherche motif dans chaine en utilisant les tables de sauts
    droite et bonsuffixe'''
    n, p = len(chaine), len(motif)
    i = 0
    while i <= n-p:
        present = True
        for j in reversed(range(p)):
            if chaine[i+j] != motif[j]:
                present = False
                k = droite[ord(chaine[i+j])]
                if k is None:
                    dec = j + 1
                elif k < j:
                    dec = j - k
                else:
                    dec = 1
                dec = max(dec, bonsuffixe[j])
                i = i + dec
                break
        if present:
            return i
    return None
```

Supposons que le motif est de longueur p, que la chaîne dans laquelle on recherche est de longueur n et que la taille de l'alphabet est une constante indépendante des entrées. La première partie de l'algorithme consiste à construire les tables de sauts, comme on l'a vu, elle est en complexité en temps et en espace en pire cas en $O(p)$.

On admet que l'algorithme Boyer-Moore complet, étant donné les deux tables de saut et d'autres modifications mineures non présentées ici, est en complexité temporelle en pire cas en $O(n)$.

Il est assez raisonnable de penser que soit $p \leq n$ quand on effectue une recherche, soit on compte chercher un même motif dans plusieurs textes et on réutilise ainsi les tables de sauts. Il n'est donc pas forcément très pertinent de parler de la complexité globale de l'algorithme, mais lorsqu'on le fait, on dit qu'elle est en $O(p+n)$. On rappelle ici le rôle de l'addition dans les complexités qui fait référence à la succession de deux traitements, un en $O(p)$ suivi d'un en $O(n)$.

42.1.4 Algorithme de Rabin-Karp

Principe

L'algorithme de Rabin-Karp est un algorithme de recherche d'un motif dans un texte qui utilise une notion d'empreinte pour déterminer, en temps constant, si il est probable que la position actuelle corresponde à une occurrence du motif.

Pour cela, si on cherche un motif de longueur p sur l'alphabet Σ, on considère une **fonction de hachage** $h : \Sigma^p \to X$. Les éléments de l'ensemble X sont appelés des empreintes et on suppose que l'égalité entre deux empreintes se vérifie en temps constant contrairement à l'égalité dans Σ^p qui se vérifie en $O(p)$ dans le pire des cas. Le plus souvent, on choisit pour X un type entier machine.

> **Note** Sûrement mettre ici des renvois vers la partie portant le plus sur la notion de fonction de hachage pour la définition la plus complète.

Bien qu'il soit normalement aussi coûteux de calculer l'image par h d'une sous-chaîne de longueur p que de tester l'égalité entre cette sous-chaîne et le motif, le point essentiel de l'algorithme de Rabin-Karp est d'utiliser une fonction de hachage permettant un calcul incrémental en temps constant :

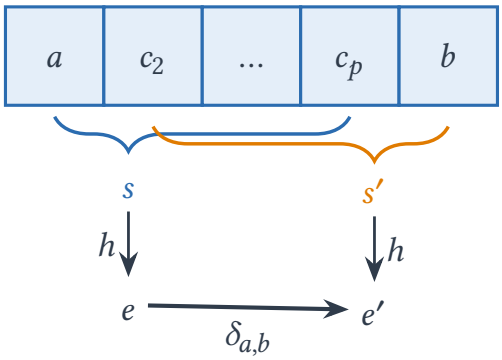


Figure 42.6 — L'empreinte de la fenêtre suivante s' se calcule à partir de celle de s.

Ici, on considère donc, pour $a, b \in \Sigma$, une fonction de mise à jour $\delta_{a,b} : X \to X$ telle que pour tout $c_2, ..., c_p \in \Sigma$ on ait $\delta_{a,b}(h(ac_2...c_p)) = h(c_2...c_pb)$.

L'algorithme de Rabin-Karp procède alors ainsi pour chercher m de longueur p dans la chaîne $s = c_0...c_{n-1}$ où $n \geq p$:

- calcul de $e_m = h(m)$ et $e = h(c_0..c_{p-1})$.
- Pour i allant de 0 à $n - p$:
  - Si $e_m = e$, on renvoie un succès pour la recherche à la position i si $m = c_i...c_{i+p-1}$
  - si $i < n - p$ on met à jour l'empreinte $e \leftarrow \delta_{c_i,c_{i+p}}(e)$.

La complexité temporelle liée à la gestion des empreintes est donc en $O(n + p) = O(n)$ car $n \geq p$. Par contre, pour calculer la complexité liée à la recherche $m = c_i...c_{i+p-1}$, il est nécessaire d'estimer la proportion de faux positifs, c'est-à-dire de positions i telles que $e_m = e$ mais $m \neq c_i...c_{i+p-1}$. On va voir dans la partie suivante qu'on peut supposer qu'elle est négligeable, ce qui permet de considérer que l'algorithme de Rabin-Karp est linéaire.

Choix d'une fonction de hachage

Réaliser une bonne fonction de hachage est une question très complexe qui dépasse le cadre du cours d'informatique de MPI. Cependant, il est possible de réaliser ici une fonction de hachage répondant aux contraintes de Rabin-Karp assez facilement.

Pour cela, on considère que les caractères sont des entiers compris entre 0 et 255, ce qui correspond au type des caractères non signés sur un octet. On peut alors identifier une chaîne de longueur p avec un nombre entre 0 et $r^p - 1$ où $r = 2^8$, on note ainsi

$$P(c_0 \dots c_{p-1}) = \sum_{i=0}^{p-1} c_i r^{p-1-i} = c_0 r^{p-1} + c_1 r^{p-2} + \dots + c_{p-1}$$

On considère de plus un entier premier q et on pose $h(s) = P(s) \bmod q$ c'est-à-dire le reste de $P(s)$ dans la division euclidienne par q. On peut ainsi définir $\delta_{a,b}(e) = (r(e - ar^{p-1}) + b) \bmod q$.

Si on précalcule $r^{p-1} \bmod q$, il suffit d'un nombre d'opérations constant, et indépendant de p, pour calculer la nouvelle empreinte à l'aide de $\delta_{a,b}$.

Le point essentiel est alors de déterminer un nombre premier q tel qu'il soit peu probable d'obtenir des faux positifs. Une analyse mathématique permet d'affirmer que chaque élément de $[\![0, q-1]\!]$ a de l'ordre de $\frac{r^p}{q}$ antécédents par h. Ainsi, si on choisit deux chaînes aléatoirement dans Σ^p, il y aura collision avec probabilité proche de $\frac{1}{q}$. En considérant q proche de la taille maximale pour le type entier considéré, on minimise donc cette probabilité.

Remarque 42.12 On peut également s'intéresser à des nombres q pour lesquels le modulo soit rapide à calculer. Un exemple classique est $q = 2^{31} - 1$ car on peut déduire le reste de a modulo q de l'écriture de a en base 2^{31}. En effet, si $a = \sum_{k=0}^{n} a_k 2^{31k}$ avec $0 \leq a_k < 2^{31}$, comme $2^{31} - 1 \mid 2^{31k} - 1$ pour $k \geq 1$, on a $2^{31k} \equiv 1\ [q]$ et ainsi $a \equiv \sum_{k=0}^{n} a_k\ [q]$. On remarque que chaque chiffre s'obtient par un simple décalage et un masquage sur 31 bits : $a_k = (a >> 31k) q$ car $q = 2^{31} - 1$ s'écrit avec 31 bits à 1. La somme $s = \sum_{k=0}^{n} a_k$ est congrue à a modulo q et strictement plus petite que a dès que $a \geq 2^{31}$: on recommence donc avec s tant que $s > q$. Enfin, si $s \leq q$, on a $s \bmod q = s$ quand $s < q$ et $s \bmod q = 0$ quand $s = q$.

On obtient alors le programme suivant :

OCaml

```
let rec fastmod a =
    let s = ref 0 in
    let x = ref a in
    let q = 0x7fffffff in
    while !x > 0 do
        s := !s + (!x land q);
        x := !x lsr 31
    done;
    if !s > q
    then fastmod !s
    else if !s = q
    then 0
    else !s
```

C

```
int64_t fastmod(int64_t a)
{
    int64_t s = 0;
    const int64_t q = 0x7fffffff;

    while (a > 0)
    {
        s = s + (a & q);
        a = a >> 31;
    }
```

```
    if (s > q)
        return fastmod(s);
    if (s == q)
        return 0;
    return s;
}
```

Python

```
def fastmod(a):
    s = 0
    q = 0x7fffffff
    while a > 0:
        s = s + (a & q)
        a = a >> 31
    if s > q:
        return fastmod(s)
    elif s == q:
        return 0

    return s
```

Le programme suivant implémente naïvement les calculs de h et de $\delta_{a,b}$, ainsi que celui de $r^k \bmod q$ par exponentiation rapide. Attention : en OCaml et en C, le reste `mod` (ou `%`) d'un entier négatif est négatif, or $e - ar^{p-1}$ peut l'être. Il faut donc ramener le résultat dans $[\![0, q-1]\!]$, ce qui n'est pas nécessaire en Python où `%` renvoie toujours un résultat positif quand $q > 0$.

OCaml

```
let hash r q s =
    let p = ref 1 in
    let e = ref 0 in
    for i = String.length s - 1 downto 0 do
        e := (!p * (Char.code s.[i]) + !e) mod q;
        p := (r * !p) mod q
    done;
    !e

let rec pow r k q = (* r^k mod q *)
    if k = 0 then 1
    else let x = pow ((r * r) mod q) (k / 2) q in
        if k mod 2 = 0 then x else (r * x) mod q

let delta r q rp a b e = (* rp est r^(p-1) mod q *)
    let x = (r * (e - rp * (Char.code a)) + Char.code b) mod q in
    if x < 0 then x + q else x
```

C

```
int64_t hash(int64_t r, int64_t q, char *s, int n)
{
    int64_t p = 1;
    int64_t e = 0;

    for (int i = n-1; i >= 0; i--)
    {
        e = (p * (unsigned char) s[i] + e) % q;
        p = (r * p) % q;
    }

    return e;
}


int64_t powmod(int64_t r, int k, int64_t q) // r^k mod q
{
```

```
    if (k == 0)
        return 1;
    int64_t x = powmod((r * r) % q, k / 2, q);
    return k % 2 == 0 ? x : (r * x) % q;
}

int64_t delta(int64_t r, int64_t q, int64_t rp,
        unsigned char a, unsigned char b, int64_t e)
{
    int64_t x = (r * (e - rp * a) + b) % q;
    return x < 0 ? x + q : x;
}
```

Python

```
def hash(r,q,s):
    e = 0
    p = 1
    for c in reversed(s):
        e = (ord(c) * p + e) % q
        p = (r * p) % q
    return e

def delta(r,q,rp,a,b,e):
    return (r * (e - rp * ord(a)) + ord(b)) % q
```

Implémentation

Une implémentation directe de l'algorithme de Rabin-Karp est donnée dans le programme qui suit. On se sert ici du caractère paresseux du `&&` pour n'effectuer le test coûteux d'égalité des chaînes qu'en cas d'égalité des empreintes.

OCaml

```
exception Trouve of int

let rabin_karp m s =
    let n = String.length s in
    let p = String.length m in
    if p = 0 then Some 0 (* le mot vide est présent en 0 *)
    else if p > n then None
    else begin
        let r = 256 in
        let q = 0x7fffffff in (* 2^(31)-1 *)
        let rp = pow r (p-1) q in
        let me = hash r q m in
        let e = ref (hash r q (String.sub s 0 p)) in
        try
            for i = 0 to n-p do
                if me = !e && m = String.sub s i p
                then raise (Trouve i);
                if i+p < n then e := delta r q rp s.[i] s.[i+p] !e
            done; None
        with Trouve k -> Some k
    end
```

C

```
int rabin_karp(char *m, char *s)
{
    const int64_t r = 256;
    const int64_t q = 0x7fffffff;
    const int p = strlen(m);
    const int n = strlen(s);
    if (p == 0)
        return 0; // le mot vide est présent en 0
    if (p > n)
        return -1; // sinon hash lirait au-delà de la fin de s
```

```
    const int64_t rp = powmod(r,p-1,q);
    const int64_t me = hash(r,q,m,p);
    int64_t e = hash(r,q,s,p);
    for (int i=0; i <n-p+1; i++)
    {
        if (me == e && strncmp(m,(s+i),p) == 0)
            return i;
        if (i+p < n)
            e = delta(r,q,rp,s[i],s[i+p],e);
    }
    return -1;
}
```

Python

```
def rabin_karp(m, s):
    p, n = len(m), len(s)
    if p == 0:
        return 0 # le mot vide est présent en 0
    if p > n:
        return None
    r, q = 256, 0x7fffffff
    rp = pow(r, p-1, q) # r^(p-1) mod q par exponentiation rapide
    me, e = hash(r,q,m), hash(r,q,s[:p])
    for i in range(0,n-p+1):
        if me == e and m == s[i:i+p]:
            return i
        if i+p < n:
            e = delta(r,q,rp,s[i],s[i+p],e)
    return None
```

Si on suppose qu'il est improbable d'obtenir un faux positif, il est possible de renvoyer un succès dès que les empreintes sont égales. L'avantage d'une telle version est alors d'être un algorithme sans retour sur les données. C'est-à-dire qu'il n'est pas nécessaire de garder en mémoire ou de réaccéder à un caractère.

L'algorithme originel de Rabin et Karp

Si on regarde l'article originel de Rabin et Karp décrivant cette méthode, on peut être étonné du fait que la méthode précédemment décrite était considérée comme déjà connue dans la littérature par les auteurs. En fait, ce qu'ils décrivent et annoncent comme étant novateur est l'utilisation d'un algorithme probabiliste en choisissant aléatoirement une fonction de hachage à chaque lancement de l'algorithme. En pratique, il s'agit de choisir aléatoirement un nombre premier q parmi un ensemble précalculé de nombres premiers.

L'algorithme que l'on vient de décrire a un pire cas qui est très improbable car on considère que la probabilité d'un faux positif est à peu près de $1/q$, donc moins de $5 \cdot 10^{-10}$ pour $q = 2^{31} - 1$. Le problème ici est la notion de probabilité sur les entrées : est-on certain que l'algorithme recevra une entrée choisie uniformément ? Rabin et Karp parlent d'un *adversaire intelligent* qui aurait connaissance de la fonction de hachage choisie pour produire des entrées en pire cas. On pourrait ainsi imaginer une *attaque* sur serveur effectuant une recherche avec Rabin-Karp suite à l'entrée d'un utilisateur. Un adversaire pourrait construire une entrée en pire cas et tenter de surcharger le serveur en l'effectuant de manière répétée.

Pour bien mettre en lumière ce phénomène, nous allons ici construire, dans un cas très simple de fonction de hachage, une telle chaîne problématique. Pour cela, considérons la fonction de hachage précédemment décrite dans le cas de motif de taille 2, avec Σ contenant les lettres de `a` à `z`, $r = 26$ et $q = 17$. On considère une recherche du motif `aa` dont l'empreinte est `0`, la même que celle des chaînes `ar` et `ra`. On peut donc considérer la chaîne `arar...ar` qui produira un faux positif à chaque étape.

Remarque 42.13 Détail des calculs. Ici on associe à `a` la valeur 0, ..., à `z` la valeur 25. On a donc

$$h(aa) = (0 \times 26 + 0) \bmod 17 = 0$$

$$h(ar) = (0 \times 26 + 17) \bmod 17 = 0$$

$$h(ra) = (17 \times 26 + 0) \bmod 17 = 0$$

L'empreinte reste ainsi nulle tout au long de l'algorithme de Rabin-Karp et on a un faux positif à chaque itération.

42.2 Compression

42.2.1 Principe

On s'intéresse ici à la compression parfaite d'un texte, c'est-à-dire, étant donné un alphabet fixé Σ, qu'on cherche à réaliser un couple de fonctions comp, dec : $\Sigma^* \to \Sigma^*$ telles que :

- pour tout mot $m \in \Sigma^*$, dec (comp (m)) $= m$
- pour la plupart des mots m qui correspondent aux données qu'on cherche à compresser, $|$ comp (m) $| < | m |$.

Remarque 42.14 Le fait que dec $\circ$ comp $= \mathrm{id}_{\Sigma^*}$ implique, comme on a pu le voir dans le cours de mathématique, que comp est injective : deux mots différents ont nécessairement des images distinctes.

Si A et B sont deux ensembles finis tels que $| A | < | B |$, il n'existe pas de fonction injective de A dans B. Ainsi, si on note L_n les mots de Σ^* de longueur au plus n, il ne peut exister de fonction injective de L_n dans L_m où $m < n$ (si $| \Sigma | \geq 2$, on a $| L_m | < | L_n |$).

Autrement dit : il est impossible d'espérer pouvoir compresser toutes les données de L_n. Si certains mots vont diminuer de longueur après compression, d'autres vont nécessairement augmenter.

Tout l'enjeu des algorithmes de compression parfaite est alors de diminuer les longueurs des mots qui nous intéressent. Par exemple, si on s'intéresse à des mots issus de textes en français, il est plus important d'arriver à compresser une phrase comme `"ceci est un texte"` plutôt qu'une suite de caractères non signifiante comme `"c2#$%1ajdn //@#3d!fn"`.

42.2.2 Algorithme d'Huffman

La définition et la construction de l'arbre de Huffman ont été présentées au paragraphe Construction de l'arbre de Huffman du chapitre sur les algorithmes gloutons. On va s'intéresser ici au processus complet permettant de compresser et décompresser des fichiers avec cet algorithme.

Calcul de la table d'occurrences

Par souci d'efficacité, on calcule une table d'occurrences pour l'ensemble des valeurs d'octets entre 0 et 255. Il suffit alors de parcourir le fichier pour incrémenter les valeurs correspondant aux octets lus.

OCaml

```
let table_occurrences nomdefichier =
    let fichier = open_in_bin nomdefichier in
    let occurrences = Array.make 256 0 in
    begin
     try
        while true do
            let c = input_byte fichier in
            occurrences.(c) <- occurrences.(c) + 1;
        done
      with End_of_file -> ()
    end;
    close_in fichier;
    occurrences
```

C

```
/* remplit occurrences[c] avec le nombre d'occurrences de l'octet c
 * dans le fichier nomdefichier */
void table_occurrences(const char *nomdefichier, int occurrences[256])
{
    FILE *fichier = fopen(nomdefichier, "rb");
    int c;
```

```c
    assert(fichier != NULL);
    for (int i = 0; i < 256; i++)
        occurrences[i] = 0;

    while ((c = fgetc(fichier)) != EOF)
        occurrences[c]++;

    fclose(fichier);
}
```

Python

```python
def table_occurrences(nomdefichier):
    occurrences = [ 0 ] * 256
    with open(nomdefichier, 'rb') as fichier:
        for c in fichier.read(): # les éléments d'un bytes sont des int
            occurrences[c] += 1
    return occurrences
```

On construit ensuite l'arbre de Huffman avec la fonction `construit_arbre` du chapitre sur les algorithmes gloutons.

✍ **Construction de l'arbre en C et en Python** Le chapitre sur les algorithmes gloutons ne donne que la version OCaml. Voici des versions C et Python de `construit_arbre`, qui fixent aussi la représentation des arbres utilisée dans la suite.

En C, plutôt que d'implémenter un tas, on cherche à chaque étape les deux plus petites fréquences dans un tableau d'au plus 256 arbres : le coût $O(|\Sigma|^2)$ est négligeable devant la lecture du fichier. En Python, on utilise le module `heapq` de la bibliothèque standard.

C

```c
/* Un arbre de Huffman : une feuille a deux fils NULL et une
 * étiquette c entre 0 et 255, un nœud interne a l'étiquette -1 */
struct arbre {
    int c;
    struct arbre *g;
    struct arbre *d;
};
typedef struct arbre arbre;

arbre *cree_arbre(int c, arbre *g, arbre *d)
{
    arbre *a = malloc(sizeof(arbre));
    a->c = c;
    a->g = g;
    a->d = d;
    return a;
}

arbre *feuille(int c)
{
    return cree_arbre(c, NULL, NULL);
}

arbre *noeud(arbre *g, arbre *d)
{
    return cree_arbre(-1, g, d);
}

bool est_feuille(arbre *a)
{
    return a->g == NULL;
}

void libere_arbre(arbre *a)
```

```c
{
    if (!est_feuille(a))
    {
        libere_arbre(a->g);
        libere_arbre(a->d);
    }
    free(a);
}

/* construit l'arbre de Huffman associé à la table occ,
 * renvoie NULL si toutes les occurrences sont nulles */
arbre *construit_arbre(int occ[256])
{
    arbre *arbres[256];
    int freq[256];
    int n = 0;

    for (int i = 0; i < 256; i++)
    {
        if (occ[i] > 0) // on ignore les occurrences nulles
        {
            arbres[n] = feuille(i);
            freq[n] = occ[i];
            n++;
        }
    }
    if (n == 0)
        return NULL;

    while (n > 1)
    {
        /* on place la plus petite fréquence en n-1
         * puis la suivante en n-2 */
        for (int k = 1; k <= 2; k++)
        {
            int imin = 0;
            for (int i = 1; i <= n-k; i++)
                if (freq[i] < freq[imin])
                    imin = i;
            arbre *ta = arbres[imin];
            int tf = freq[imin];
            arbres[imin] = arbres[n-k];
            freq[imin] = freq[n-k];
            arbres[n-k] = ta;
            freq[n-k] = tf;
        }
        arbres[n-2] = noeud(arbres[n-1], arbres[n-2]);
        freq[n-2] = freq[n-1] + freq[n-2];
        n--;
    }

    return arbres[0];
}
```

Python

```python
import heapq

### Une feuille est représentée par son étiquette, un entier
### entre 0 et 255, et un nœud par le couple (g, d) de ses fils.

def construit_arbre(occ):
    '''Construit l'arbre de Huffman associé à la table occ,
    lève ValueError si toutes les occurrences sont nulles.'''
    # le deuxième champ, tous distincts, évite d'avoir à
    # comparer des arbres en cas d'égalité des fréquences
    arbres = [ (occ[c], c, c) for c in range(256) if occ[c] > 0 ]
    if arbres == []:
```

```
        raise ValueError("texte vide")
    heapq.heapify(arbres)
    compteur = 256
    while len(arbres) > 1:
        fx, _, x = heapq.heappop(arbres)
        fy, _, y = heapq.heappop(arbres)
        heapq.heappush(arbres, (fx+fy, compteur, (x, y)))
        compteur += 1
    return arbres[0][2]
```

Sérialisation de l'arbre de Huffman

Afin de décompresser, il est nécessaire de connaître l'arbre de Huffman donnant le code préfixe. Pour cela, il faut stocker cet arbre dans le fichier comme une série d'octets, on parle de *sérialisation*. Cette notion sera prolongée dans le chapitre FIXME.

On choisit ici la représentation récursive `repr(a)` de l'arbre `a` définie ainsi :

- Si `a = Noeud(g,d)`, `repr(a) = 0 repr(g) repr(d)`
- Si `a = Feuille(c)`, `repr(a) = 1 c`.

Exemple 42.15

Si `a = Noeud(Feuille 42, Noeud(Feuille 16, Feuille 64))`, on obtient la suite d'octets : `0 1 42 0 1 16 1 64`.

La lecture et l'écriture de la sérialisation s'effectuent alors simplement par récurrence :

OCaml

```
let rec output_arbre f a =
    match a with
    | Noeud (x, y) -> begin
        output_byte f 0;
        output_arbre f x;
        output_arbre f y
    end
    | Feuille c -> begin
        output_byte f 1;
        output_byte f c
    end

let rec input_arbre f =
    let code = input_byte f in
    match code with
    | 0 -> let g = input_arbre f in
        let d = input_arbre f in
        Noeud(g,d)
    | _ -> Feuille (input_byte f)
```

C

```
void output_arbre(FILE *f, arbre *a)
{
    if (est_feuille(a))
    {
        fputc(1, f);
        fputc(a->c, f);
    } else {
        fputc(0, f);
        output_arbre(f, a->g);
        output_arbre(f, a->d);
    }
}

arbre *input_arbre(FILE *f)
{
```

```c
    int code = fgetc(f);
    assert(code != EOF);
    if (code == 0)
    {
        arbre *g = input_arbre(f);
        arbre *d = input_arbre(f);
        return noeud(g, d);
    }
    int c = fgetc(f);
    assert(c != EOF);
    return feuille(c);
}
```

Python

```python
def output_arbre(f, a):
    if isinstance(a, int): # feuille
        f.write(bytes([1, a]))
    else:
        g, d = a
        f.write(bytes([0]))
        output_arbre(f, g)
        output_arbre(f, d)

def input_arbre(f):
    code = f.read(1)[0]
    if code == 0:
        g = input_arbre(f)
        d = input_arbre(f)
        return (g, d)
    return f.read(1)[0]
```

Écriture dans un fichier un bit à la fois

Note À déplacer éventuellement dans une partie spécifique sur la gestion de fichiers.

Le propre de l'algorithme de Huffman est d'associer à chaque caractère un codage binaire de longueur variable. Afin de pouvoir écrire ce codage dans un fichier, il est nécessaire de grouper les bits par paquet de huit (octet en français, byte en anglais).

Ainsi, par exemple, si on a le codage suivant :

c	'a'	'b'	'c'
code(c)	0	100	101

et qu'on doit encoder `"abbaca"`, on obtient le mot binaire `010010001010` qu'on complète avec des 0 à la fin et qu'on sépare en octets : `01001000 10100000`. On obtient donc les deux octets, convertis en décimal, 72 et 160. Ce sont eux qu'on va écrire dans un fichier.

Une technique usuelle pour cela est de garder un accumulateur qui correspond à l'octet en train d'être construit ainsi que le nombre de bits qui ont été accumulés. Dès qu'on a accumulé 8 bits, on peut construire l'octet, l'écrire dans le fichier, puis réinitialiser ces variables.

Quand on rajoute un bit b à l'accumulateur, on veut passer de $acc = b_1 \ldots b_k$ à $b_1 \ldots b_k b = 2acc + b$.

On en déduit l'implémentation assez directe suivante :

OCaml

```ocaml
type out_channel_bits = {
    o_fichier : out_channel;
    mutable o_accumulateur : int;
    mutable o_bits_accumules : int
}
```

```
let open_out_bits fn =
    { o_fichier = open_out_bin fn; o_accumulateur = 0; o_bits_accumules = 0 }

let output_bit f b =
    if f.o_bits_accumules = 8
    then begin
        output_byte f.o_fichier f.o_accumulateur;
        f.o_accumulateur <- 0;
        f.o_bits_accumules <- 0
    end;
    f.o_accumulateur <- 2 * f.o_accumulateur + (if b then 1 else 0);
    f.o_bits_accumules <- 1 + f.o_bits_accumules
```

C

```
struct out_channel_bits {
    FILE *fichier;
    int accumulateur;
    int bits_accumules;
};
typedef struct out_channel_bits out_channel_bits;

/* la structure renvoyée est libérée par close_out_bits */
out_channel_bits *open_out_bits(const char *fn)
{
    out_channel_bits *f = malloc(sizeof(out_channel_bits));
    f->fichier = fopen(fn, "wb");
    assert(f->fichier != NULL);
    f->accumulateur = 0;
    f->bits_accumules = 0;
    return f;
}

void output_bit(out_channel_bits *f, bool b)
{
    if (f->bits_accumules == 8)
    {
        fputc(f->accumulateur, f->fichier);
        f->accumulateur = 0;
        f->bits_accumules = 0;
    }
    f->accumulateur = 2 * f->accumulateur + (b ? 1 : 0);
    f->bits_accumules++;
}
```

Python

```
import os

class Bitpacking:
    '''La classe Bitpacking se comporte comme un fichier Python
    tout en fournissant des méthodes spécifiques à la lecture
    et à l'écriture de bits.'''

    def __init__(self, filename, mode):
        self.file = open(filename, mode)
        self.accumulateur = 0
        self.bits_accumules = 0

        self.input = mode[0] == 'r'
        if self.input:
            self.init_input()

    def write_bit(self, b):
        if self.bits_accumules == 8:
            self.file.write(bytes([self.accumulateur]))
            self.accumulateur = 0
            self.bits_accumules = 0
```

```
        self.accumulateur = 2 * self.accumulateur \
                + (1 if b else 0)
        self.bits_accumules += 1
```

Il reste à traiter la question des zéros finaux, si l'accumulateur contient k bits au moment de la fermeture du fichier, où $0 < k < 8$, il faut ajouter $8 - k$ zéros. On appelle cela du *padding* de l'anglais pour rembourrage. Ici, cela correspond à faire un décalage binaire vers la gauche d'autant (*shift left* en anglais). Comme il sera nécessaire de se souvenir que ces zéros ne sont pas signifiants à la lecture, on rajoute un octet final contenant leur nombre $8 - k$.

On obtient alors la fonction de fermeture de fichier suivante :

OCaml

```
let close_out_bits f =
    if f.o_bits_accumules = 0
    then output_byte f.o_fichier 0
    else begin
        let padding = 8 - f.o_bits_accumules in
        output_byte f.o_fichier (Int.shift_left f.o_accumulateur padding);
        output_byte f.o_fichier padding
    end;
    close_out f.o_fichier
```

C

```
/* écrit le padding, ferme le fichier et libère la structure */
void close_out_bits(out_channel_bits *f)
{
    if (f->bits_accumules == 0)
        fputc(0, f->fichier);
    else {
        int padding = 8 - f->bits_accumules;
        fputc(f->accumulateur << padding, f->fichier);
        fputc(padding, f->fichier);
    }
    fclose(f->fichier);
    free(f);
}
```

Python

```
def close(self):
    if not self.input:
        self.close_output()
    self.file.close()

def close_output(self):
    if self.bits_accumules == 0:
        self.file.write(b'\x00')
    else:
        padding = 8 - self.bits_accumules
        self.file.write(bytes([
            self.accumulateur << padding,
            padding ]))

### pour pouvoir écrire with Bitpacking(...) as f:
### la fermeture est alors automatique en sortie de bloc
def __enter__(self):
    return self

def __exit__(self, type, value, tb):
    self.close()
```

Remarque 42.16 Une autre possibilité consiste à ajouter un entier décrivant la taille des données non compressées. C'est d'ailleurs parfois un problème avec les formats car si la taille est stockée sur 4 octets cela limite la taille d'un fichier pouvant être compressé.

Pour la lecture, on procède de même en faisant attention à deux points :

- on va lire les bits dans l'octet de la gauche vers la droite, c'est-à-dire du bit de poids le plus fort au bit de poids le plus faible. Ainsi, si l'accumulateur contient $acc = b_1 \ldots b_8$, il suffit de faire un *et* bit à bit avec `0b10000000 = 0x80 = 128` pour obtenir acc `0x80` $= b_1 0 \ldots 0$ donc un nombre qui vaut 0 si et seulement si $b_1 = 0$. Après avoir effectué cette lecture, il suffit de décaler vers la gauche en multipliant l'accumulateur par 2 : $2acc = b_2 \ldots b_8 0$.
- on doit tenir compte des zéros finaux, pour ça, on a besoin de savoir qu'on est en train de lire le dernier caractère du fichier. On calcule donc la taille du fichier à son ouverture et on teste si l'octet lu est l'avant-dernier, auquel cas on lit le dernier octet et on diminue d'autant le nombre de bits signifiants dans l'accumulateur. Symétriquement, si l'octet à lire est le dernier du fichier, c'est qu'il ne reste que l'octet de padding et donc plus aucune donnée : on signale alors la fin du fichier. Ce cas se produit à la fin de la lecture, mais aussi d'emblée quand aucun bit n'a été écrit.

On obtient alors le programme suivant pour la lecture :

OCaml

```
type in_channel_bits = {
    i_fichier : in_channel;
    mutable i_accumulateur : int;
    mutable i_bits_accumules : int;
    i_taille : int
}

let open_in_bits fn =
    let fichier = open_in_bin fn in
    { i_fichier = fichier;
      i_accumulateur = 0; i_bits_accumules = 0; i_taille = in_channel_length fichier }

let input_bit f =
    if f.i_bits_accumules = 0
    then begin
        (* s'il ne reste que l'octet de padding, il n'y a plus de données *)
        if pos_in f.i_fichier >= f.i_taille - 1
        then raise End_of_file;
        f.i_accumulateur <- input_byte f.i_fichier;
        f.i_bits_accumules <- 8;
        if pos_in f.i_fichier = f.i_taille - 1
        then begin
            let pad = input_byte f.i_fichier in
            f.i_bits_accumules <- f.i_bits_accumules - pad
        end
    end;
    let bit = f.i_accumulateur land 0x80 <> 0 in
    f.i_accumulateur <- 2 * f.i_accumulateur;
    f.i_bits_accumules <- f.i_bits_accumules - 1;
    bit

let close_in_bits f = close_in f.i_fichier
```

C

```
struct in_channel_bits {
    FILE *fichier;
    int accumulateur;
    int bits_accumules;
    long taille;
};
typedef struct in_channel_bits in_channel_bits;
```

```c
/* la structure renvoyée est libérée par close_in_bits */
in_channel_bits *open_in_bits(const char *fn)
{
    in_channel_bits *f = malloc(sizeof(in_channel_bits));
    f->fichier = fopen(fn, "rb");
    assert(f->fichier != NULL);
    fseek(f->fichier, 0, SEEK_END);
    f->taille = ftell(f->fichier);
    fseek(f->fichier, 0, SEEK_SET);
    f->accumulateur = 0;
    f->bits_accumules = 0;
    return f;
}

/* renvoie le bit lu (0 ou 1), ou -1 s'il n'y a plus de données */
int input_bit(in_channel_bits *f)
{
    if (f->bits_accumules == 0)
    {
        // s'il ne reste que l'octet de padding, il n'y a plus de données
        if (ftell(f->fichier) >= f->taille - 1)
            return -1;
        f->accumulateur = fgetc(f->fichier);
        f->bits_accumules = 8;
        if (ftell(f->fichier) == f->taille - 1)
        {
            int pad = fgetc(f->fichier);
            f->bits_accumules = f->bits_accumules - pad;
        }
    }
    int bit = (f->accumulateur & 0x80) != 0;
    f->accumulateur = 2 * f->accumulateur;
    f->bits_accumules--;
    return bit;
}

void close_in_bits(in_channel_bits *f)
{
    fclose(f->fichier);
    free(f);
}
```

Python

```python
def init_input(self):
    self.file.seek(0, os.SEEK_END)
    self.taille = self.file.tell()
    self.file.seek(0, os.SEEK_SET)

def read_bit(self):
    if self.bits_accumules == 0:
        # s'il ne reste que l'octet de padding, il n'y a plus de données
        if self.file.tell() >= self.taille - 1:
            return b''
        c = self.file.read(1)
        self.accumulateur = ord(c)
        self.bits_accumules = 8
        if self.file.tell() == self.taille - 1:
            pad = ord(self.file.read(1))
            self.bits_accumules -= pad
    bit = (self.accumulateur & 0x80) != 0
    self.accumulateur *= 2
    self.bits_accumules -= 1
    return bit
```

Compression d'un octet

Pour pouvoir compresser un octet, il est nécessaire d'obtenir le chemin qui mène jusqu'à la feuille dont il est l'étiquette dans l'arbre de Huffman. Pour cela, on commence par calculer l'ensemble des chemins de l'arbre

de Huffman sous forme d'une table à 256 entrées qui contient le chemin associé à un octet s'il est présent dans l'arbre ou un chemin vide sinon. On parlera de représentation plate de l'arbre de Huffman.

Il suffit de faire un parcours exhaustif de l'arbre (FIXME référence aux parcours d'arbres) pour réaliser cette table :

OCaml

```
let chemins a =
    let a_plat = Array.make 256 [] in
    let rec parcours a chemin =
        match a with
        | Noeud (g, d) ->
            parcours g (false::chemin);
            parcours d (true::chemin)
        | Feuille c -> a_plat.(c) <- List.rev chemin
    in
    begin match a with
    | Feuille c -> a_plat.(c) <- [false] (* feuille unique : code 0 *)
    | Noeud _ -> parcours a []
    end;
    a_plat
```

C

```
/* parcours de a, où chemin[0..profondeur-1] est le chemin depuis la racine */
void parcours(arbre *a, char *chemin, int profondeur, char *a_plat[256])
{
    if (est_feuille(a))
    {
        chemin[profondeur] = '\0';
        a_plat[a->c] = malloc(profondeur + 1);
        strcpy(a_plat[a->c], chemin);
    } else {
        chemin[profondeur] = '0';
        parcours(a->g, chemin, profondeur + 1, a_plat);
        chemin[profondeur] = '1';
        parcours(a->d, chemin, profondeur + 1, a_plat);
    }
}

/* remplit a_plat[c] avec le chemin menant à la feuille c, sous la forme
 * d'une chaîne de '0' (gauche) et de '1' (droite), ou NULL si c est absent.
 * Les chaînes sont allouées et devront être libérées. */
void chemins(arbre *a, char *a_plat[256])
{
    // au plus 256 feuilles donc une hauteur au plus 255
    char chemin[257];

    for (int c = 0; c < 256; c++)
        a_plat[c] = NULL;

    if (est_feuille(a)) // feuille unique : code 0
    {
        chemin[0] = '0';
        parcours(a, chemin, 1, a_plat);
    } else
        parcours(a, chemin, 0, a_plat);
}
```

Python

```
def chemins(a):
    a_plat = [ [] ] * 256
    def parcours(a, chemin):
        if isinstance(a, int): # feuille
            a_plat[a] = chemin
        else:
```

```
            g, d = a
            parcours(g, chemin + [False])
            parcours(d, chemin + [True])
    if isinstance(a, int): # feuille unique : code 0
        a_plat[a] = [False]
    else:
        parcours(a, [])
    return a_plat
```

Afin de compresser un octet, on va donc aller lire le chemin dans cette table puis écrire le mot binaire correspondant grâce aux fonctions d'écriture bit à bit :

OCaml

```
let compresse_byte f a_plat c =
    let rec output_bool_list l =
        match l with
        | t::q -> Bitpacking.output_bit f t; output_bool_list q
        | [] -> ()
    in output_bool_list a_plat.(c)
```

✍ **C**

```
void compresse_byte(out_channel_bits *f, char *a_plat[256], int c)
{
    for (char *b = a_plat[c]; *b != '\0'; b++)
        output_bit(f, *b == '1');
}
```

Python

```
def compresse_byte(f, a_plat, c):
    for b in a_plat[c]:
        f.write_bit(b)
```

Remarque 42.17 À chaque fois qu'on va compresser un octet, on va parcourir la liste correspondant à son chemin. Comme les chemins les plus longs sont les moins fréquents, cela ne pose pas vraiment de problèmes.

Cependant, il est possible d'optimiser cela en ne stockant pas le chemin mais la fonction d'écriture elle-même. Ainsi, on ne va plus stocker des listes de booléens mais des fonctions du type `Bitpacking.out_channel_bit -> unit` qui vont réaliser l'écriture compressée de l'octet correspondant.

Au cours du parcours de l'arbre, on maintient une fonction correspondant à l'écriture du préfixe du chemin `chemin`. Si l'appel est effectué sur un nœud, on remplace `chemin` par la fonction qui appelle `chemin` puis écrit le bit correspondant au côté gauche ou droit.

Cela correspond au programme suivant :

OCaml

```
let chemins_continuation a =
    let a_plat = Array.make 256 (fun _ -> ()) in
    let rec parcours a chemin =
        match a with
        | Noeud (g, d) ->
            parcours g (fun f -> chemin f; Bitpacking.output_bit f false);
            parcours d (fun f -> chemin f; Bitpacking.output_bit f true)
        | Feuille c -> a_plat.(c) <- chemin
    in
    begin match a with
    | Feuille c -> (* feuille unique : code 0 *)
        a_plat.(c) <- (fun f -> Bitpacking.output_bit f false)
    | Noeud _ -> parcours a (fun _ -> ())
    end;
    a_plat
```

```
let compresse_byte_continuation f a_plat c =
    a_plat.(c) f
```

Cette représentation des chemins partiels par des fonctions est très classique dans le style de programmation fonctionnelle par passage de continuations.

Décompression d'un octet

Pour décompresser un octet, il suffit de parcourir l'arbre de Huffman en lisant bit à bit le fichier compressé en descendant à gauche ou à droite selon que le bit lu soit 0 ou non. Dès qu'on arrive sur une feuille, on écrit dans le nouveau fichier le caractère correspondant.

OCaml

```
let decompresse_byte f a =
    let rec descend a =
        match a with
        | Feuille c -> c
        | Noeud(g,d) ->
            descend (if Bitpacking.input_bit f then d else g)
    in
    match a with
    | Feuille c -> (* feuille unique : on consomme son code 0 *)
        ignore (Bitpacking.input_bit f); c
    | Noeud _ -> descend a
```

C

```
/* renvoie l'octet décompressé, ou -1 s'il n'y a plus de données */
int decompresse_byte(in_channel_bits *f, arbre *a)
{
    if (est_feuille(a)) // feuille unique : on consomme son code 0
        return input_bit(f) < 0 ? -1 : a->c;

    while (!est_feuille(a))
    {
        int b = input_bit(f);
        if (b < 0)
            return -1;
        a = b ? a->d : a->g;
    }
    return a->c;
}
```

Python

```
def decompresse_byte(f, a):
    '''Renvoie l'octet décompressé ou None s'il n'y a plus de données'''
    if isinstance(a, int): # feuille unique : on consomme son code 0
        return None if f.read_bit() == b'' else a
    while not isinstance(a, int):
        b = f.read_bit()
        if b == b'':
            return None
        g, d = a
        a = d if b else g
    return a
```

Remarque 42.18 Quand le fichier ne contient qu'un seul octet distinct, l'arbre de Huffman est réduit à une feuille et le chemin qui y mène est le mot vide. Comme le rappelle le chapitre sur les algorithmes gloutons, ce code vide est optimal à condition de connaître le nombre de caractères à décoder : c'est la convention retenue dans le TP Huffman, qui mémorise la longueur du texte. Ici, le format de fichier ne stocke pas cette longueur, la fin des données n'est connue qu'à travers le padding : avec des codes vides, la décompression produirait indéfiniment le même octet sans jamais lire de bit. On convient donc d'associer

> le code `0` à une feuille unique, ce qui coûte un bit par octet, et `decompresse_byte` lit ce bit avant de renvoyer l'étiquette. Cela concerne aussi `chemins` et sa variante par continuations.

Compression et décompression de fichiers

En mettant bout à bout l'ensemble des fonctions, on obtient la fonction suivante qui réalise la compression complète d'un fichier :

OCaml

```
let compresse_fichier nom_in nom_out =
    let occ = table_occurrences nom_in in
    let a = if Array.for_all (fun x -> x = 0) occ
        then Feuille 0 (* fichier vide : arbre arbitraire *)
        else construit_arbre occ in
    let f_out = Bitpacking.open_out_bits nom_out in
    output_arbre f_out.o_fichier a;
    let a_plat = chemins a in
    let f_in = open_in_bin nom_in in
    begin
      try
        while true do
            let c = input_byte f_in in
            compresse_byte f_out a_plat c
        done
      with End_of_file -> ()
    end;
    close_in f_in;
    Bitpacking.close_out_bits f_out
```

C

```
void compresse_fichier(const char *nom_in, const char *nom_out)
{
    int occ[256];
    char *a_plat[256];
    int c;

    table_occurrences(nom_in, occ);
    arbre *a = construit_arbre(occ);
    if (a == NULL) // fichier vide : arbre arbitraire
        a = feuille(0);

    out_channel_bits *f_out = open_out_bits(nom_out);
    output_arbre(f_out->fichier, a);
    chemins(a, a_plat);

    FILE *f_in = fopen(nom_in, "rb");
    assert(f_in != NULL);
    while ((c = fgetc(f_in)) != EOF)
        compresse_byte(f_out, a_plat, c);
    fclose(f_in);
    close_out_bits(f_out);

    for (int i = 0; i < 256; i++)
        free(a_plat[i]);
    libere_arbre(a);
}
```

Python

```
def compresse_fichier(nom_in, nom_out):
    occ = table_occurrences(nom_in)
    if sum(occ) == 0:
        a = 0 # fichier vide : arbre arbitraire
    else:
        a = construit_arbre(occ)
    a_plat = chemins(a)
    with open(nom_in, 'rb') as f_in, \
```

```
            Bitpacking(nom_out, 'wb') as f_out:
        output_arbre(f_out.file, a)
        for c in f_in.read():
            compresse_byte(f_out, a_plat, c)
```

On obtient de même la fonction de décompression suivante :

OCaml

```
let decompresse_fichier nom_in nom_out =
    let f_in = Bitpacking.open_in_bits nom_in in
    let a = input_arbre f_in.i_fichier in
    let f_out = open_out_bin nom_out in
    begin
      try
          while true do
              let c = decompresse_byte f_in a in
              output_byte f_out c
          done
      with End_of_file -> ()
    end;
    Bitpacking.close_in_bits f_in;
    close_out f_out
```

C

```
void decompresse_fichier(const char *nom_in, const char *nom_out)
{
    in_channel_bits *f_in = open_in_bits(nom_in);
    arbre *a = input_arbre(f_in->fichier);
    FILE *f_out = fopen(nom_out, "wb");
    int c;

    assert(f_out != NULL);
    while ((c = decompresse_byte(f_in, a)) >= 0)
        fputc(c, f_out);

    close_in_bits(f_in);
    fclose(f_out);
    libere_arbre(a);
}
```

Python

```
def decompresse_fichier(nom_in, nom_out):
    with Bitpacking(nom_in, 'rb') as f_in, \
            open(nom_out, 'wb') as f_out:
        a = input_arbre(f_in.file)
        while True:
            c = decompresse_byte(f_in, a)
            if c is None:
                break
            f_out.write(bytes([c]))
```

Remarque 42.19 Pour un fichier vide, il n'y a pas d'arbre de Huffman : `construit_arbre` échoue. On écrit alors un arbre arbitraire réduit à une feuille, suivi d'aucun bit. Le fichier compressé fait trois octets : `1 0` pour l'arbre et l'octet final `0` indiquant l'absence de padding. À la décompression, après lecture de l'arbre, il ne reste que l'octet final, donc la première lecture de bit signale la fin du fichier et on obtient bien un fichier vide, sans cas particulier.

Exemple 42.20

En compressant ainsi l'intégrale de Proust, on passe de 7543767 octets à 4249758 octets. A titre de comparaison, l'outil unix `zip` permet d'obtenir un fichier de 2724213 octets.

Remarque 42.21 Le format de fichier présenté ici est rudimentaire. Les formats usuels sont en général plus complexes pour gérer

- l'identification : c'est-à-dire pouvoir déterminer qu'un fichier est un fichier compressé par un certain programme. Les fichiers `zip` commencent ainsi par les deux lettres `PK`.
- l'extensibilité : il est possible qu'on souhaite changer le format de sérialisation de l'arbre, ou même l'algorithme. En rajoutant un système de version sur les différentes parties, on peut permettre de faire évoluer un type de fichier en préservant la compatibilité avec les versions précédentes.

42.2.3 Algorithme de Lempel-Ziv-Welch

L'algorithme d'Huffman est efficace, mais il présente un désavantage majeur : il nécessite de lire le contenu d'un fichier dans son intégralité pour pouvoir déterminer un code préfixe optimal. Il est toutefois possible de modifier l'algorithme pour lever cette limitation. Dans ce paragraphe, nous allons plutôt étudier une autre technique de compression qui se programme assez facilement, qui se révèle même plus efficace que Huffman sur de gros textes (voir les mesures à la fin de ce paragraphe) et permet de compresser des flux plutôt que des fichiers. C'est-à-dire qu'on peut compresser et décompresser des données au fur et à mesure qu'elles sont transmises.

Il s'agit de l'algorithme de Lempel-Ziv-Welch, appelé communément compression LZW, et qui est une modification faite en 1984 par Welch de l'algorithme de LZ78 de Lempel et Ziv.

Principe de la compression

L'idée de l'algorithme LZW est de faire avancer une fenêtre sur le texte en maintenant une table des motifs déjà rencontrés. Quand on rencontre un motif déjà vu, on le code avec une référence vers la table et quand on rencontre un nouveau motif, on le code tel quel en rajoutant une entrée dans la table.

Pour la table, on peut utiliser un tableau dynamique de motifs (ref FIXME) dont la taille ne pourra pas dépasser 2^d éléments ou directement un tableau de 2^d valeurs optionnelles, dans la mesure où d est en général petit. Ainsi, on pourra référencer chaque motif avec un mot de d bits. Afin de pouvoir retrouver efficacement l'indice associé à un motif, on utilise une table de hachage (ref FIXME) réalisant l'inverse de la table.

L'algorithme procède alors ainsi pour compresser :

- on initialise la table avec une entrée pour chaque caractère, donc chaque octet, en considérant des caractères sur 8 bits.
- on maintient une variable contenant un suffixe m du texte lu, appelé motif courant, qui est toujours présent dans la table ; il est initialisé avec la première lettre du texte.
- on lit alors chaque caractère x :
  - Soit mx est dans la table, et alors on remplace le motif courant par $m \leftarrow mx$.
  - Soit mx n'est pas dans la table, par construction m y est nécessairement, on produit alors le code correspondant à m, on rajoute une entrée dans la table pour mx si elle contient moins de 2^d éléments et on repart de $m \leftarrow x$.
- quand tous les caractères ont été lus, on produit le code correspondant à m.

Voici les différentes étapes pour la compression de la chaîne `AABABAAAB` qui produit la suite d'entiers `65,65,66,257,256,257` qui seront alors codés dans un fichier sur d bits. Les 256 premières entrées de la table ont été volontairement ignorées. On remarque juste que `A` correspond à l'index 65 et `B` à l'index 66.

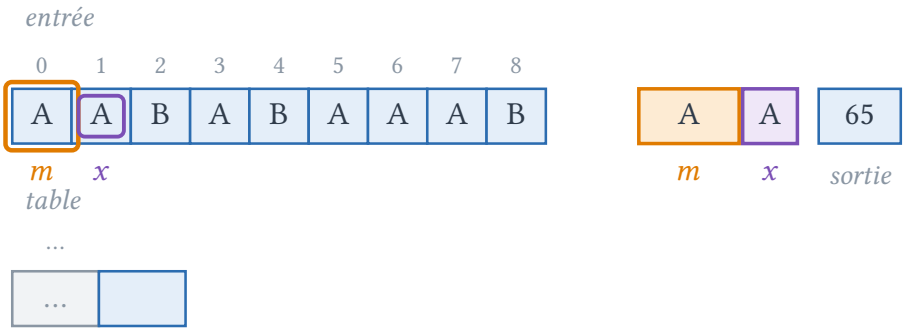


Figure 42.7 — m = A, x = A : AA n'est pas dans la table, on produit 65 et on ajoute AA.

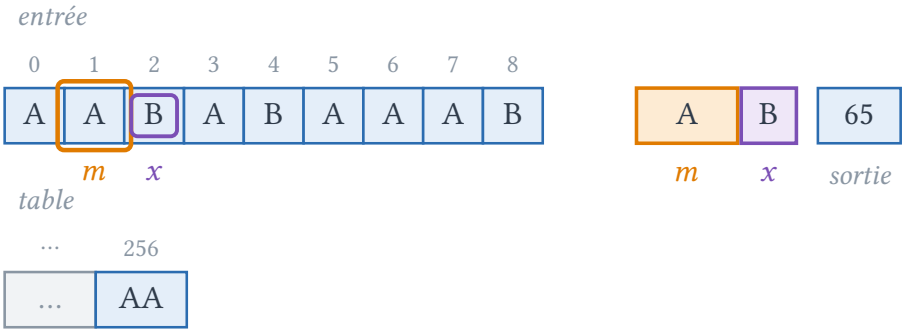


Figure 42.8 — m = A, x = B : on produit 65 et on ajoute AB.

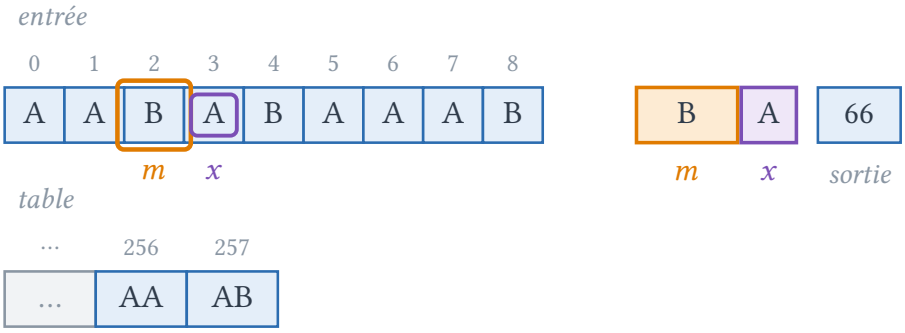


Figure 42.9 — m = B, x = A : on produit 66 et on ajoute BA.

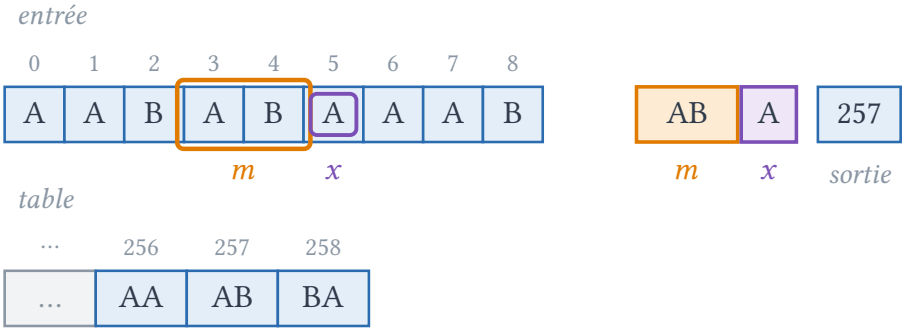


Figure 42.10 — m = AB, x = A : on produit 257 et on ajoute ABA.

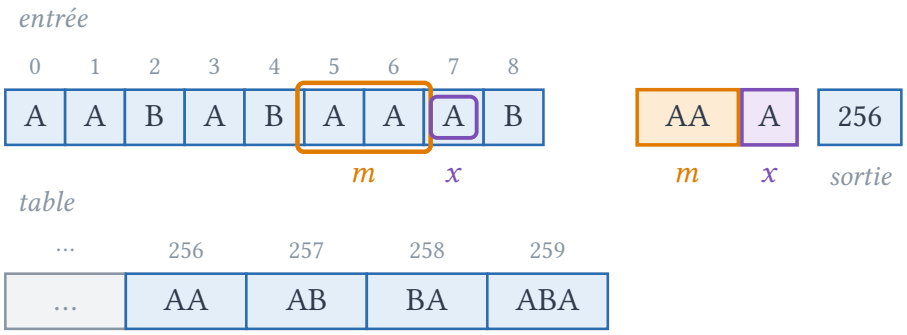


Figure 42.11 — m = AA, x = A : on produit 256 et on ajoute AAA.

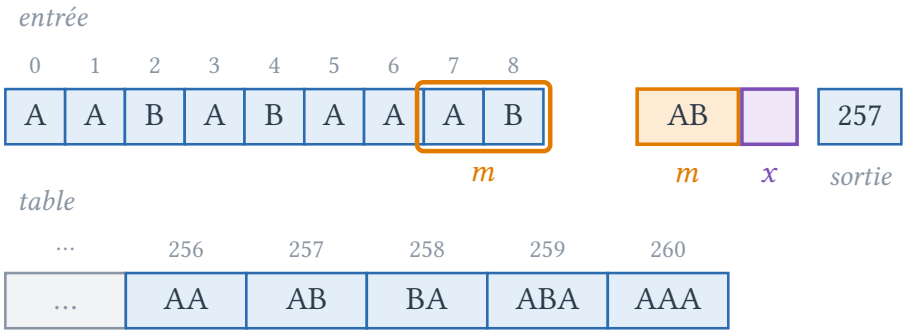


Figure 42.12 — Fin du texte : on produit 257, le code de m = AB.

Principe de la décompression

Pour décompresser, on effectue la procédure précédente en sens inverse. Cependant, il faut reconstruire la table en même temps qu'on lit le fichier compressé. Dans la majorité des cas, c'est assez immédiat. Pour le premier code lu, il s'agit forcément d'une référence à un des 256 caractères, donc on le reproduit. À partir du second code lu :

- on lit un code n où $n <$ | Fira Sans(table) | et Fira Sans(table)$[n] = xm'$, x est un caractère et m' un mot.
- on écrit xm' dans le fichier de sortie.
- on rajoute ensuite mx dans la table où c est le précédent code lu et Fira Sans(table)$[c] = m$.

En faisant ainsi, on reproduit le processus de compression mais en remplissant la table avec un temps de retard. En effet, si on reprend le principe exposé plus haut, une entrée pour mx est ajoutée dans la table quand on lit le caractère x et que le motif lu précédemment est m, on repart alors avec x pour motif lu. C'est exactement ce qu'on fait ici en tenant compte du premier caractère de Fira Sans(table)$[n]$. Contrairement à la compression, il est inutile ici de retrouver l'indice associé à un motif. On peut donc ignorer la table de hachage utilisée par la compression.

Voici les étapes de décompression de l'exemple précédent :

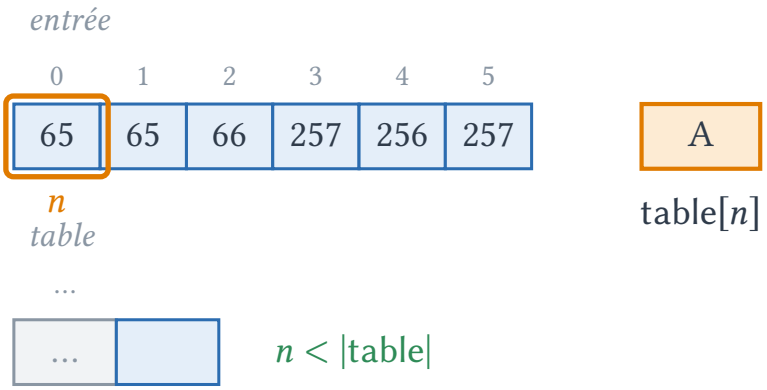


Figure 42.13 — Premier code, 65 : on écrit A.

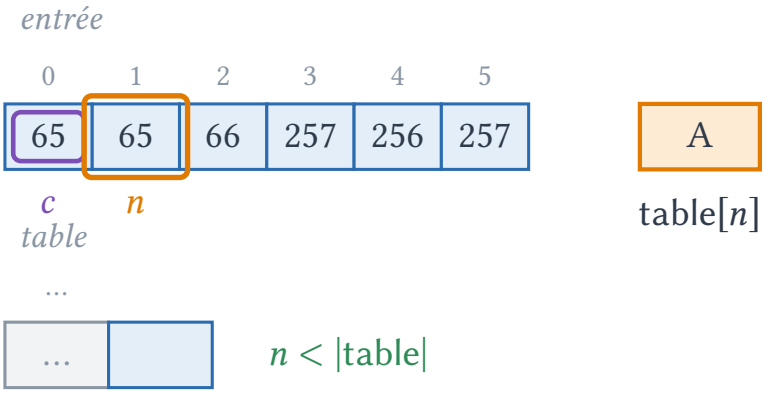


Figure 42.14 — $n = 65$: on écrit A et on ajoute AA à la table.

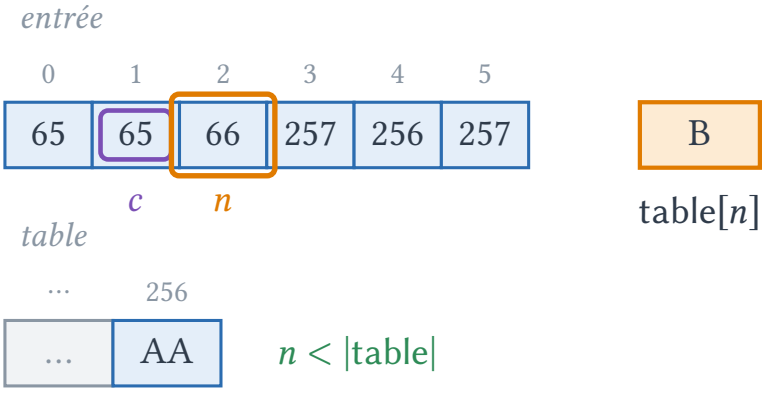


Figure 42.15 — $n = 66$: on écrit B et on ajoute AB.

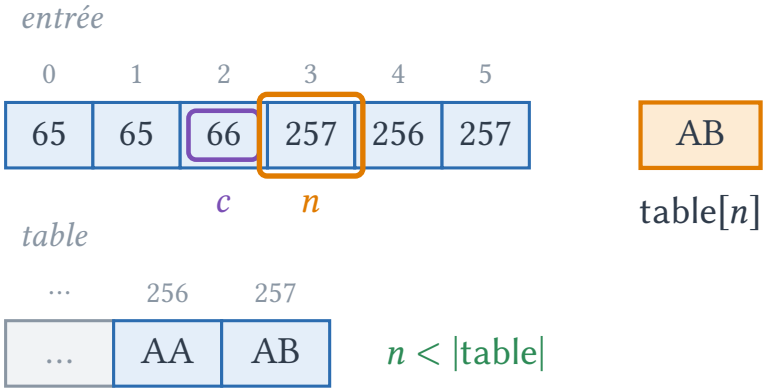


Figure 42.16 — $n = 257$: on écrit AB et on ajoute BA.

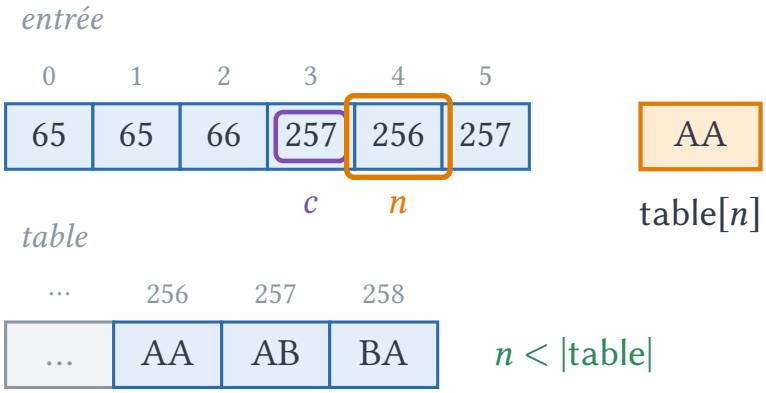


Figure 42.17 — $n = 256$: on écrit AA et on ajoute ABA.

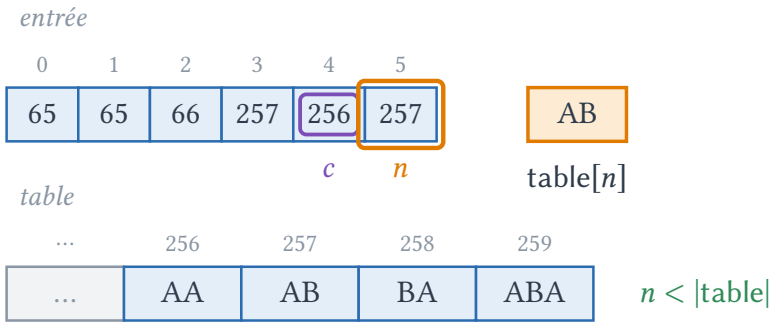


Figure 42.18 — $n = 257$: on écrit AB et on ajoute AAA.

Il reste toutefois un cas à traiter, celui où $n = |\text{Fira Sans(table)}|$, c'est-à-dire quand on lit un code qui n'est pas encore présent dans la table. Pour comprendre ce cas, il est important d'identifier précisément quand il se produit dans le processus de compression. Comme on vient de le voir, on rajoute une entrée pour mx après avoir produit le code c correspondant à m. Pour que le code n ne soit pas présent dans la table, il faut donc que n corresponde à cette entrée mx. Or, quand on a compressé, on est reparti du motif x à ce moment là, donc nécessairement m commence par x. Cela signifie qu'on peut reconstruire Fira Sans(table)$[n]$ en décompressant avec mx où x est la première lettre de Fira Sans(table)$[c] = m$.

On en déduit alors la procédure complète suivante :

- on initialise la table avec une entrée pour chaque caractère, donc chaque octet, en considérant des caractères sur 8 bits.
- on maintient une variable c contenant le dernier code lu qu'on initialise avec le premier code en produisant le caractère correspondant.

- pour chaque code n :
  - Soit $n <$ | Fira Sans(table) | et Fira Sans(table)$[n] = xm'$ où x est un caractère, alors on écrit xm' en sortie
  - Soit $n =$ | Fira Sans(table) | et alors on écrit en sortie Fira Sans(table)$[c]x$ où x est le premier caractère de Fira Sans(table)$[c]$
  - Dans tous les cas, on ajoute Fira Sans(table)$[c]x$ à la table (si elle contient moins de 2^d éléments), puis on remplace $c \leftarrow n$.

Il y a un exemple classique où on a besoin de traiter ce cas : celui où le même caractère est présent plusieurs fois en début de fichier. Comme LZW est utilisé pour compresser des formats d'images où les pixels sont des indices dans une palette de 256 couleurs avec une couleur pour la transparence, le cas où la même couleur est présente au début du fichier est fréquent. Voici ici un exemple de compression et de décompression pour AAA :

Compression

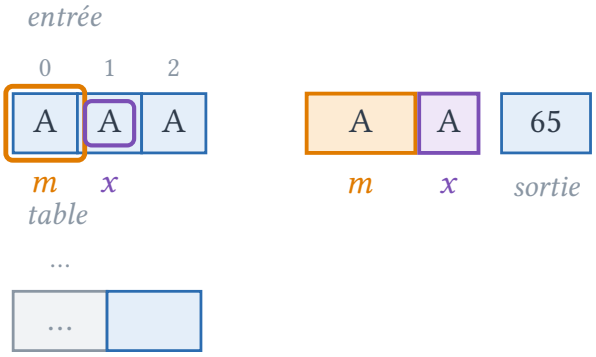


Figure 42.19 — m = A, x = A : on produit 65 et on ajoute AA.

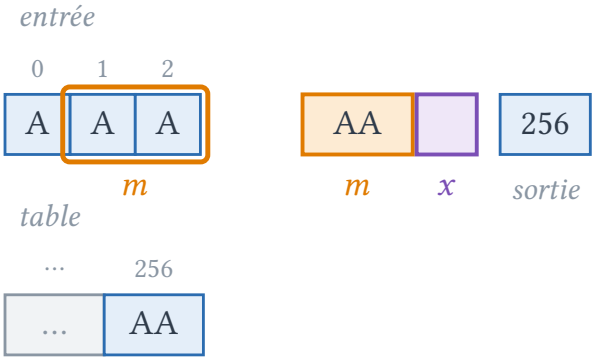


Figure 42.20 — Fin du texte : on produit 256, le code de AA.

Décompression

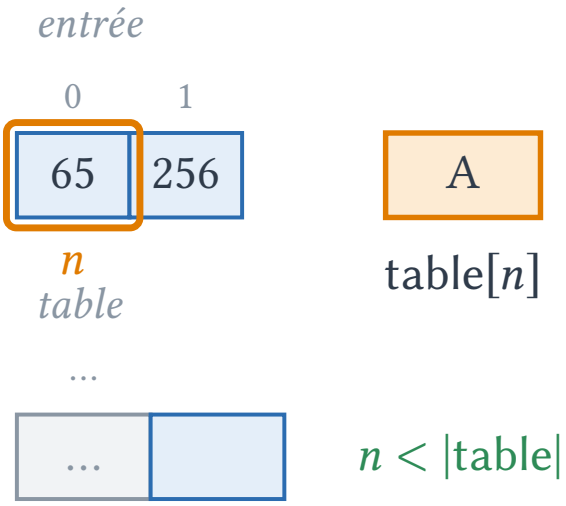


Figure 42.21 — Premier code, 65 : on écrit A.

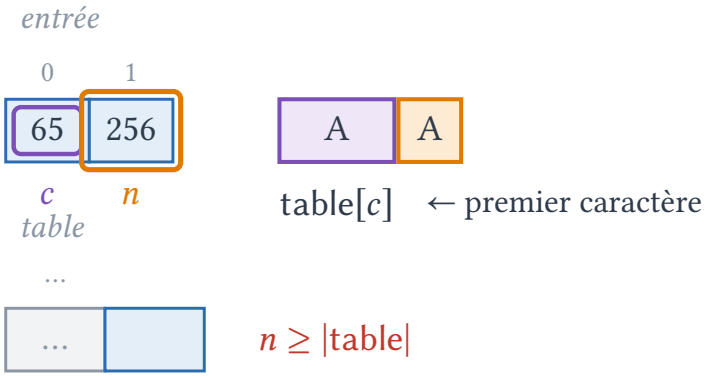


Figure 42.22 — 256 n'est pas encore dans la table : on écrit table[c] suivi de son premier caractère, AA.

Remarque 42.22 On peut pousser un peu plus l'analyse précédente afin d'identifier précisément les situations menant à ces cas problématiques. Comme on vient de le voir, il est nécessaire que le texte à compresser contienne un motif de la forme $xwxwx$ où w est un mot et x une lettre. Mais pour que le motif lu contienne xw au moment où on ajoute l'entrée xwx il faut que xw lui même soit dans la table, ce qui signifie que chaque préfixe de xw est également dans la table au moment où on commence à le lire, mais que si le motif précédent est w' alors $w'x$ n'était pas dans la table.

On considère un mot $w = a_1...a_n$ où toutes les lettres sont distinctes, et on va essayer de construire un mot p tel que la compression de p permettra d'avoir a_1, a_1a_2, $a_1a_2a_3$, ..., $a_1...a_n$ dans la table.

Si on lit a_1a_2, on va ajouter a_1a_2 dans la table (a_1 y étant déjà). On continue alors en partant de a_2 comme motif, pour pouvoir ajouter $a_1a_2a_3$ à la table, il faut lire $a_1a_2a_3$ ensuite car à la lecture de a_1, comme a_2a_1 n'est pas dans la table on l'ajoute et m devient a_1, puis a_1a_2. Ainsi $a_1a_2a_1a_2a_3$ va ajouter a_1, a_1a_2, $a_1a_2a_3$ dans la table. En continuant ainsi avec $a_1a_2a_3a_4$, comme a_3a_1 n'est pas dans la table, on va procéder de même.

Le mot $p = a_1a_2a_1a_2a_3a_1a_2a_3a_4...a_1a_2...a_n$ va donc permettre d'ajouter tous les préfixes de w dans la table.

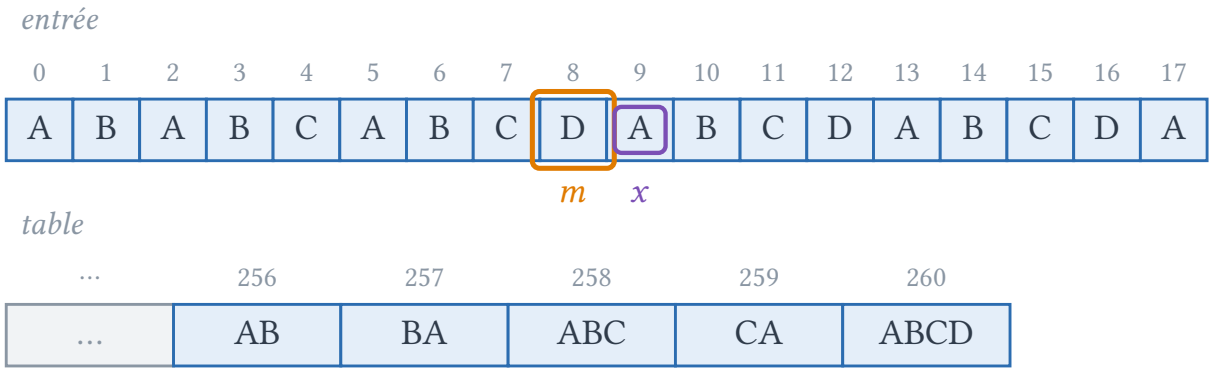
Maintenant, pour obtenir le cas précédent, il suffit de lire le texte $pwwa_1x = pa_1w_0a_1w_0a_1x$ où $w_0 = a_2...a_n$ et x n'est pas l'un des a_i. En effet, on lit d'abord p ce qui permet d'ajouter tous les préfixes. Comme a_na_1 n'est pas dans la table, on continue avec a_1 comme motif, on l'augmente jusqu'à lire a_1w_0 puis on ajoute une entrée pour $a_1w_0a_1$ en ayant a_1 comme motif, et là on va lire $a_1w_0a_1x$ et à la lecture de x, produire le code de $a_1w_0a_1$ qui est le dernier saisi dans la table.

Remarquons qu'on peut aussi ne considérer que $pwwa_1$ dans la mesure où la fin du fichier provoquera aussi l'écriture du code.

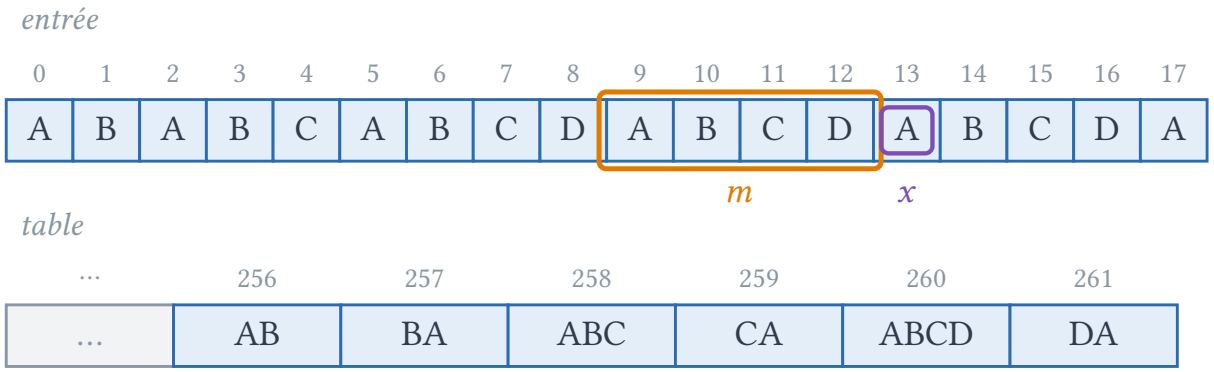
Si on considère w =ABCD, on a p =ABABCABCD et on pourra ainsi compresser le texte

$$pwwA = pAw_0Aw_0A = \text{Fira Sans(ABABCABCDABCDABCDA)}$$

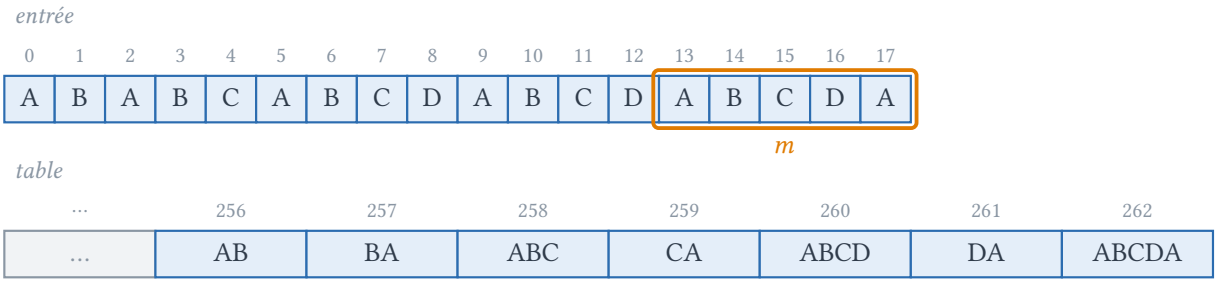
Après lecture de p on est dans la configuration suivante :



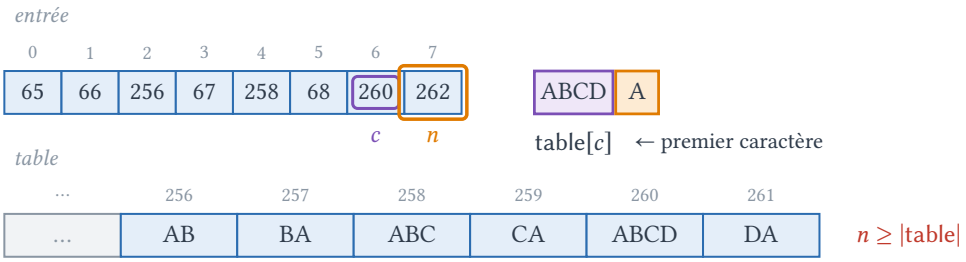
Puis après lecture de $w A = A w_0 A$:



Enfin, la lecture du reste du texte va déclencher l'ajout de `ABCDA` dans la table et l'écriture de son code.



À la décompression, on produira bien le texte correspondant avec le principe présenté plus haut :



Implémentation

Avant de commencer à implémenter l'algorithme, il est nécessaire de définir des fonctions de manipulation des entiers sur d bits et de lecture/écriture dans un fichier.

Tout d'abord, on définit des fonctions d'écriture d'entiers sous forme codée sur d bits à l'aide des fonctions vues précédemment :

OCaml

```
let input_code f longueur_code =
    let acc = ref 0 in
    for i = 0 to longueur_code - 1 do
        let b = input_bit f in
        if b
        then acc := !acc + (1 lsl i)
    done;
    !acc

let output_code f code longueur_code =
    assert (code < 1 lsl longueur_code);

    let acc = ref code in
    for i = 0 to longueur_code - 1 do
        output_bit f (!acc mod 2 = 1);
        acc := !acc / 2
    done
```

C

```
/* renvoie le code lu, ou -1 s'il n'y a plus de données */
int input_code(in_channel_bits *f, int longueur_code)
{
    int acc = 0;
    for (int i = 0; i < longueur_code; i++)
    {
        int b = input_bit(f);
        if (b < 0)
        {
            assert(i == 0); // pas au milieu d'un code
            return -1;
        }
        if (b)
            acc = acc + (1 << i);
    }
    return acc;
}

void output_code(out_channel_bits *f, int code, int longueur_code)
{
    assert(code < 1 << longueur_code);
    for (int i = 0; i < longueur_code; i++)
    {
        output_bit(f, code % 2 == 1);
        code = code / 2;
    }
}
```

Python

```
def read_code(self, longueur_code):
    acc = 0
    for i in range(longueur_code):
        b = self.read_bit()
        if b == b'':
            assert(i == 0) # pas au milieu d'un code
            return None
        if b:
            acc += 1 << i
    return acc
def write_code(self, code, longueur_code):
    assert(code < 1 << longueur_code)
    for i in range(longueur_code):
        self.write_bit( code % 2 == 1 )
        code //= 2
```

On implémente alors assez directement la compression :

OCaml

```
type table_bidir = {
    elements : string array;
    indices : (string, int) Hashtbl.t;
    mutable n_elements : int
}

let string_of_byte b = String.make 1 (Char.chr b)

let cree_table longueur_code =
    let taille_table = 1 lsl longueur_code in
    let elements = Array.make taille_table "" in
    let indices = Hashtbl.create taille_table in
    for i = 0 to 255 do
        let s = string_of_byte i in
        elements.(i) <- s;
        Hashtbl.add indices s i
    done;
    { elements = elements; indices =  indices; n_elements = 256 }

let ajoute_entree table s =
    table.elements.(table.n_elements) <- s;
    Hashtbl.add table.indices s table.n_elements;
    table.n_elements <- table.n_elements + 1

let compresse_fichier nom_in nom_out longueur_code =
    let table = cree_table longueur_code in
    let taille_table = 1 lsl longueur_code in
    let f_out = Bitpacking.open_out_bits nom_out in
    let f_in = open_in_bin nom_in in
    if in_channel_length f_in > 0 (* rien à écrire pour un fichier vide *)
    then begin
        let m = ref (string_of_byte (input_byte f_in)) in
        begin
          try
            while true do
                let c = string_of_byte (input_byte f_in) in
                let mc = !m ^ c in
                match Hashtbl.find_opt table.indices mc with
                | Some _ -> m := mc
                | None -> begin
                    let i = Hashtbl.find table.indices !m in
                    Bitpacking.output_code f_out i longueur_code;
                    if table.n_elements < taille_table
                    then ajoute_entree table mc;
                    m := c
                end
            done
```

```
          with End_of_file -> ()
        end;
        (* code final *)
        let i = Hashtbl.find table.indices !m in
        Bitpacking.output_code f_out i longueur_code
    end;
    close_in f_in;
    Bitpacking.close_out_bits f_out
```

Python

```python
def compresse_(nom_in, nom_out, longueur_code=12):
    table = [ bytes([c]) for c in range(256) ]
    indices = { bytes([c]) : c for c in range(256) }

    with open(nom_in, 'rb') as fin, \
            Bitpacking(nom_out, 'wb') as fout:
        m = fin.read(1)
        if m == b'': # fichier vide : rien à écrire
            return
        while True:
            c = fin.read(1)
            if c == b'': break
            if m+c in indices:
                m = m+c
            else:
                fout.write_code(indices[m], longueur_code)
                if len(table) < 1 << longueur_code:
                    indices[m+c] = len(table)
                    table.append(m+c)
                m = c

        # code final
        fout.write_code(indices[m], longueur_code)
```

✍ **C**

```c
/* En C, l'entrée mx de la table est représentée par le couple
 * (code de m, x) : prefixe[i] est le code de m (-1 pour les 256
 * premières entrées) et octet[i] le caractère x. La table de hachage
 * indices, à adressage ouvert, associe à un couple son code. */
struct table_lzw {
    int *prefixe;
    int *octet;
    int n_elements;
    int taille_table;  // 2^longueur_code
    int *indices;      // de taille 2 * taille_table, -1 si case vide
};
typedef struct table_lzw table_lzw;

table_lzw *cree_table(int longueur_code)
{
    table_lzw *t = malloc(sizeof(table_lzw));
    t->taille_table = 1 << longueur_code;
    t->prefixe = malloc(sizeof(int) * t->taille_table);
    t->octet = malloc(sizeof(int) * t->taille_table);
    t->indices = malloc(sizeof(int) * 2 * t->taille_table);
    for (int h = 0; h < 2 * t->taille_table; h++)
        t->indices[h] = -1;
    for (int i = 0; i < 256; i++)
    {
        t->prefixe[i] = -1;
        t->octet[i] = i;
    }
    t->n_elements = 256;
    return t;
}
```

```c
void libere_table(table_lzw *t)
{
    free(t->prefixe);
    free(t->octet);
    free(t->indices);
    free(t);
}

/* case de indices où se trouve, ou devrait se trouver, le couple (m,x) */
int case_hachage(table_lzw *t, int m, int x)
{
    int h = (m * 256 + x) % (2 * t->taille_table);
    while (t->indices[h] >= 0
           && (t->prefixe[t->indices[h]] != m || t->octet[t->indices[h]] != x))
        h = (h + 1) % (2 * t->taille_table);
    return h;
}

/* code de l'entrée mx, ou -1 si elle est absente */
int cherche(table_lzw *t, int m, int x)
{
    return t->indices[case_hachage(t, m, x)];
}

void ajoute_entree(table_lzw *t, int m, int x)
{
    int i = t->n_elements;
    t->prefixe[i] = m;
    t->octet[i] = x;
    t->indices[case_hachage(t, m, x)] = i;
    t->n_elements++;
}

void compresse_fichier(const char *nom_in, const char *nom_out, int longueur_code)
{
    table_lzw *table = cree_table(longueur_code);
    out_channel_bits *f_out = open_out_bits(nom_out);
    FILE *f_in = fopen(nom_in, "rb");
    assert(f_in != NULL);
    int m = fgetc(f_in); // code du motif courant

    if (m != EOF) // rien à écrire pour un fichier vide
    {
        int c;
        while ((c = fgetc(f_in)) != EOF)
        {
            int mc = cherche(table, m, c);
            if (mc >= 0)
                m = mc;
            else {
                output_code(f_out, m, longueur_code);
                if (table->n_elements < table->taille_table)
                    ajoute_entree(table, m, c);
                m = c;
            }
        }
        output_code(f_out, m, longueur_code); // code final
    }

    fclose(f_in);
    close_out_bits(f_out);
    libere_table(table);
}
```

Et on procède de même pour la décompression :

OCaml

```
let decompresse_fichier nom_in nom_out longueur_code =
    let table = cree_table longueur_code in
    let taille_table = 1 lsl longueur_code in
    let f_in = Bitpacking.open_in_bits nom_in in
    let f_out = open_out_bin nom_out in
    begin
       try (* End_of_file dès ici pour un fichier vide *)
          let code = ref (Bitpacking.input_code f_in longueur_code) in
          output_string f_out table.elements.(!code);
          while true do
              let nouveau = Bitpacking.input_code f_in longueur_code in
              let s_code = table.elements.(!code) in
              let s =
                  if nouveau = table.n_elements
                  then let x = String.sub s_code 0 1
                       in s_code ^ x
                  else table.elements.(nouveau) in
              let x = String.sub s 0 1 in
              output_string f_out s;
              if table.n_elements < taille_table
              then ajoute_entree table (s_code ^ x);
              code := nouveau
        done
      with End_of_file -> ()
    end;
    Bitpacking.close_in_bits f_in;
    close_out f_out
```

Python

```
def decompresse_(nom_in, nom_out, longueur_code=12):
    table = [ bytes([c]) for c in range(256) ]

    with Bitpacking(nom_in, 'rb') as fin, \
            open(nom_out, 'wb') as fout:

        code = fin.read_code(longueur_code)
        if code is None: # fichier vide
            return
        fout.write(table[code])

        while True:
            nouveau = fin.read_code(longueur_code)
            if nouveau is None:
                break

            if len(table) == nouveau:
                x = table[code][0:1] # car bytes
                s = table[code]+x
            else:
                s = table[nouveau]
                x = s[0:1] # car bytes
            fout.write(s)
            if len(table) < 1 << longueur_code:
               table.append(table[code]+x)
            code = nouveau
```

C

```
/* écrit dans f le motif d'indice i et renvoie son premier caractère.
 * tampon, de taille taille_table, contient au moins la longueur du motif */
int ecrit_entree(FILE *f, table_lzw *t, int i, unsigned char *tampon)
{
    int k = t->taille_table;
    // on remonte les préfixes en remplissant le tampon par la fin
    while (i >= 0)
    {
        k--;
```

```
        tampon[k] = t->octet[i];
        i = t->prefixe[i];
    }
    fwrite(tampon + k, 1, t->taille_table - k, f);
    return tampon[k];
}

void decompresse_fichier(const char *nom_in, const char *nom_out, int longueur_code)
{
    table_lzw *table = cree_table(longueur_code);
    unsigned char *tampon = malloc(table->taille_table);
    in_channel_bits *f_in = open_in_bits(nom_in);
    FILE *f_out = fopen(nom_out, "wb");
    assert(f_out != NULL);
    int code = input_code(f_in, longueur_code);

    if (code >= 0) // sinon fichier vide
    {
        ecrit_entree(f_out, table, code, tampon);
        int nouveau;
        while ((nouveau = input_code(f_in, longueur_code)) >= 0)
        {
            int x;
            if (nouveau == table->n_elements)
            {   // on écrit table[code] suivi de son premier caractère
                x = ecrit_entree(f_out, table, code, tampon);
                fputc(x, f_out);
            } else
                x = ecrit_entree(f_out, table, nouveau, tampon);
            if (table->n_elements < table->taille_table)
                ajoute_entree(table, code, x);
            code = nouveau;
        }
    }

    close_in_bits(f_in);
    fclose(f_out);
    free(tampon);
    libere_table(table);
}
```

Remarque 42.23 Pour un fichier vide, il n'y a pas de motif courant initial : la compression n'écrit aucun code et le fichier compressé se réduit à l'octet final de padding. La décompression détecte alors la fin des données dès la lecture du premier code et produit un fichier vide.

Dans la version C, un motif n'est pas stocké comme une chaîne mais comme le couple formé du code de son préfixe et de son dernier caractère. Cela évite de recopier des chaînes et de devoir hacher des chaînes de longueur quelconque ; en contrepartie, la décompression doit remonter la chaîne des préfixes pour écrire un motif. On remarque qu'un motif de la table d'indice $i \geq 256$ est de longueur au plus $i - 254$, ce qui justifie la taille du tampon.

Impact de la longueur du code

Afin d'étudier l'impact de la longueur du code sur la taille des fichiers compressés, on considère deux fichiers :

- `proust.txt` contenant, en 7543768 octets, l'intégrale de *à la recherche du temps perdu* de Marcel Proust
- `code.py` contenant 8566 octets de code source Python

On note np la taille en nombre d'octets après compression du fichier `proust.txt` et tp le nombre d'entrées dans la table à la fin du processus. De même, on note nc et tc les valeurs respectives pour le fichier `code.py`.

On obtient alors les valeurs suivantes en fonction du nombre d de bits du code :

d	np	nc	tp	tc
8	7543768	8566	256	256
9	5056398	5709	512	512
10	4195124	4431	1024	1024
11	3864174	3948	2048	2048
12	3612505	4039	4096	2947
13	3434424	4376	8192	2947
14	3262145	4712	16384	2947
15	3131790	5049	32768	2947
16	2998639	5385	65536	2947
17	2682264	5722	131072	2947
18	2554323	6058	262144	2947
19	2500314	6395	524288	2947
20	2557656	6731	1023317	2947

On constate qu'il est nécessaire d'avoir un texte riche pour bénéficier d'une grande longueur de code. Le fichier `code.py` ne contenant pas plus que 2947 motifs. Même si le fichier `proust.txt` en contient plus que les tailles considérées ici, il y a un compromis qui s'établit entre la richesse de la table et la taille du code. Ainsi, il semble que le fichier `proust.txt` soit compressé de manière optimale avec un code de longueur 19.

42.3 Problèmes supplémentaires

42.3.1 Transformation de Burrows-Wheeler

La transformation de Burrows-Wheeler (1994) ne compresse pas : elle permute les lettres d'un texte de façon réversible, en sorte que le résultat contienne de longues plages de lettres identiques, qu'il est ensuite facile de compresser. C'est la première étape de l'outil `bzip2`.

On considère un mot u de longueur n qui se termine par un caractère spécial `$`, présent une seule fois et plus petit que toutes les autres lettres. On écrit les n rotations de u, c'est-à-dire les mots vw tels que $u = wv$, et on les trie dans l'ordre lexicographique. La transformée Fira Sans(BWT)(u) est le mot formé des **dernières** lettres de ces rotations triées.

Exemple 42.24

Pour u = `ananas$`, on obtient les rotations triées suivantes, où l'on note F la première colonne et L la dernière :

ligne	rotation	F	L
0	`$ananas`	`$`	`s`
1	`ananas$`	`a`	`$`
2	`anas$an`	`a`	`n`
3	`as$anan`	`a`	`n`
4	`nanas$a`	`n`	`a`
5	`nas$ana`	`n`	`a`
6	`s$anana`	`s`	`a`

On a donc Fira Sans(BWT)(u) = `s$nnaaa`. Les lettres égales ont été regroupées : la dernière lettre d'une ligne est celle qui précède, dans u, le début de la ligne. Les lignes étant triées selon ce qui suit, des

contextes semblables, comme `an` ou `as` dans un texte en français, se retrouvent côte à côte et sont souvent précédés de la même lettre.

Une fois cette transformation effectuée, on applique typiquement le codage *move to front* présenté au paragraphe suivant, qui transforme les plages de lettres identiques en suites de petits entiers, puis un codage de Huffman.

Le point remarquable est que l'on peut retrouver u à partir de L seul. Tout d'abord, F s'obtient en triant les lettres de L. Ensuite, chaque ligne est une rotation : L_i est la lettre qui précède F_i dans u, vu de manière circulaire. Il reste à savoir, pour une lettre L_i, sur quelle ligne elle se trouve en première position.

Propriété 42.25

Pour toute lettre c, la k-ième occurrence de c dans L et la k-ième occurrence de c dans F correspondent à la même position dans u.

Preuve Les lignes qui se terminent par c sont de la forme xc et les faire tourner d'un cran donne les lignes cx qui commencent par c. Or, pour deux mots $x \neq y$ de même longueur, $xc < yc \Leftrightarrow x < y \Leftrightarrow cx < cy$. Les lignes xc sont distinctes, car la lettre `$` n'apparaît qu'une fois, donc les deux familles de lignes sont rangées dans le même ordre.

□

On en déduit la fonction Fira Sans(LF) qui à la ligne i associe la ligne commençant par la lettre L_i considérée. Comme `$` est minimal, la ligne 0 commence par `$` : L_0 est donc la dernière lettre de u avant `$`. La lettre précédente est $L_{\text{Fira Sans(LF)}(0)}$, puis $L_{\text{Fira Sans(LF)(Fira Sans(LF)}(0))}$, et ainsi de suite : on reconstruit u de droite à gauche.

Exemple 42.26

Avec L = `s$nnaaa`, on part de la ligne 0 : L_0 = `s`, première occurrence de `s` dans L, donc on passe à la ligne de la première occurrence de `s` dans F, la ligne 6. Ensuite L_6 = `a`, troisième `a` de L, d'où la ligne 3, puis L_3 = `n`, deuxième `n`, d'où la ligne 5, et ainsi de suite. On lit successivement `s`, `a`, `n`, `a`, `n`, `a` et on retrouve `ananas$`.

Un tri stable des indices de L selon leur lettre donne directement la correspondance entre les occurrences de L et celles de F. Les implémentations suivantes sont volontairement naïves pour la transformation, qui trie explicitement les n rotations :

OCaml

```
let bwt (u:string) : string =
    (* u doit finir par un unique caractère '$' plus petit que les autres *)
    let n = String.length u in
    let rotation i = String.sub u i (n-i) ^ String.sub u 0 i in
    let rotations = Array.init n rotation in
    Array.sort compare rotations;
    String.init n (fun r -> rotations.(r).[n-1])

let inverse_bwt (l:string) : string =
    let n = String.length l in
    (* ordre.(r) : indice dans l de la lettre en position r de la première
       colonne, le tri stable conservant l'ordre des lettres égales *)
    let ordre = Array.init n (fun i -> i) in
    Array.stable_sort (fun i j -> compare l.[i] l.[j]) ordre;
    let lf = Array.make n 0 in
    Array.iteri (fun r i -> lf.(i) <- r) ordre;
    let lettres = Bytes.make n '$' in
    let i = ref 0 in (* la ligne 0 commence par '$' *)
    for k = n-2 downto 0 do
        Bytes.set lettres k l.[!i]; (* lettre qui précède la ligne i *)
```

```
        i := lf.(!i)
    done;
    Bytes.to_string lettres
```

Python

```python
def bwt(u):
    '''Transformée de Burrows-Wheeler de u, qui doit finir par
    un unique caractère $ plus petit que les autres'''
    n = len(u)
    rotations = sorted(u[i:] + u[:i] for i in range(n))
    return ''.join(r[n-1] for r in rotations)

def inverse_bwt(l):
    n = len(l)
    # ordre[r] : indice dans l de la lettre en position r de la
    # première colonne, le tri étant stable
    ordre = sorted(range(n), key=lambda i: l[i])
    lf = [ 0 ] * n
    for r in range(n):
        lf[ordre[r]] = r
    lettres = []
    i = 0 # la ligne 0 commence par $
    for _ in range(n-1):
        lettres.append(l[i]) # lettre qui précède la première de la ligne i
        i = lf[i]
    return ''.join(reversed(lettres)) + '$'
```

Remarque 42.27 Trier explicitement les rotations coûte $O(n^2 \log n)$ comparaisons de caractères dans le pire cas. En pratique, on obtient l'ordre des rotations en temps linéaire à l'aide d'une *table des suffixes*, ce qui dépasse le cadre de ce cours. L'inversion, elle, se fait ici en $O(n \log n)$, et même en $O(n)$ avec un tri par comptage puisque l'alphabet est fini.

42.3.2 Move to front

42.3.3 La structure de données **corde**

42.3.4 L'algorithme de Knuth-Morris-Pratt

L'algorithme naïf oublie tout ce qu'il a lu dès qu'une comparaison échoue et recommence à la position suivante. L'algorithme de Knuth-Morris-Pratt (1977) lit la chaîne de gauche à droite, sans jamais revenir en arrière dans la chaîne, en exploitant ce qu'on sait déjà : après avoir reconnu un préfixe du motif, la seule information utile est la structure de ce préfixe, qu'on précalcule. On obtient ainsi une complexité en pire cas en $O(n+p)$, ce qu'on avait seulement admis pour Boyer-Moore.

Cette partie reprend les notations du TP Knuth-Morris-Pratt, où l'on programme en OCaml le calcul des bords présenté ci-dessous.

Bords maximaux

On rappelle qu'un bord d'un mot u est un mot qui est à la fois préfixe et suffixe propre de u. Le mot vide μ est un bord de tout mot non vide.

Définition 42.28

Le **bord maximal** d'un mot non vide u est son bord de longueur maximale, on le note $\beta(u)$.

Pour un motif $m = m_0 \ldots m_{p-1}$ et $i \in [\![1, p]\!]$, on note $\ell_i = |\, \beta(m_0 \ldots m_{i-1}) \,|$ la longueur du bord maximal du préfixe de longueur i, et on pose par convention $\ell_0 = 0$.

Un bord de u étant un préfixe de u, il est caractérisé par sa longueur : c'est pour cela qu'on ne manipule que des longueurs.

Exemple 42.29

Pour $m = abacabab$, on obtient :

i	0	1	2	3	4	5	6	7	8
ℓ_i	0	0	0	1	0	1	2	3	2

Par exemple, $\beta(abacaba) = aba$ et $\beta(abacabab) = ab$.

Lemme 42.30

Soit u un mot non vide. Les bords de u sont exactement les mots $\beta(u), \beta^2(u), ..., \beta^r(u) = \mu$ où r est le premier entier tel que $\beta^r(u) = \mu$.

Preuve La suite des longueurs $|\ \beta^k(u)\ |$ est strictement décroissante tant que $\beta^k(u) \neq \mu$, donc r existe.

Si v est un bord de u et w un mot plus court que v, alors w est un bord de u si et seulement si w est un bord de v. En effet, w et v sont tous deux préfixes de u, donc w est préfixe de u si et seulement si il est préfixe de v. De même pour les suffixes.

Par récurrence, chaque $\beta^k(u)$ est un bord de u : c'est un bord de $\beta^{k-1}(u)$, plus court, donc un bord de u. Réciproquement, soit w un bord de u et k maximal tel que $|\ w\ | \leq |\ \beta^k(u)\ |$ (on a $|\ w\ | \leq |\ \beta(u)\ |$ par maximalité de $\beta(u)$). Si $w \neq \beta^k(u)$, alors w est plus court que $\beta^k(u)$ donc est un bord de $\beta^k(u)$, d'où $|\ w\ | \leq |\ \beta^{k+1}(u)\ |$, ce qui contredit la maximalité de k. Donc $w = \beta^k(u)$.

□

Ainsi, les longueurs des bords du préfixe $m_0...m_{i-1}$ sont $\ell_i, \ell_{\ell_i}, \ell_{\ell_{\ell_i}}, ..., 0$: le tableau des ℓ_i contient toute l'information sur les bords de tous les préfixes.

Théorème 42.31

Soit $i \in [\![1, p-1]\!]$ et $v = m_0...m_{i-1}$. Les bords non vides de vm_i sont exactement les mots wm_i où w est un bord de v tel que $m_{|\ w\ |} = m_i$.

Ainsi, $\ell_{i+1} = 1 + k$ où k est le plus grand élément de $\{\ell_i, \ell_{\ell_i}, ..., 0\}$ tel que $m_k = m_i$, et $\ell_{i+1} = 0$ s'il n'en existe pas.

Preuve Soit z un bord non vide de vm_i. Comme z est un suffixe non vide, il s'écrit $z = wm_i$ avec w suffixe de v. Comme z est un préfixe propre de vm_i, on a $|\ z\ | \leq |\ v\ |$ donc z est un préfixe de v : on a $w = m_0...m_{|\ w\ |-1}$ et $m_{|\ w\ |} = m_i$. Enfin $|\ w\ | < |\ z\ | \leq |\ v\ |$ donc w est un préfixe et un suffixe propre de v, c'est-à-dire un bord.

Réciproquement, si w est un bord de v avec $m_{|\ w\ |} = m_i$, alors $wm_i = m_0...m_{|\ w\ |}$ est un préfixe de vm_i, c'est un suffixe de vm_i car w est suffixe de v, et il est propre car $|\ w\ | < |\ v\ |$.

La formule s'en déduit avec le lemme précédent, qui énumère les longueurs des bords de v par ordre décroissant.

□

On en déduit le calcul du tableau $[\ell_0, ..., \ell_p]$ par la fonction `bords_maximaux_prefixes` : pour chaque i, on essaie les longueurs de bords $k = \ell_i, \ell_{\ell_i}, ...$ jusqu'à trouver $m_k = m_i$ ou atteindre 0. L'implémentation OCaml fait l'objet du TP ; on donne ici les versions C et Python (cette dernière est celle de l'énoncé du TP) :

C

```c
/* renvoie le tableau [l_0, ..., l_n] des longueurs des bords maximaux
 * des préfixes de mot ; il a été alloué et devra être libéré */
int *bords_maximaux_prefixes(char *mot)
{
    int n = strlen(mot);
    int *longueurs = malloc(sizeof(int) * (n + 1));
```

```
    for (int i = 0; i <= n; i++)
        longueurs[i] = 0;

    for (int i = 1; i < n; i++)
    {
        int k = longueurs[i];
        while (mot[k] != mot[i] && k > 0)
            k = longueurs[k];
        if (mot[k] == mot[i])
            longueurs[i+1] = k + 1;
    }

    return longueurs;
}
```

Python

```
def bords_maximaux_prefixes(mot):
    """Calcule la liste des longueurs des bords maximaux
    des préfixes d'un mot."""
    n = len(mot)
    longueurs = [0] * (n + 1)
    for i in range(1, n):
        k = longueurs[i]
        while mot[k] != mot[i] and k > 0:
            k = longueurs[k]
        if mot[k] == mot[i]:
            longueurs[i + 1] = k + 1
    return longueurs
```

Propriété 42.32

`bords_maximaux_prefixes` effectue $O(p)$ opérations sur un mot de longueur p.

Preuve On utilise une analyse amortie en suivant la valeur de k. Au début de l'itération i, on a $k = \ell_i$. Chaque passage dans la boucle `while` diminue strictement k, car $\ell_k < k$, et à la fin de l'itération $\ell_{i+1} \leq k + 1$. En notant t_i le nombre de passages dans la boucle `while` lors de l'itération i, on a donc $\ell_{i+1} \leq \ell_i - t_i + 1$, soit $t_i \leq \ell_i - \ell_{i+1} + 1$. En sommant, le nombre total de passages est majoré par

$$\sum_{i=1}^{p-1} (\ell_i - \ell_{i+1} + 1) = \ell_1 - \ell_p + p - 1 \leq p - 1$$

car la somme est télescopique et $\ell_1 = 0$. Chaque itération de la boucle `for` effectue par ailleurs un nombre constant d'opérations.

□

Recherche

La recherche procède exactement de la même manière, en lisant la chaîne $s = s_0 \ldots s_{n-1}$ de gauche à droite. On maintient la longueur k du plus long préfixe du motif qui soit suffixe de la partie lue $s_0 \ldots s_{j-1}$. À la lecture de s_j, on cherche à prolonger ce préfixe : si $m_k \neq s_j$, on essaie le bord maximal de $m_0 \ldots m_{k-1}$, de longueur ℓ_k, et ainsi de suite. Si on atteint $k = p$, le motif est trouvé à l'indice $j - p + 1$.

OCaml

```
exception Trouve of int

let recherche_KMP (motif:string) (bords:int array)
    (chaine:string) : int option =
    let n = String.length chaine in
    let p = String.length motif in
    let k = ref 0 in (* longueur du préfixe du motif reconnu *)
    try
        if p = 0 then raise (Trouve 0);
```

```
        for j = 0 to n-1 do
            while !k > 0 && motif.[!k] <> chaine.[j] do
                k := bords.(!k)
            done;
            if motif.[!k] = chaine.[j] then incr k;
            if !k = p then raise (Trouve (j-p+1))
        done;
        None
    with Trouve i -> Some i
```

C

```c
/* cherche motif dans chaine en utilisant le tableau bords renvoyé par
 * bords_maximaux_prefixes(motif). Renvoie l'indice de la première
 * occurrence ou -1 s'il n'est pas présent */
int recherche_KMP(char *motif, int *bords, char *chaine)
{
    int n = strlen(chaine);
    int p = strlen(motif);
    int k = 0; // longueur du préfixe du motif reconnu

    if (p == 0)
        return 0;

    for (int j = 0; j < n; j++)
    {
        while (k > 0 && motif[k] != chaine[j])
            k = bords[k];
        if (motif[k] == chaine[j])
            k++;
        if (k == p)
            return j - p + 1;
    }

    return -1;
}
```

Python

```python
def recherche_KMP(motif, bords, chaine):
    '''Cherche motif dans chaine en utilisant le tableau bords
    renvoyé par bords_maximaux_prefixes(motif)'''
    n, p = len(chaine), len(motif)
    if p == 0:
        return 0
    k = 0 # longueur du préfixe du motif reconnu
    for j in range(n):
        while k > 0 and motif[k] != chaine[j]:
            k = bords[k]
        if motif[k] == chaine[j]:
            k = k + 1
        if k == p:
            return j - p + 1
    return None
```

La correction repose sur le même raisonnement que le théorème précédent : les préfixes du motif qui sont suffixes de $s_0 \ldots s_{j-1}$ sont exactement $m_0 \ldots m_{k-1}$ et ses bords, car deux suffixes d'un même mot dont l'un est plus court sont suffixes l'un de l'autre. Le plus long préfixe du motif suffixe de $s_0 \ldots s_j$ est alors de la forme ws_j où w est le plus long de ces mots tel que $m_{|w|} = s_j$, et c'est bien celui que l'on trouve en essayant les bords par longueur décroissante.

L'analyse amortie est identique : k augmente d'au plus 1 à chaque itération de la boucle `for` et diminue strictement à chaque passage dans la boucle `while`, il y a donc au plus n passages dans celle-ci au total. La recherche est en $O(n)$, et en comptant le prétraitement du motif, l'algorithme de Knuth-Morris-Pratt est en $O(n + p)$ dans le pire cas.

Remarque 42.33

- Contrairement à Boyer-Moore, Knuth-Morris-Pratt n'effectue jamais de saut dans la chaîne : chaque caractère de s est lu une seule fois, au sens où on ne revient jamais en arrière. Comme pour la variante de Rabin-Karp sans vérification, on peut donc l'appliquer à un flux de données. En revanche, sur un texte usuel, Boyer-Moore est souvent plus rapide car il ne lit qu'une partie des caractères.
- Le TP Knuth-Morris-Pratt propose une autre façon d'utiliser le tableau des bords pour la recherche, ainsi que d'autres applications (mots primitifs, préfixes palindromes).
- La valeur k maintenue pendant la recherche est l'état d'un automate déterministe reconnaissant les mots qui se terminent par le motif, d'ensemble d'états $[\![0, p]\!]$. Les transitions sont calculées à la volée à l'aide du tableau des bords plutôt que stockées dans une table de taille $(p+1)\,|\,\Sigma\,|$ (voir le chapitre Langages réguliers et automates).

42.3.5 Extensions à l'analyse d'images

43

TRAVAUX PRATIQUES 43

Compression de Huffman

DANS CE CHAPITRE

43.1 Introduction
43.2 Huffman
43.3 Compression
43.4 Décompression
43.5 Lecture/Écriture dans des fichiers

43.1 Introduction

43.1.1 Objectif

L'objectif de ce TP est de faire un programme `mp2zip.exe` qui nous permettra d'exécuter

- `mp2zip comp fichier` pour compresser un fichier en un fichier `fichier.mp2zip` par Huffman
- `mp2zip decomp fichier.mp2zip` pour décompresser un fichier compressé précédemment par Huffman.

43.1.2 Modules

Pour cela, on va découper le TP en plusieurs sous-parties qui seront codées pour certaines dans des modules.

On a déjà vu les modules en `OCaml` quand on a utilisé par exemple `List`, `Printf` ou encore `Graphics`. Pour simplifier, on va utiliser `dune` ici et en compilant le projet, si on a un fichier `monfichier.ml` on pourra accéder aux fonctions et types définis avec `Monfichier.mafonction`.

Donc, pour vous, tout se passe comme si on travaillait avec plusieurs fichiers `.c` comme on a pu le faire, sauf qu'il n'y a pas besoin de faire de `.h` : tout le contenu est accessible à côté. En utilisant la syntaxe `Module.fonction` on n'a pas besoin de faire l'équivalent de `#include` (c'est `open Module` qui joue ce rôle, mais comme pour les `from module import *` en `Python`, on évite).

Remarque : on pourrait vouloir cacher une partie de l'interface aux appelants, comme on peut le faire avec un `.h` qui ne rend pas tout visible. C'est possible en `OCaml`, on crée pour ça un fichier `.mli` contenant l'interface. Ici, on bénéficie juste d'un mécanisme de `OCaml` pour générer des interfaces maximales par défaut.

Note : il faut créer dans votre répertoire pour le TP :

- un fichier `mp2zip.ml` qui sera le programme principal et qui appellera les fonctions des autres modules
- un fichier `dune` avec le contenu :

```
(executable
    (name mp2zip))
```

- si vous avez une erreur de `PROJECT ROOT`, un fichier `dune-project` contenant la ligne `(lang dune 3.0)`

43.1.3 Tas et files de priorité

On pourra utiliser les fichiers suivants :

`tas.ml` :

```
type 'a t = {
    elements : 'a array;
    mutable taille : int
}

let cree taille_max defaut =
    {
        elements = Array.make taille_max defaut;
        taille = 0
    }

let est_vide tas = tas.taille = 0

let taille tas = tas.taille

let pere a =
    if a = 0
    then failwith "racine"
    else (a+1)/2 - 1

let gauche a = 2*(a+1) - 1
let droite a = 2*(a+1)

let maximum tas = tas.elements.(0)

let nth tas i = tas.elements.(i)

let swap v x y =
    let t = v.(x) in
    v.(x) <- v.(y);
    v.(y) <- t

let ajoute x tas =
    let pos = ref tas.taille in
    tas.elements.(tas.taille) <- x;
    tas.taille <- tas.taille + 1;
    while !pos > 0 && nth tas (pere !pos) < nth tas !pos do
        let p = pere !pos in
        swap tas.elements p !pos;
        pos := p
    done

let max_pere_fils tas a =
    let max_courant = ref a in

    if tas.taille > gauche a
        && nth tas (gauche a) > nth tas !max_courant
    then max_courant := gauche a;
    if tas.taille > droite a
        && nth tas (droite a) > nth tas !max_courant
    then max_courant := droite a;
    !max_courant

let extrait_maximum tas =
    assert (tas.taille > 0);
    tas.taille <- tas.taille - 1;
    let x = tas.elements.(0) in
    if tas.taille > 0
    then begin
        let pos = ref 0 in
        tas.elements.(0) <- tas.elements.(tas.taille);
        while max_pere_fils tas !pos <> !pos do
            let max_fils = max_pere_fils tas !pos in
```

```
            swap tas.elements !pos max_fils;
            pos := max_fils
        done
    end;
    x
```

`filePriorite.ml` :

```
type ('a, 'b) t = ('b * 'a) Tas.t

let cree taille_max defval defprio =
    Tas.cree taille_max (defprio, defval)

let ajoute file x p =
    Tas.ajoute (p, x) file

let retire file =
    let _, x = Tas.extrait_maximum file in
    x
```

✍ **Attention** : `Tas` est un tas **max**, donc `FilePriorite.retire` renvoie l'élément de priorité **maximale**. Pour l'algorithme de Huffman, où l'on veut extraire les deux arbres de plus petites fréquences, on pourra utiliser comme priorité l'opposé de la fréquence.

43.1.4 Au secours, je n'ai pas dune

On peut simuler l'usage de plusieurs fichiers dans un seul en les emballant dans des modules :

```
module Tas = struct
    type 'a t = {
        elements : 'a array;
        mutable taille : int
    }

    let cree taille_max defaut =
        {
            elements = Array.make taille_max defaut;
            taille = 0
        }

    let est_vide tas = tas.taille = 0

    let taille tas = tas.taille

    let pere a =
        if a = 0
        then failwith "racine"
        else (a+1)/2 - 1

    let gauche a = 2*(a+1) - 1
    let droite a = 2*(a+1)

    let maximum tas = tas.elements.(0)

    let nth tas i = tas.elements.(i)

    let swap v x y =
        let t = v.(x) in
        v.(x) <- v.(y);
        v.(y) <- t

    let ajoute x tas =
        let pos = ref tas.taille in
        tas.elements.(tas.taille) <- x;
        tas.taille <- tas.taille + 1;
        while !pos > 0 && nth tas (pere !pos) < nth tas !pos do
            let p = pere !pos in
```

```
            swap tas.elements p !pos;
            pos := p
        done

    let max_pere_fils tas a =
        let max_courant = ref a in

        if tas.taille > gauche a
            && nth tas (gauche a) > nth tas !max_courant
        then max_courant := gauche a;
        if tas.taille > droite a
            && nth tas (droite a) > nth tas !max_courant
        then max_courant := droite a;
        !max_courant

    let extrait_maximum tas =
        assert (tas.taille > 0);
        tas.taille <- tas.taille - 1;
        let x = tas.elements.(0) in
        if tas.taille > 0
        then begin
            let pos = ref 0 in
            tas.elements.(0) <- tas.elements.(tas.taille);
            while max_pere_fils tas !pos <> !pos do
                let max_fils = max_pere_fils tas !pos in
                swap tas.elements !pos max_fils;
                pos := max_fils
            done
        end;
        x
end

module FilePriorite = struct
    type ('a, 'b) t = ('b * 'a) Tas.t

    let cree taille_max defval defprio =
        Tas.cree taille_max (defprio, defval)

    let ajoute file x p =
        Tas.ajoute (p, x) file

    let retire file =
        let _, x = Tas.extrait_maximum file in
        x
end
```

43.2 Huffman

Fichier `huffman.ml`

Ce fichier va contenir les fonctions liées à la création et l'utilisation du code de Huffman.

On va utiliser le type

```
type arbre = Char of int | Node of arbre * arbre
```

pour l'arbre.

On identifie ainsi complètement un caractère à l'octet, et donc l'entier entre 0 et 255, qui le représente.

Les occurrences de caractères seront données dans un tableau `int array` de 256 valeurs, où la ième valeur donne la fréquence du caractère de numéro i. On aura alors des valeurs nulles pour les caractères non présents, elles seront ignorées lorsqu'on va construire l'arbre.

- `construit_arbre : int array -> arbre` qui renvoie l'arbre de Huffman pour un tableau d'occurrences en utilisant l'algorithme glouton.

- `codes : arbre -> bool list array` qui renvoie un tableau de 256 listes de booléens donnant le code de chacun des caractères avec la convention `true = 1 = droite` et `false = 0 = gauche`. On les donnera dans l'ordre de la racine vers la feuille, donc il faudra surement faire un `List.rev` sur l'accumulateur une fois la feuille atteinte.

✍ **Cas limites**

On adopte la même convention que dans le cours sur les algorithmes gloutons :

- si toutes les occurrences sont nulles (texte vide), il n'y a pas d'arbre et `construit_arbre` lève l'exception `Invalid_argument` ;
- si un seul caractère `c` apparaît, `construit_arbre` renvoie `Char c` et le code de `c` est la liste vide.

Vérifier que vos fonctions respectent cette convention. Que vaut alors la compression d'une chaîne comme `"aaaa"` ? Pourquoi ne permet-elle pas, à elle seule, de retrouver la chaîne de départ ?

43.3 Compression

Fichier `compression.ml`

Pour compresser, on va adopter la stratégie suivante :

- on calcule le tableau de fréquences d'une chaîne
- on en déduit son arbre de Huffman
- on calcule les codes de chaque caractère

Ainsi, on va écrire les fonctions :

- `stats : string -> int array` qui renvoie le tableau de fréquences associé à une chaine de caractères passée en argument.
- `compresse : string -> Huffman.arbre * bool list` qui renvoie l'arbre de Huffman et la compression de la chaine passée en argument.

43.4 Décompression

Fichier `decompression.ml`

Pour décompresser, on va adopter la stratégie suivante :

- on lit l'arbre de Huffman
- tant que la liste en entrée n'a pas été intégralement parcourue, on décode un caractère à l'aide de l'arbre et on l'ajoute

Ainsi, on va écrire les fonctions :

- `decompresse : Huffman.arbre * bool list -> string` qui va adopter la stratégie précédente pour renvoyer la chaîne décompressée.

✍ **Remarque 43.1** D'après la question sur les cas limites, quand un seul caractère apparaît, la liste de bits est vide et la boucle précédente ne produit rien. On transmet donc aussi la **longueur** de la chaîne d'origine : `compresse` renvoie un triplet `(longueur, arbre, bits)` et `decompresse` décode exactement `longueur` caractères (avec l'arbre `Char c`, chaque caractère est décodé sans lire aucun bit). Pour la chaîne vide, on renvoie une longueur nulle et `decompresse` doit renvoyer `""` sans consulter l'arbre : on pourra utiliser une valeur arbitraire, par exemple `Char 0`, ou un type `option`.

43.5 Lecture/Écriture dans des fichiers

43.5.1 Écrire/Lire des octets

On va avoir besoin d'ouvrir les fichiers en lecture **binaire** pour cela.

> **Remarque 43.2** La distinction `binaire/texte` dépend des plateformes. Il s'agit notamment de `Windows` qui traite différemment les sauts de ligne depuis les années 80 et qui oblige à pratiquer cette distinction.
>
> En `OCaml`, les interfaces de fichiers sont différentes et la séparation est plus profonde.

On dispose ainsi de deux fonctions

- `open_out_bin : string -> out_channel`
- `open_in_bin : string -> in_channel`

pour ouvrir un fichier en mode binaire en écriture ou en lecture.

Pour écrire un octet, un `int` entre 0 et 255 en `OCaml`, on va utiliser :

- `output_byte : out_channel -> int -> unit`

Pour lire un octet :

- `input_byte : in_channel -> int`

Les fichiers sont toujours fermés au moment où la valeur est supprimée de la mémoire par le ramasse-miettes, mais on peut forcer la fermeture avec `close_out` et `close_in`. Pour un fichier en écriture, c'est une manière de s'assurer que tout a bien été écrit.

43.5.2 Sérialisation d'arbres binaires

On parle de sérialisation d'une structure de donnée quand il s'agit de l'écrire dans un fichier car on la présente alors sous la forme d'une série d'octets.

Pour sérialiser un arbre binaire strict, on peut remarquer qu'il est uniquement caractérisé par l'ordre préfixe de ses nœuds et feuilles.

On va supposer qu'on écrit dans le fichier

- `0<x><y>` pour représenter `Node(x,y)` où `<x>` est la sérialisation de `x`.
- `1<c>` pour représenter `Char c`

Exemple Écrire l'arbre dont la sérialisation est `0 1 0 0 1 1 1 2` (chaque chiffre représentant un octet).

En déduire deux fonctions :

- `output_arbre : out_channel -> arbre -> unit` qui écrit un arbre dans un fichier ouvert en binaire
- `input_arbre : in_channel -> arbre` qui lit un arbre

43.5.3 Compression au niveau octet

Modifier les fonctions précédentes pour que les données soient lues depuis un fichier puis écrites dans un fichier.

Pour réaliser les statistiques puis compresser, il va être nécessaire de relire le fichier, pour cela on peut utiliser :

- `seek_in inc 0` qui revient au début du fichier associé à `inc : in_channel`

Votre programme `mp2zip` devrait maintenant pouvoir être appelé ainsi :

- `mp2zip comp fichier` va produire un `fichier.mp2zip` compressé
- `mp2zip decomp fichier.mp2zip` va produire un `fichier.out` décompressé (un nom différent de fichier pour ne pas écraser en faisant des tests)

> **Remarque 43.3** On pourra modifier les interfaces précédentes ainsi :
>
> - `stats : in_channel -> int array`
> - `compresse : in_channel -> out_channel -> unit`
> - `decompresse : in_channel -> out_channel -> unit`
>
> Pour lire un fichier jusqu'au bout on peut utiliser la structure :

```
try
    while true do
       let b = input_byte inc in
       ....
    done;
    failwith "Impossible"
with End_of_file -> ...
```

43.5.4 Compression au niveau bit

La compression précédente va faire augmenter sensiblement la taille des fichiers car on prend 1 octet pour 1 bit de données.

Adapter le code précédent pour regrouper les bits par octet, c'est-à-dire par paquet de 8.

Remarque 43.4 Ce n'est pas si simple car il va falloir temporiser les écritures et conserver des données en lecture pour ne pas passer à l'octet suivant.

Il y a également un problème pour les bits finaux : si jamais la taille n'est pas un multiple de 8, le dernier octet ne sera pas rempli entièrement, il faut donc un mécanisme pour gérer cela.

Écrire la longueur du fichier d'origine en tête du fichier compressé, comme suggéré plus haut, résout ce problème : on arrête la décompression dès que ce nombre de caractères a été produit. Penser à tester avec un fichier vide et un fichier ne contenant qu'un seul caractère répété.

PARTIE V

Système

44 Gestion de la mémoire dans un programme compilé 661

CHAPITRE 44

Gestion de la mémoire dans un programme compilé

DANS CE CHAPITRE

44.1 Organisation de la mémoire
44.2 Pointeurs
44.3 Portée d'un identificateur
44.4 Piles d'exécution, variables locales et paramètres
44.5 Allocation dynamique

Note **Roadmap** :

- reprendre les exemples en `C` en utilisant des `int` suite au changement dans le programme.

Remarque 44.1 Ce chapitre se concentre sur la manière dont un programme compilé gère la mémoire. Il est question, en particulier, de la notion de variable. Le modèle dans lequel on se place est celui du langage C où une variable est un emplacement mémoire.

44.1 Organisation de la mémoire

Note Ici, je fais le choix d'une présentation assez informelle pour ne pas qu'elle soit trop liée à la réalité d'un compilateur en particulier.

Un programme compilé gère la mémoire d'un ordinateur de deux manières très différentes

- statiquement : c'est le cas des variables locales ou globales définies dans le programme. Au moment de la compilation, le compilateur dispose de l'information suffisante pour prévoir de la place en mémoire pour stocker ces données.
- dynamiquement : c'est le cas des objets dont la taille n'est connue qu'à l'exécution et peut varier selon l'état du programme. C'est alors au moment de l'exécution que le programme va faire une demande d'allocation pour obtenir une place mémoire.

En termes d'allocation statique, on peut distinguer plusieurs types de mémoire :

- les variables globales initialisées qui seront stockées dans le binaire et placées en mémoire dans une zone spécifique chargée avec le binaire
- les variables globales non initialisées dont seule la déclaration sera dans le binaire et qui seront allouées, placées en mémoire et initialisées à 0 au moment du chargement du binaire
- les variables locales et les paramètres qui sont placés dans une pile afin de les allouer uniquement au moment de l'exécution du bloc ou de la fonction

L'allocation dynamique utilise une zone mémoire appelée tas dont une possible organisation est développée dans la partie Réalisation d'un système d'allocation de mémoire.

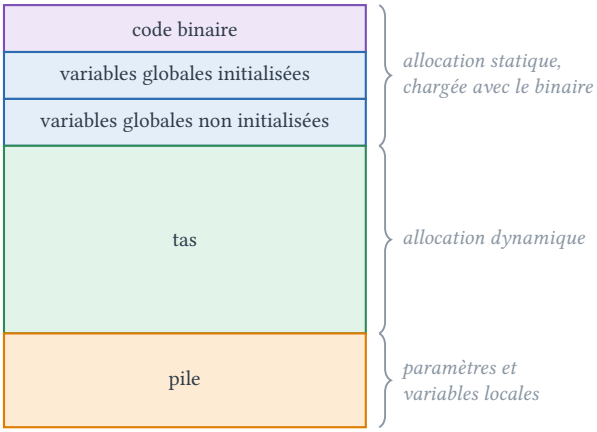


Figure 44.1 — Structure de la mémoire associée à un programme

Considérons le programme C suivant.

```c
const int a  = 42;
int b[] = { 1, 2, 3 };
int c;

int f(int x, int y)
{
    int z = x;
    z = z * y;
    return z;
}

int main(int argc, char **argv)
{
    const int d = 1664;

    c = f(a, d);

    return 0;
}
```

Il est possible d'observer la manière dont sa mémoire sera répartie en utilisant la commande `objdump` :

```
$ gcc -c memoire.c
$ objdump -x memoire.o

memoire.o:     file format elf64-x86-64
memoire.o
architecture: i386:x86-64, flags 0x00000011:
HAS_RELOC, HAS_SYMS
start address 0x0000000000000000

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .text         00000053  0000000000000000  0000000000000000  00000040  2**0
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         0000000c  0000000000000000  0000000000000000  00000094  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000004  0000000000000000  0000000000000000  000000a0  2**2
                  ALLOC
  3 .rodata       00000004  0000000000000000  0000000000000000  000000a0  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .comment      00000013  0000000000000000  0000000000000000  000000a4  2**0
                  CONTENTS, READONLY
  5 .note.GNU-stack 00000000  0000000000000000  0000000000000000  000000b7  2**0
                  CONTENTS, READONLY
  6 .eh_frame     00000058  0000000000000000  0000000000000000  000000b8  2**3
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA
```

```
SYMBOL TABLE:
0000000000000000 l    df *ABS*  0000000000000000 memoire.c
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l    d  .rodata    0000000000000000 .rodata
0000000000000000 l    d  .note.GNU-stack    0000000000000000 .note.GNU-stack
0000000000000000 l    d  .eh_frame  0000000000000000 .eh_frame
0000000000000000 l    d  .comment   0000000000000000 .comment
0000000000000000 g     O .rodata    0000000000000004 a
0000000000000000 g     O .data  000000000000000c b
0000000000000000 g     O .bss   0000000000000004 c
0000000000000000 g     F .text  000000000000001f f
000000000000001f g     F .text  0000000000000034 main


RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000042 R_X86_64_PLT32    f-0x0000000000000004
0000000000000048 R_X86_64_PC32     c-0x0000000000000004


RELOCATION RECORDS FOR [.eh_frame]:
OFFSET           TYPE              VALUE
0000000000000020 R_X86_64_PC32     .text
0000000000000040 R_X86_64_PC32     .text+0x000000000000001f
```

On retrouve dans les sections mémoire :

- `.text` contenant le programme binaire
- `.data` contenant les variables globales initialisées et non constantes
- `.bss` contenant les variables globales non initialisées (ou initialisées à 0)
- `.rodata` contenant les variables globales initialisées mais constantes.

Ici, le schéma mémoire est un peu plus compliqué car une zone mémoire distincte est prévue pour les variables constantes initialisées pour des raisons de sécurité.

La pile et le tas sont automatiques et n'ont pas besoin de figurer dans le binaire, c'est pour cela qu'on ne les trouve pas dans la liste.

Dans cette description mémoire, on trouve la table des symboles qui décrit où vont se trouver en mémoire certaines variables.

Nom de variable	Sorte de déclaration	Section mémoire
a	constante globale initialisée	`.rodata`
b	globale initialisée	`.data`
c	globale non init.	`.bss`

La section suivante permettra de compléter le tableau en étudiant comment les paramètres et variables locales sont gérés. On remarque cependant qu'il n'y a pas de symboles pour ceux-ci. En effet, le nom des variables locales est a priori perdu après la compilation contrairement aux variables globales.

44.2 Pointeurs

Dans le modèle du C, une variable est un emplacement mémoire, et chaque emplacement mémoire possède une **adresse** : c'est le numéro de son premier octet. Une adresse est donc un simple entier, mais le C lui associe un type pour savoir comment interpréter ce qui s'y trouve.

Définition 44.2

Un **pointeur** est une valeur qui représente une adresse mémoire. Si `T` est un type, le type `T *` est celui des pointeurs vers une valeur de type `T`.

- Si `x` est une variable de type `T`, l'expression `&x` est l'adresse de `x`, de type `T *`.
- Si `p` est de type `T *`, l'expression `*p` désigne l'emplacement mémoire d'adresse `p`, vu comme une valeur de type `T`. On dit qu'on **déréférence** `p`.

```c
#include <stdio.h>

int main(void)
{
    int x = 3;
    int *p = &x; // p contient l'adresse de x

    printf("adresse de x : %p, valeur pointée : %d\n", (void *)p, *p);
    *p = 4; // on écrit à l'adresse contenue dans p, donc dans x
    printf("x vaut maintenant %d\n", x);

    return 0;
}
```

Ce programme affiche par exemple :

```
adresse de x : 0x16bc467e0, valeur pointée : 3
x vaut maintenant 4
```

L'adresse affichée change d'une exécution à l'autre : `x` est une variable locale, placée dans la structure pile de `main`.

Remarque 44.3

- L'étoile joue deux rôles différents : dans une déclaration `int *p`, elle fait partie du type ; dans une expression `*p`, c'est l'opérateur de déréférencement. Attention, dans `int *p, q;` seule `p` est un pointeur, `q` est un `int`.
- `*p` peut être placé à gauche d'une affectation : `*p = 4` modifie la mémoire pointée, pas `p` lui-même.
- Un pointeur est lui-même une variable, qui occupe 8 octets sur une architecture 64 bits, quel que soit le type pointé. On peut donc prendre son adresse : `&p` est de type `int **`.
- La valeur spéciale `NULL` (l'adresse 0) indique un pointeur qui ne pointe sur rien. La déréférencer est une erreur, voir la partie Protection mémoire.

44.2.1 Passage par adresse

En C, les paramètres d'une fonction sont passés **par valeur** : comme on l'a vu dans la partie Piles d'exécution, l'appel copie la valeur des arguments dans la structure pile de la fonction appelée. Modifier un paramètre ne modifie donc que cette copie.

Pour qu'une fonction puisse modifier une variable de l'appelant, on lui passe l'adresse de cette variable : c'est le **passage par adresse**. L'exemple classique est l'échange de deux variables.

```c
void echange_faux(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}

void echange(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

int main(void)
{
```

```
    int x = 1;
    int y = 2;

    echange_faux(x, y); // x = 1, y = 2 : seules les copies ont été échangées
    echange(&x, &y);    // x = 2, y = 1
    return 0;
}
```

Lors de l'appel `echange(&x, &y)`, la structure pile de `echange` contient deux paramètres `a` et `b` qui sont des copies des **adresses** de `x` et de `y`. Ces adresses désignent des emplacements de la structure pile de `main` : `*a = *b` écrit donc directement dans `x`.

> **Remarque 44.4** C'est aussi ce mécanisme qui permet à une fonction de renvoyer plusieurs résultats, ou d'éviter de recopier une grosse structure en passant son adresse. On le retrouve dans `scanf("%d", &n)`, qui doit écrire dans `n`.

44.2.2 Pointeurs et tableaux

Les éléments d'un tableau sont rangés de manière contiguë en mémoire. Dans presque toutes les expressions, un tableau `t` est converti automatiquement en l'adresse de sa première case `&t[0]`.

On peut ajouter un entier à un pointeur : si `p` est de type `T *`, alors `p + i` est l'adresse située `i` cases de type `T` plus loin, c'est-à-dire `i * sizeof(T)` octets plus loin. C'est l'**arithmétique des pointeurs**. Par définition, `t[i]` est exactement `*(t + i)`.

```
#include <stdio.h>

int somme(int *t, int n)
{
    int s = 0;
    for (int i = 0; i < n; i++) {
        s += t[i]; // identique à s += *(t + i);
    }
    return s;
}

int main(void)
{
    int t[4] = { 1, 2, 3, 4 };
    int *p = t; // t est converti en &t[0]

    printf("%d %d %d\n", *p, *(p + 2), p[3]);        // 1 3 4
    printf("%p %p\n", (void *)p, (void *)(p + 1));   // adresses distantes de 4
    printf("somme : %d\n", somme(t, 4));             // 10
    printf("somme de la fin : %d\n", somme(t + 2, 2)); // 7
    printf("%zu %zu\n", sizeof(t), sizeof(p));       // 16 8

    return 0;
}
```

> **Remarque 44.5**
>
> - Passer un tableau à une fonction revient donc à passer l'adresse de sa première case : le tableau n'est pas copié et la fonction peut le modifier. En revanche, sa taille est perdue, c'est pourquoi on passe toujours la longueur `n` en paramètre supplémentaire.
> - Dans `somme`, `sizeof(t)` vaudrait 8, la taille d'un pointeur, et non la taille du tableau. C'est une erreur classique.
> - L'appel `somme(t + 2, 2)` montre qu'on peut faire travailler une fonction sur une portion de tableau sans rien recopier.
> - Un tableau n'est pas pour autant un pointeur : on ne peut pas écrire `t = p`, et `sizeof(t)` dans `main` donne bien 16 octets.

- Rien ne vérifie qu'un accès `t[i]` reste dans les bornes du tableau : un débordement lit ou écrit silencieusement dans une autre zone mémoire.

44.2.3 Pointeurs vers le tas

Les pointeurs sont indispensables pour l'allocation dynamique, présentée dans la partie suivante : la fonction `malloc` renvoie l'adresse d'une zone du tas, et la seule manière d'y accéder est de conserver ce pointeur. Contrairement à une variable locale, cette zone survit à la fin de la fonction qui l'a allouée, jusqu'à l'appel de `free`.

```c
#include <assert.h>
#include <stdlib.h>

int *cree_tableau(int n, int v)
{
    int *t = malloc(n * sizeof(int));
    assert(t != NULL);
    for (int i = 0; i < n; i++) {
        t[i] = v;
    }
    return t;
}

int main(void)
{
    int *t = cree_tableau(5, 7);
    t[0] = 1;
    free(t); // c'est à l'appelant de libérer la zone
    t = NULL;
    return 0;
}
```

44.2.4 Erreurs classiques

Le compilateur ne vérifie pas qu'un pointeur désigne une zone valide. Les erreurs suivantes produisent un comportement *indéterminé* : le programme peut planter, afficher des valeurs absurdes ou sembler fonctionner normalement, ce qui les rend difficiles à détecter.

- **Pointeur pendant** (*dangling pointer*) : un pointeur vers une zone qui n'existe plus. C'est le cas d'un pointeur vers une variable locale d'une fonction terminée, dont la structure pile a été dépilée :

```c
int *adresse_locale(void)
{
    int x = 42;
    return &x; // x disparaît avec la structure pile de l'appel
}
```

  Le compilateur produit ici un avertissement. C'est aussi le cas d'un pointeur vers une zone du tas après son `free`, ce qui est plus sournois quand plusieurs pointeurs désignent la même zone :

```c
#include <stdlib.h>

int main(void)
{
    int *t = malloc(3 * sizeof(int));
    int *u = t; // u et t pointent sur la même zone
    free(t);
    u[0] = 1;   // u est un pointeur pendant
    return 0;
}
```

- **Double libération** (*double free*) : appeler deux fois `free` sur la même adresse, ou appeler `free` sur une adresse qui ne provient pas de `malloc`. Des exemples sont donnés dans la partie Libération. Dans le système d'allocation réalisé plus loin, ce sont les `assert` de la fonction `liberation` qui détectent ces erreurs.

- **Fuite mémoire** : perdre le dernier pointeur vers une zone allouée avant de l'avoir libérée. La zone ne pourra plus jamais être libérée.

```
int *t = malloc(3 * sizeof(int));
t = malloc(5 * sizeof(int)); // la première zone n'est plus accessible
free(t);
```

Remarque 44.6

- Une bonne habitude est d'affecter `NULL` à un pointeur juste après l'avoir libéré : une utilisation ultérieure provoque alors une erreur immédiate au lieu d'un comportement indéterminé, et `free(NULL)` ne fait rien.
- L'option de compilation `-fsanitize=address` de `gcc` et `clang` ajoute des vérifications à l'exécution. Le programme précédent s'arrête alors avec le message `ERROR: AddressSanitizer: heap-use-after-free` qui indique la ligne fautive. Sous Linux, elle signale aussi les fuites mémoire à la fin de l'exécution. Il est vivement conseillé de l'utiliser pendant la mise au point des programmes.

44.3 Portée d'un identificateur

En ce qui concerne une variable dans un programme, on peut définir deux notions d'apparence assez similaire.

D'une part la portée d'un identificateur qui correspond au texte du programme :

Définition 44.7

La **portée** d'un identificateur est définie par la zone du texte d'un programme dans laquelle il est possible d'y faire référence sans erreurs.

Remarque 44.8 Dans le langage C, un identificateur peut être utilisé dès sa déclaration mais tant que la variable n'est pas initialisée, le comportement n'est pas spécifié et il faut considérer cela comme une erreur. Le compilateur produit ainsi un avertissement quand on utilise le paramètre `-Wall`.

Dans le cas d'une définition, il est ainsi possible de faire référence à l'identificateur dans l'expression de son initialisation : `int x = x`. Ce cas est pathologique et le fait qu'on compte la ligne de déclaration dans la portée ne devrait pas inciter à écrire ce genre de code qui produira, de toute façon, une erreur avec les options `-Wall -Werror` sous `clang` (sous `gcc`, il faut ajouter l'option `-Winit-self`).

Dans le programme :

```
int a = 1;

int f (int x)
{
    int y = x + a;
    return y;
}

int g()
{
    int z = 3;
    return z + f(z);
}
```

La portée des identificateurs est :

Identificateur	Portée
a	1-13
x	3-7
y	5-7
f	3-13
g	9-13
z	11-13

Pour une fonction, afin de pouvoir écrire des fonctions récursives, l'identificateur est utilisable dès la fin de son en-tête, donc en particulier dans le corps de la fonction.

Comme la portée est une notion associée aux identificateurs, elle est indépendante de la notion de variables. Si on considère le programme suivant :

```
int f()
{
    int i = 3;

    return i;
}

int g()
{
    int i = 5;

    return i+1;
}
```

L'identificateur `i` a pour portée les lignes 3-6 et 10-13.

Un autre phénomène plus complexe peut se produire quand on redéfinit un identificateur dans sa portée.

Considérons le programme suivant

```
int f()
{
    int i = 3;

    for (int j = 0; j < 3; j++)
    {
        int i = 4;

        i += j;
    }

    return i;
}
```

Ici, l'identificateur `i` a pour portée les lignes 3-13 mais dans les lignes 7-10 il y a un phénomène dit de masquage où la première définition est cachée par la seconde.

L'identificateur associé à une variable globale a pour portée l'ensemble des lignes allant de sa déclaration à la fin du fichier.

> **Remarque 44.9** En C, la portée d'un identificateur est **statique** : elle dépend uniquement du texte du programme au moment de la compilation.
>
> En Python, la portée d'un identificateur est également déterminée par le texte du programme, mais le fait qu'un identificateur soit effectivement défini à un endroit donné peut dépendre de l'exécution. Par exemple si on considère le programme Python

```
if condition:
    x = 3
```

Utiliser `x` après ces lignes provoque une erreur (`NameError`) ou non selon que la `condition` est réalisée ou non : cette vérification n'a lieu qu'à l'exécution.

44.4 Piles d'exécution, variables locales et paramètres

On a vu qu'en raison de leur durée de vie, les variables globales étaient allouées dès le chargement du programme. Pour les variables locales ainsi que la mécanique des appels, on utilise une **pile**.

Cette pile d'exécution est représentée en mémoire par un tableau et un indicateur de sommet de pile (le *pointeur de pile*).

Le remplissage de ce tableau s'effectue souvent des adresses hautes vers les adresses faibles : on empile en faisant diminuer les adresses.

Remarque 44.10 En fait, il existe des architectures où les adresses sont croissantes. Ce qui importe est que le tas et la pile aient des comportements opposés pour qu'ils puissent grandir dans la même zone mémoire.

Un compilateur peut faire le choix d'utiliser directement des registres processeurs pour les variables locales ou pour passer des paramètres à une fonction. Ici, pour simplifier, on va supposer que ce n'est pas le cas et que tout passe par la pile d'exécution.

Remarque 44.11 Afin de pouvoir appeler une fonction dans une bibliothèque potentiellement compilée avec un autre compilateur, il est nécessaire d'avoir une convention d'appels de fonctions. Une telle convention est appelée une *interface applicative binaire* (Application Binary Interface).

La convention `System V AMD64 ABI` qui est celle de Linux et macOS sur des architectures x86 64 bits consiste à utiliser des registres entiers pour les six premiers arguments entiers ou pointeurs et des registres flottants pour les huit premiers arguments flottants. Les arguments suivants sont alors passés sur la pile (donc dès le septième entier/pointeur ou neuvième flottant).

La convention `cdecl` qui est assez répandue sur les architectures x86 32 bits consiste à utiliser la pile. Par contre la valeur de retour est présente dans des registres comme pour la convention `System V AMD64 ABI`.

Lors d'un appel d'une fonction passant par la pile, on commence par empiler les paramètres (souvent de la droite vers la gauche) puis on empile l'adresse à laquelle doit revenir l'exécution une fois que la fonction aura terminé son exécution.

Au début de l'exécution de cette fonction, on sauvegarde sur la pile l'adresse de base de la structure pile de l'appelant (registre `ebp` en 32 bits), puis on déplace le sommet de pile pour réserver de la place pour les variables locales.

L'empreinte sur la pile d'un appel de fonction est appelée une structure pile (**stack frame** en anglais). La pile est alors organisée, depuis l'appel au point d'entrée du programme, par empilement et dépilement de structures piles.

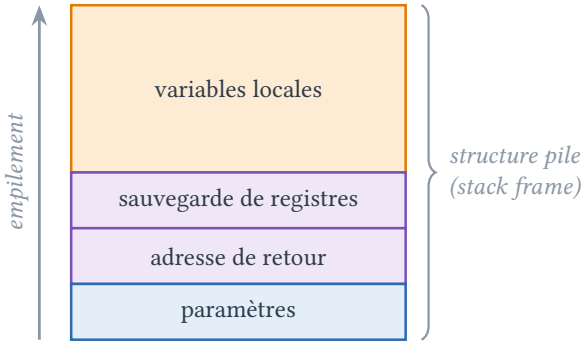


Figure 44.2 — Structure pile associée à un appel de fonction

Voici un exemple possible de l'état de la pile d'exécution lors de l'exécution d'un programme compilé avec la norme `cdecl` (Il suffit d'ajouter l'argument `-m32` pour compiler en 32 bits). Sur cette figure, le sommet de pile (adresses les plus faibles) est en haut et la structure pile de `main` est en dessous.

```
int f(int a, int b)
{
    int c = 3;
    /* pile ici après l'appel en l13 */
    c = c + b;
    c = c * a;
    return c;
}

int main()
{
    int x = f(10,12);
}
```

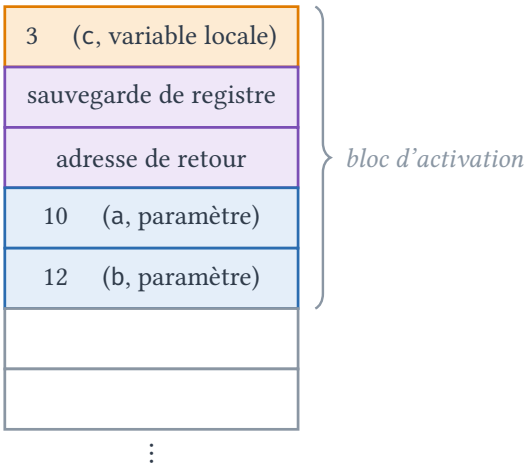


Figure 44.3 — Bloc d'activation de l'appel f(10,12) sur la pile

À chaque appel de fonction, on va donc empiler une structure de pile complète, puis la dépiler à la sortie. Ce mécanisme est essentiel pour permettre la récursivité car il permet d'effectuer plusieurs appels d'une même fonction sans risquer que la mémoire utilisée lors d'un des appels interfère avec un autre. On comprend également les limites de la récursivité ici car cet empilement successif de structures de piles peut dépasser la taille maximale de la pile d'exécution : on parle alors de *dépassement de pile* ou **stack overflow** en anglais.

> **Remarque 44.12** Il est possible de définir des variables locales qui soient situées au même emplacement mémoire pour tous les appels d'une fonction, en les déclarant avec le mot-clé `static` : on parle de variables locales *statiques*.

Ce mécanisme est essentiellement géré comme les variables globales et il ne sera pas développé dans la suite.

Note Je me demande s'il faudrait parler plus précisément de la manière dont la pile est gérée avec les registres ebp/esp. Mais ça ne me semble pas apporter grand chose ici.

44.5 Allocation dynamique

Comme cela a été vu dans la section Organisation de la mémoire, il est possible d'allouer dynamiquement de la mémoire. Pour gérer cette allocation dynamique, on passe par une zone mémoire appelée le *tas* ainsi que par un mécanisme d'allocation et de libération de mémoire au niveau du système.

Pour l'utilisateur, cette gestion interne est transparente et on peut se contenter de considérer qu'il y a deux mécanismes :

- l'**allocation** mémoire où on demande à ce qu'une zone mémoire d'une certaine taille soit allouée
- la **libération** mémoire où on signale que la zone mémoire peut être récupérée par le système.

Naturellement, la mémoire d'un ordinateur étant finie, il est très important de libérer au plus tôt la mémoire non utilisée pour éviter d'épuiser la mémoire. Quand un programme ne libère pas toute la mémoire qu'il alloue, on parle de **fuite mémoire**. L'empreinte mémoire d'un tel programme peut alors croître jusqu'à rendre le programme ou le système inutilisable.

44.5.1 Allocation

Pour allouer une zone mémoire, on utilise la fonction `malloc` dans `stdlib.h` de signature :

```
void *malloc(size_t size)
```

Ici `size` indique le nombre d'**octets** à allouer et la fonction renvoie un pointeur vers le premier octet alloué. Comme la fonction ne connaît pas le type d'objets alloués, on utilise ainsi le type `void *`.

Ce type joue un rôle spécial : une valeur de type `void *` est convertie implicitement vers n'importe quel type pointeur, on peut donc affecter directement la valeur de retour sans rien avoir à écrire d'autre que l'appel à `malloc` :

```
char *t = malloc(n);
```

Remarque 44.13 Dans le langage C++ qui peut être vu comme un successeur de C, il est obligatoire de préciser ici le nouveau type à l'aide d'un **transtypage**. Pour convertir la valeur `x` vers le type `char *` on écrit alors `(char *)x`. Ainsi, pour allouer un tableau de `n` caractères, on utilisera :

```
char *t = (char *) malloc(n);
```

Bien que ce ne soit pas nécessaire en C, il est fréquent de rencontrer des programmes présentant de tels transtypages qui sont superflus mais corrects syntaxiquement en C.

Pour obtenir la taille à allouer, il peut être utile d'utiliser l'opérateur `sizeof` qui prend en entrée un type ou une valeur et renvoie sa taille. Ainsi on peut allouer un tableau de n entiers non signés ainsi :

```
unsigned int *t = malloc( sizeof(unsigned int) * n );
```

et cet appel ne dépend pas de la taille prise par un `unsigned int` sur l'architecture.

Une autre raison de l'utilisation de `sizeof` est l'extensibilité. Par exemple, si on a un `struct point` représentant des points en 2D :

```
struct point {
    float x;
    float y;
};
```

on peut allouer un tableau de n points ainsi :

```
struct point *t = malloc( sizeof(struct point) * n );
```

Si jamais on change la structure pour représenter des points en 3D ainsi :

```
struct point {
    float x;
    float y;
    float z;
};
```

il sera inutile de changer le code d'allocation du tableau car `sizeof(struct point)` tiendra compte automatiquement du changement.

Si jamais une erreur empêche d'allouer la mémoire - ce qui peut être le cas s'il n'y a plus de mémoire disponible - le pointeur renvoyé par `malloc` a la valeur spéciale `NULL`.

> **Remarque 44.14** La zone mémoire renvoyée par `malloc` n'est pas initialisée. On ne peut pas supposer qu'elle soit remplie de la valeur 0. Il faut donc manuellement initialiser la mémoire après le retour de `malloc`.

44.5.2 Libération

Pour libérer la mémoire, on utilise la fonction `free` également présente dans `stdlib.h` et dont la signature est :

```
void free(void *ptr);
```

> **Remarque 44.15** Il est très important d'utiliser uniquement un pointeur obtenu précédemment par un appel à `malloc` (et pas encore libéré) et de ne pas l'utiliser plus d'une fois.
>
> Le programme suivant provoque une erreur `free(): invalid pointer` à l'exécution mais est détecté par un avertissement du compilateur : `warning: attempt to free a non-heap object 'a'`.
>
> ```
> #include <stdlib.h>
>
> int main()
> {
>     int a;
>     free(&a);
>     return 0;
> }
> ```
>
> Le programme suivant alloue un tableau de deux `char` et appelle `free` sur l'adresse de la seconde case. En faisant cela, il n'y a pas d'avertissement car on appelle `free` sur un objet qui est effectivement sur le tas. On obtient alors à nouveau une erreur à l'exécution `free(): invalid pointer`.
>
> ```
> #include <stdlib.h>
>
> int main()
> {
>     char *a = malloc(2);
>     free(&(a[1]));
>     return 0;
> }
> ```
>
> Le programme suivant libère deux fois la mémoire et provoque l'erreur `free(): double free detected in tcache 2` à l'exécution.
>
> ```
> #include <stdlib.h>
>
> int main()
> {
> ```

```
    char *a = malloc(2);
    free(a);
    free(a);
    return 0;
}
```

44.5.3 Protection mémoire

Comme on l'a vu dans la première partie, quand un programme s'exécute il a un environnement mémoire constitué de plusieurs zones, parfois appelées **segments**, avec le droit d'écriture dans certaines d'entre elles.

Le système d'exploitation protège ainsi la mémoire et, au niveau matériel, l'unité de gestion de la mémoire connait les adresses accessibles à un programme. En cas d'accès anormal, le matériel provoque une erreur qui remonte au système d'exploitation qui termine l'exécution du programme avec une erreur souvent intitulée `Segmentation fault`.

Voici quelques exemples commentés produisant des erreurs de type `segmentation fault` à l'exécution.

Lecture à l'adresse 0 (pointeur `NULL`), ce qui provoque en pratique toujours une erreur sur les systèmes usuels. Même problème avec une adresse inaccessible ou invalide.

```
int main()
{
    int *a = 0;

    return a[0];
}
```

Écriture dans une zone en lecture seule comme le segment du code.

```
int main()
{
    int *a = (int*) (&main);
    // a pointe sur le corps de la fonction main

    *a = 0;

    return 0;
}
```

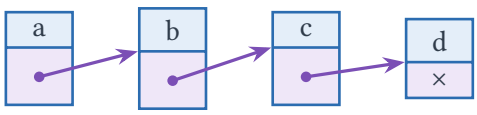
44.5.4 Réalisation d'un système d'allocation de mémoire

Note Prérequis : listes chaînées

Afin de comprendre comment fonctionne le tas, et en particulier malloc et free, on va simuler ici ce comportement en allouant une grande plage de mémoire avec malloc et en gérant le découpage et l'allocation de celle-ci.

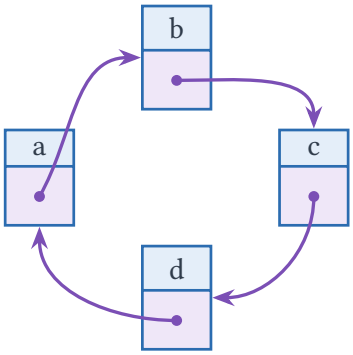
Pour gérer les blocs mémoires libres, on utilise une liste circulaire. Une liste circulaire est une liste chaînée avec un lien supplémentaire entre le premier et le dernier maillon, ce qui fait qu'on peut considérer n'importe quel maillon comme étant la *tête* de la liste.

Une liste chaînée :



Ici le dernier maillon comprend un pointeur qui ne pointe sur rien indiqué par ×, en pratique il a la valeur `NULL`.

Une liste circulaire :



Le seul changement est donc de faire pointer le dernier maillon sur le premier. Le fait d'avoir préservé les valeurs dans les maillons permet ici de voir ce qu'est devenu le premier maillon, mais on peut accéder à cette liste par n'importe lequel de ces maillons.

Les nœuds de la liste circulaire des blocs auront pour valeur un triplet `(adresse,taille,libre)` qui indique qu'à l'adresse `adresse` il y a un bloc de `taille` octets et le booléen `libre` indique si ce bloc a été alloué ou non.

Pour cela on commence par définir une structure `bloc` et une fonction de création d'un bloc :

```
struct bloc {
    void *adresse;
    uint32_t taille;
    bool libre;
    struct bloc *suivant;
};

struct bloc *cree_bloc(void *adresse, uint32_t taille, bool libre)
{
    struct bloc *b = malloc(sizeof(struct bloc));
    b->adresse = adresse;
    b->taille = taille;
    b->libre = libre;
    return b;
}
```

✍ Ce code nécessite d'inclure `stdlib.h`, `stdint.h` (pour `uint32_t`) et `stdbool.h` (pour `bool`), ainsi que `assert.h` et `stdio.h` pour la suite.

On définit ensuite deux variables globales :

- `blocs_libres` qui va pointer sur un maillon de la liste circulaire des blocs
- `plage_memoire` qui pointe sur l'adresse de la plage mémoire que l'on va gérer et servira à la libérer en sortie de programme.

```
struct bloc *blocs_libres;
void *plage_memoire;
```

La fonction `creation_blocs_libres` permet de créer la liste circulaire avec un premier bloc qui pointe sur lui-même et qui correspond à l'adresse que l'on va placer dans `plage_memoire`.

```
void creation_blocs_libres(uint32_t taille_bloc_initial)
{
    plage_memoire = malloc(taille_bloc_initial);
    blocs_libres = cree_bloc(plage_memoire, taille_bloc_initial, true);
    blocs_libres->suivant = blocs_libres; // boucle initiale
}
```

Pour libérer la liste à la sortie du programme, on définit la fonction `destruction_blocs_libres` qui présente ainsi le parcours usuel d'une liste circulaire : on procède comme pour une liste chaînée classique mais, au

lieu d'utiliser un test `bloc_courant->suivant == NULL` pour l'arrêt, il faut se souvenir du premier bloc et tester pour voir si on est revenu au point de départ. On n'oublie pas de libérer l'espace `plage_memoire` à la fin.

```
void destruction_blocs_libres()
{
    struct bloc *premier_bloc = blocs_libres;
    struct bloc *bloc_courant = premier_bloc;

    // on boucle pour libérer chaque maillon
    while (true) {
        struct bloc *bloc_suivant = bloc_courant->suivant;
        free(bloc_courant);
        if (bloc_suivant == premier_bloc) break;
        bloc_courant = bloc_suivant;
    }

    // on libère la plage mémoire initiale
    free(plage_memoire);
}
```

Pour allouer t octets, on parcourt la liste des blocs jusqu'à trouver un bloc b libre de taille $b.t$ telle que $b.t \geq t$. Si un tel bloc n'existe pas, on renvoie le pointeur `NULL` signe d'un échec d'allocation. Sinon, on indique que le bloc est occupé, on va renvoyer l'adresse du bloc obtenu mais, si $b.t > t$ on insère après b un nouveau bloc libre de taille $b.t - t$. Dans tous les cas, on fait pointer la liste des blocs libres vers le bloc qui suit b, qui est peut-être le bloc nouvellement créé et a de grandes chances d'être libre.

Ce mécanisme est implémenté dans la fonction `allocation` :

```
void *allocation(uint32_t taille)
{
    struct bloc *premier_bloc = blocs_libres;
    struct bloc *bloc_courant = premier_bloc;

    while (!bloc_courant->libre || bloc_courant->taille < taille)
    {
        bloc_courant = bloc_courant->suivant;
        if (bloc_courant == premier_bloc)
        {
            // Retour au point de départ : échec d'allocation
            return NULL;
        }
    }

    // bloc_courant pointe sur un bloc libre de bonne taille

    void *adresse = bloc_courant->adresse;
    bloc_courant->libre = false;
    if (bloc_courant->taille > taille) {
        // on le sépare en deux pour récupérer la place
        struct bloc *bloc_libre = cree_bloc((uint8_t *)adresse + taille,
                bloc_courant->taille - taille, true);
        bloc_courant->taille = taille;
        bloc_libre->suivant = bloc_courant->suivant;
        bloc_courant->suivant = bloc_libre;
    }

    // On pointe sur le bloc suivant qui est sûrement libre
    blocs_libres = bloc_courant->suivant;

    return adresse;
}
```

Pour libérer un bloc, on parcourt la liste jusqu'à trouver le bloc correspondant à l'adresse à libérer et on indique que le bloc est libre. Ici, il y a deux `assert` permettant de s'assurer que l'adresse est bien celle d'un bloc et que le bloc n'a pas déjà été libéré.

```
void liberation(void *adresse)
{
    struct bloc *premier_bloc = blocs_libres;
    struct bloc *bloc_courant = premier_bloc;

    while (bloc_courant->adresse != adresse)
    {
        bloc_courant = bloc_courant->suivant;
        // adresse invalide
        assert(bloc_courant != premier_bloc);
    }

    // pas de double libération
    assert(!bloc_courant->libre);

    // on libère l'adresse
    bloc_courant->libre = true;
}
```

Voici un premier programme de test de ces fonctions qui alloue 10 octets puis effectue plusieurs allocations. L'allocation de `c` échoue car il n'y a plus de place libre.

```
int main(void)
{
    creation_blocs_libres(10);

    uint8_t *a = allocation(5);
    uint8_t *b = allocation(3);
    uint8_t *c = allocation(3);
    liberation(b);
    uint8_t *d = allocation(2);

    printf("Allocation a:%p b:%p c:%p d:%p\n",
           (void *)a, (void *)b, (void *)c, (void *)d);

    destruction_blocs_libres();

    return 0;
}
```

Ce programme affiche alors

```
Allocation a:0x55c6c0c922a0 b:0x55c6c0c922a5 c:(nil) d:0x55c6c0c922a8
```



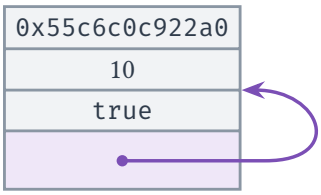
On remarque que `d` n'est pas placé à l'adresse libérée de `b` : la recherche commence au bloc pointé par `blocs_libres`, c'est-à-dire au bloc libre de 2 octets créé lors de l'allocation de `b`.

Voici l'évolution de la liste circulaire en présentant les maillons sous la forme :

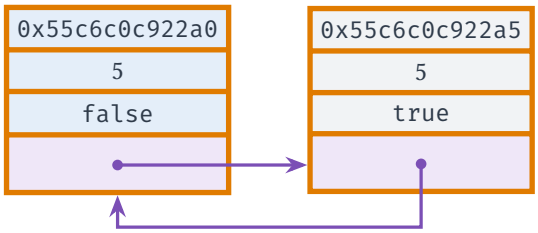


L'évolution de la liste des blocs entre les lignes 3 et 9 est alors :

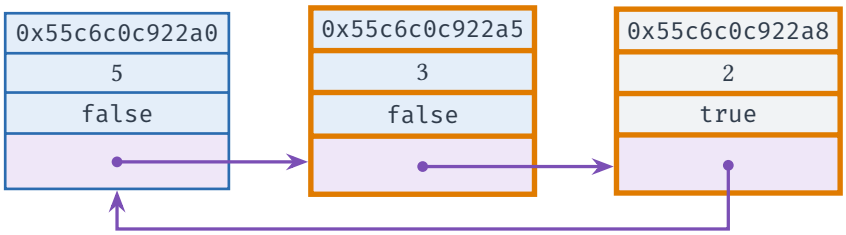
- Ligne 3



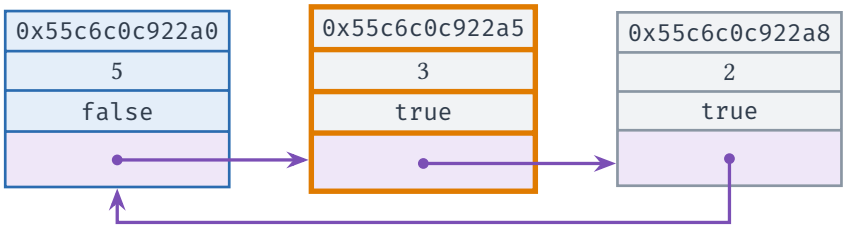
- Ligne 5



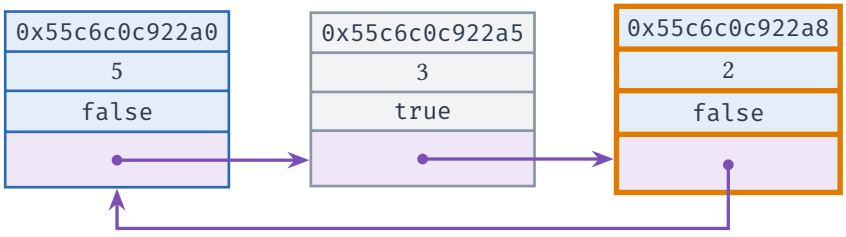
- Ligne 6



- Ligne 8



- Ligne 9



Cette méthode d'allocation a un défaut majeur : elle fragmente l'espace libre. Dans le programme suivant, il sera impossible d'allouer `b` car la liste circulaire contient deux blocs libres de 5 octets et non un bloc libre de 10 octets.

```
creation_blocs_libres(10);

uint8_t *a = allocation(5);
liberation(a);
uint8_t *b = allocation(10);
```

> **Remarque 44.16** On peut observer ce phénomène sur le diagramme précédent à la ligne 8 où deux blocs contigus sont libres et pourraient être fusionnés en un unique bloc de 5 octets.

On peut éviter cela en effectuant une phase de coalescence des blocs libres à la libération. Comme la liste circulaire est simplement chaînée, on ne peut fusionner un bloc libre qu'avec les blocs suivants. En utilisant une liste circulaire doublement chaînée, on pourrait également fusionner avec les blocs précédents.

Deux précautions sont nécessaires :

- les maillons de la liste sont rangés par adresses croissantes à partir du bloc d'adresse `plage_memoire` (les découpages insèrent toujours le nouveau bloc juste après le bloc découpé) ; le maillon qui suit le dernier bloc est donc le premier bloc de la plage, qui ne lui est pas contigu en mémoire : il ne faut pas les fusionner ;
- le maillon pointé par `blocs_libres` peut faire partie des maillons fusionnés et donc libérés : on fait alors pointer `blocs_libres` sur le bloc libéré.

Pour cela on change la fonction `liberation` ainsi :

```
void liberation(void *adresse)
{
    struct bloc *premier_bloc = blocs_libres;
    struct bloc *bloc_courant = premier_bloc;

    while (bloc_courant->adresse != adresse)
    {
        bloc_courant = bloc_courant->suivant;
        // adresse invalide
        assert(bloc_courant != premier_bloc);
    }

    // pas de double libération
    assert(!bloc_courant->libre);

    // on libère l'adresse
    bloc_courant->libre = true;

    premier_bloc = bloc_courant;
    bloc_courant = bloc_courant->suivant;

    // on fusionne avec les blocs libres suivants qui sont contigus
    while (bloc_courant != premier_bloc && bloc_courant->libre
            && bloc_courant->adresse != plage_memoire)
    {
        struct bloc* actuel = bloc_courant;
        premier_bloc->taille += bloc_courant->taille;
        bloc_courant = bloc_courant->suivant;
        free(actuel); // on libere le bloc inutile
    }

    premier_bloc->suivant = bloc_courant;
    // blocs_libres pointait peut-être sur un maillon libéré
    blocs_libres = premier_bloc;
}
```

PARTIE VI

Logique

45 Calcul propositionnel 681
46 TP Calcul propositionnel en C 695
47 TP SAT-solver 701

CHAPITRE 45

Calcul propositionnel

DANS CE CHAPITRE

45.1 Syntaxe
45.2 Sémantique du calcul propositionnel
45.3 Formes normales
45.4 Problème SAT

45.1 Syntaxe

45.1.1 Variables propositionnelles

On considère un ensemble infini dénombrable $\mathscr{V}$ dont les éléments sont appelés des *variables propositionnelles*. On note usuellement ces éléments par des lettres majuscules $A, B, X, Y\ldots$.

Par rapport à ce qui a été fait en mathématiques, ces variables représentent une sorte d'abstraction des formules atomiques des mathématiques. On a pu voir des formules comme « $x = y$ » ou « n est pair », ici on les représentera par des variables.

On va alors s'intéresser à la logique en elle-même indépendamment des mathématiques.

45.1.2 Formules en tant qu'arbre

Grâce aux arbres, on peut donner une définition très précise des formules du calcul propositionnel :

Définition 45.1

Une formule du calcul propositionnel est un arbre ayant des nœuds binaires, étiquetés par $\wedge, \vee, \to$ ou $\leftrightarrow$, des nœuds unaires étiquetés par $\neg$ et dont les feuilles sont étiquetées par $\mathscr{V}$.

Remarque 45.2 Les symboles $\wedge, \vee, \to$ et $\leftrightarrow$ et $\neg$ sont appelés des connecteurs logiques. Les quatre premiers sont dits binaires, ou d'arité 2, le dernier est dit unaire, ou d'arité 1.

Pour le moment, il ne s'agit que de symboles mais ils ne sont pas choisis au hasard et on leur attribuera un sens dans la suite de manière cohérente avec l'usage mathématique.

- $\wedge$ est appelé la conjonction et se lit *et*
- $\vee$ est appelé la disjonction et se lit *ou*
- $\to$ est appelé l'implication et se lit *implique*
- $\leftrightarrow$ est appelé l'équivalence et se lit *équivaut*
- $\neg$ est appelé la négation et se lit *non*

Exemple 45.3

Les formules X et $A \wedge (B \to \neg A)$ sont représentées par les arbres :

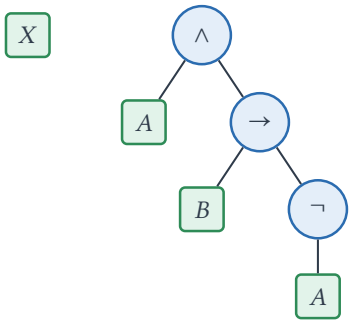


Figure 45.1 — Arbres des formules X et $A \wedge (B \to \neg A)$: connecteurs en nœuds, variables en feuilles.

```
où $A,B,X \in \mathcal{V}$.
```

De cette définition, on déduit directement la notion de taille et de hauteur d'une formule, il s'agit de la taille ou de la hauteur en tant qu'arbre.

On note $\mathrm{Var}(f)$ l'ensemble des variables apparaissant dans f, c'est-à-dire l'ensemble des étiquettes des feuilles de l'arbre f.

45.1.3 Définition inductive des formules

Il est plus traditionnel, bien qu'équivalent, de définir les formules sous la forme d'un ensemble inductif :

Définition 45.4

Les formules du calcul propositionnel sont les éléments du plus petit ensemble $\mathscr{F}$ tel que :

- $\mathscr{V} \subset \mathscr{F}$
- $\forall f, f' \in \mathscr{F}, \forall \circ \in \{\wedge, \vee, \to, \leftrightarrow\}, (f \circ f') \in \mathscr{F}$.
- $\forall f \in \mathscr{F}, \neg f \in \mathscr{F}$.

On peut résumer cela sous la forme de l'équation inductive suivante :

$$\mathscr{F} = \mathscr{V} \mid (\mathscr{F} \wedge \mathscr{F}) \mid (\mathscr{F} \vee \mathscr{F}) \mid (\mathscr{F} \to \mathscr{F}) \mid (\mathscr{F} \leftrightarrow \mathscr{F}) \mid \neg\mathscr{F}$$

Quand on définit ainsi l'ensemble, on peut se demander la nature réelle des objets que l'on a défini. Ici, il y a deux approches classiques :

- Les formules sont des mots sur l'alphabet $\mathscr{V} \cup \{(,), \neg, \vee, \wedge, \to, \leftrightarrow\}$. Par exemple $((X \vee Y) \wedge \neg Z) \in \mathscr{F}$ mais $(X \neg Y) \notin \mathscr{F}$. C'est la vision *historique* des formules comme des chaînes de caractères.
- Les formules sont des arbres comme on l'a vu précédemment. L'avantage principal étant que tous ces arbres sont des formules sans ambigüité de lecture possible.

Que ce soit sous forme de chaîne ou d'arbre, on adopte une représentation plate des formules. Comme cela peut induire beaucoup de parenthèses, on va adopter une convention de priorité : en cas d'ambigüité, c'est la construction la plus prioritaire qui s'effectue.

On a les règles de priorité usuelles : $\neg > \wedge, \vee > \to, \leftrightarrow$. On justifiera de plus dans la suite qu'on peut considérer $\wedge$ et $\vee$ comme étant associatives. Enfin, on convient que $\to$ est associative à droite.

Exemple 45.5

- $(A \wedge (B \vee C))$ s'écrit $A \wedge (B \vee C)$
- $(A \vee (B \vee C))$ s'écrit $A \vee B \vee C$
- $(A \to (B \to C))$ s'écrit $A \to B \to C$
- $((A \to B) \to C)$ s'écrit $(A \to B) \to C$

45.2 Sémantique du calcul propositionnel

On considère ici un ensemble à deux éléments $\mathbb{B} = \{V, F\}$ dont les éléments sont appelés des valeurs de vérité. On parle de la valeur *vraie* pour V et *fausse* pour F. C'est une pure convention, il n'y a aucune véracité mathématique pour appuyer ces termes. Quelque part, c'est la même situation que pour les bases directes.

45.2.1 Valuation et valeur de vérité d'une formule

Définition 45.6

Une valuation est une fonction $v : \mathscr{V} \to \mathbb{B}$.

C'est une forme d'environnement pour les formules comme pourrait l'être une affectation de valeurs à des variables pour évaluer des expressions arithmétiques. On va donc pouvoir s'appuyer sur une telle valuation pour définir la valeur de vérité d'une formule comme une évaluation :

Définition 45.7

Soit v une valuation et $f \in \mathscr{F}$. On appelle **valeur de vérité** de f sous la valuation v, l'élément $[f]_v \in \mathbb{B}$ défini inductivement par :

- $\forall X \in \mathscr{V}, [X]_v = v(X)$
- $\forall f \in \mathscr{F}, [\neg f]_v = \begin{cases} V \text{ si } [f]_v = F \\ F \text{ sinon} \end{cases}$
- $\forall f, f' \in \mathscr{F}, \forall \circ \in \{\wedge, \vee, \to, \leftrightarrow\}$ on détermine $[f \circ f']_v$ en fonction de $[f]_v$ et $[f']_v$ selon la table suivante :

$[f]_v$	$[f']_v$	$[f \vee f']_v$	$[f \wedge f']_v$	$[f \to f']_v$	$[f \leftrightarrow f']_v$
V	V	V	V	V	V
V	F	V	F	F	F
F	V	V	F	V	F
F	F	F	F	V	V

Exemple 45.8

Si $f = A \vee B \to A \wedge C$ et qu'on sait que $v(A) = v(C) = V$ et $v(B) = F$, alors $[A \vee B]_v = V$ et $[A \wedge C]_v = V$ selon la table. On a alors $[f]_v = V$.

45.2.2 Table de vérité

Une formule ne pouvant faire apparaître qu'un nombre fini de variables, les valuations comportent beaucoup d'informations n'ayant aucune influence sur la valeur de vérité de f. Cela est précisé dans le théorème suivant :

Théorème 45.9

Soit $f \in \mathscr{F}$ et $v, v' : \mathscr{V} \to \mathbb{B}$ deux valuations.

Si $\forall X \in \text{Var}(f), v(X) = v'(X)$ alors $[f]_v = [f]_{v'}$.

Preuve Par induction structurelle sur f, on montre la propriété $P(f)$: « pour toutes valuations v, v' coïncidant sur $\text{Var}(f)$, $[f]_v = [f]_{v'}$ ».

- Si $X \in \mathscr{V}$ et $f = X$, alors $X \in \text{Var}(f)$ et $[f]_v = v(X) = v'(X) = [f]_{v'}$.
- Si $P(f)$ est vraie et si v, v' coïncident sur $\text{Var}(\neg f) = \text{Var}(f)$, alors $[f]_v = [f]_{v'}$ et donc $[\neg f]_v = V \Leftrightarrow [f]_v = F \Leftrightarrow [f]_{v'} = F \Leftrightarrow [\neg f]_{v'} = V$. Ainsi, comme il n'y a que deux valeurs possibles, $[\neg f]_v = [\neg f]_{v'}$.
- Si $P(f)$ et $P(f')$ sont vraies et si v, v' coïncident sur $\text{Var}(f \circ f') = \text{Var}(f) \cup \text{Var}(f')$, elles coïncident sur $\text{Var}(f)$ et sur $\text{Var}(f')$, donc $[f]_v = [f]_{v'}$ et $[f']_v = [f']_{v'}$. Or, $[f \circ f']_v$ ne dépend que de $[f]_v$ et $[f']_v$ selon les tables de la définition pour $\circ \in \{\vee, \wedge, \to, \leftrightarrow\}$. On en conclut directement que $[f \circ f']_v = [f \circ f']_{v'}$.

□

Ainsi, on peut se concentrer sur les valeurs de vérité associées aux variables présentes dans une formule. Ce faisant, on introduit le concept de tables de vérité :

Définition 45.10

Soit $f \in \mathscr{F}$ dont les n variables (deux à deux distinctes) sont $X_1, ..., X_n$.

On appelle **table de vérité** de f une fonction

$$T_f : \mathbb{B}^n \to \mathbb{B}$$

telle que pour toute valuation v on ait

$$[f]_v = T_f(v(X_1), ..., v(X_n))$$

Théorème 45.11

Toute formule a une table de vérité et elle est unique à l'ordre près de l'énumération de ses variables.

Preuve Soit $f \in \mathscr{F}$ et $X_1, ..., X_n$ ses variables.

1. Si $b = (b_1, ..., b_n) \in \mathbb{B}^n$, on note v_b la valuation définie par
$$v_b : Y \in \mathscr{V} \mapsto \begin{cases} b_i & \text{si } Y = X_i \text{ pour } i \in [\![1, n]\!] \\ F & \text{sinon} \end{cases}$$
On pose alors $T_f : b \mapsto [f]_{v_b}$.
2. Soit v' une valuation et $b = (v'(X_1), ..., v'(X_n))$. On a v_b et v' identiques sur les variables de f, donc d'après le théorème précédent $[f]_{v'} = [f]_{v_b} = T_f(b)$. Ainsi T_f est une table de vérité de f.
3. Si T' est une autre table de vérité pour le même ordre d'énumération, on a pour $b \in \mathbb{B}^n$, par définition de la table appliquée à la valuation v_b, $T'(b) = [f]_{v_b} = T_f(b)$. Donc $T' = T_f$.

□

Remarque 45.12 On représente usuellement une table de vérité sous la forme d'une table avec une colonne par variable et une colonne pour la valeur de f.

Par exemple, pour $f = (A \wedge B) \vee C$:

A	B	C	f
V	V	V	V
V	V	F	V
V	F	V	V
V	F	F	F
F	V	V	V
F	V	F	F
F	F	V	V
F	F	F	F

45.2.3 Satisfiabilité, tautologie

Définition 45.13

Soit $f \in \mathscr{F}$ et $v : \mathscr{V} \to \mathbb{B}$.

On note $v \vDash f$ quand $[f]_v = V$ et on dit que v est un modèle de f. On note ainsi $\text{Mod}(f) = \{v \mid v \vDash f\}$.

- Si $\text{Mod}(f) \neq \emptyset$ on dit que f est **satisfiable**.
- Si $\text{Mod}(f) = \mathbb{B}^{\mathscr{V}}$ on dit que f est **une tautologie**.
- Si $\text{Mod}(f) = \emptyset$ on dit que f est **une antilogie**.

Théorème 45.14

- f est une tautologie ssi T_f est constante de valeur V
- f est une antilogie ssi T_f est constante de valeur F
- f est satisfiable ssi $V \in T_f(\mathbb{B}^n)$.

✍ **Preuve** Pour toute valuation v, $[f]_v = T_f(v(X_1), ..., v(X_n))$ et, réciproquement, tout $b \in \mathbb{B}^n$ s'écrit $(v_b(X_1), ..., v_b(X_n))$ avec la valuation v_b de la preuve précédente. Ainsi $\left\{[f]_v \mid v \in \mathbb{B}^{\mathscr{V}}\right\} = T_f(\mathbb{B}^n)$, d'où les trois équivalences.

□

Exemple 45.15

- $X \vee \neg X$ est une tautologie appelée le tiers-exclu.
- $X \wedge \neg X$ est une antilogie : c'est la contradiction, dont la négation $\neg(X \wedge \neg X)$ est le principe de non-contradiction.
- $((A \to B) \to A) \to A$ est une tautologie.

Remarque 45.16 Il est pratique d'introduire des formules constantes $\top$ et $\bot$ pour représenter une tautologie et une antilogie quelconque.

45.2.4 Équivalence entre deux formules

Définition 45.17

Soient $f, f' \in \mathscr{F}$. On dit que f et f' sont équivalentes, et l'on note $f \Leftrightarrow f'$ ou $f \equiv f'$, lorsque f et f' ont mêmes modèles, c'est-à-dire lorsque $\text{Mod}(f) = \text{Mod}(f')$.

Théorème 45.18

L'équivalence logique est une relation d'équivalence.

Remarque 45.19

- Cela signifie que deux formules équivalentes ne peuvent être distinguées par leur valeur de vérité sous une valuation.
- f est une tautologie si et seulement si $f \Leftrightarrow \top$
- f est une antilogie si et seulement si $f \Leftrightarrow \bot$

Théorème 45.20

Soient $f, f' \in \mathscr{F}$ avec $\text{Var}(f) = \text{Var}(f')$, variables énumérées dans le même ordre pour les deux tables.

On a $f \Leftrightarrow f'$ si et seulement si $T_f = T_{f'}$.

Remarque 45.21 Cela permet de vérifier l'équivalence entre deux formules en établissant les deux tables de vérité et en vérifiant qu'elles sont identiques. Comme une telle table comporte 2^n lignes pour n variables, c'est souvent fastidieux.

Il y a de nombreuses équivalences logiques qui sont fondamentales car elles permettent de simplifier les formules pour travailler sur des formules plus simples. On va démontrer ici les principales :

Théorème 45.22

Soient $P, Q, R \in \mathscr{V}$

- $\neg\neg P \Leftrightarrow P$
- $P \to Q \Leftrightarrow \neg P \vee Q$
- $P \leftrightarrow Q \Leftrightarrow (P \wedge Q) \vee (\neg P \wedge \neg Q)$

Lois de de Morgan :

- $\neg(P \vee Q) \Leftrightarrow \neg P \wedge \neg Q$
- $\neg(P \wedge Q) \Leftrightarrow \neg P \vee \neg Q$

Commutativité :

- $P \wedge Q \Leftrightarrow Q \wedge P$
- $P \vee Q \Leftrightarrow Q \vee P$

Associativité :

- $P \wedge (Q \wedge R) \Leftrightarrow (P \wedge Q) \wedge R$
- $P \vee (Q \vee R) \Leftrightarrow (P \vee Q) \vee R$

Distributivité :

- $P \wedge (Q \vee R) \Leftrightarrow (P \wedge Q) \vee (P \wedge R)$
- $P \vee (Q \wedge R) \Leftrightarrow (P \vee Q) \wedge (P \vee R)$

Preuve Il suffit de faire de fastidieuses vérifications à l'aide de tables de vérité (le théorème précédent s'applique car les deux membres ont les mêmes variables). Par exemple, pour la première loi de distributivité, les deux membres valent V exactement lorsque P vaut V et l'un de Q, R vaut V. On peut se contenter de ne démontrer qu'une partie d'entre elles grâce à la double négation et aux lois de de Morgan.

□

45.2.5 Substitution et congruence

On vient de démontrer des équivalences mais le fait d'utiliser des variables semble en restreindre la portée. En effet, si on considère la formule $\neg\neg(A \to B)$ il ne semble pas possible d'en déduire directement qu'elle est équivalente à $A \to B$ car l'équivalence $\neg\neg X \Leftrightarrow X$ ne s'applique que pour une variable.

Or, dès le départ, les variables propositionnelles ne se voulaient qu'être des abstractions sur des propriétés. On va démontrer ici un lemme de substitution qui permettra de déduire de nouvelles équivalences en remplaçant une variable par une formule.

Définition 45.23

Soit $f, g \in \mathscr{F}$ et $X \in \mathscr{V}$. On définit la substitution de X par g, notée $f[g/X]$ par induction ainsi :

- Si $Y \in \mathscr{V} \setminus \{X\}$, $Y[g/X] = Y$
- $X[g/X] = g$
- Si $f \in \mathscr{F}$, $(\neg f)[g/X] = \neg f[g/X]$
- Si $\circ \in \{\wedge, \vee, \to, \leftrightarrow\}$ et $f, f' \in \mathscr{F}$, $(f \circ f')[g/X] = f[g/X] \circ f'[g/X]$

Lemme 45.24

Soit $f, g \in \mathscr{F}, X \in \mathscr{V}$ et $v : \mathscr{V} \to \mathbb{B}$.

On pose $v' : \mathscr{V} \to \mathbb{B}$ telle que $v'(X) = [g]_v$ et $v'(Y) = v(Y)$ pour $Y \neq X$.

On a alors $[f[g/X]]_v = [f]_{v'}$.

Preuve Par induction structurelle sur f (à g, X et v fixés).

- Si $f = X$, $[X[g/X]]_v = [g]_v = v'(X) = [X]_{v'}$.
- Si $f = Y \in \mathscr{V} \setminus \{X\}$, $[Y[g/X]]_v = [Y]_v = v(Y) = v'(Y) = [Y]_{v'}$.
- Si $f = \neg f_1$ avec la propriété vraie pour f_1, alors $[(\neg f_1)[g/X]]_v = [\neg(f_1[g/X])]_v$ est la négation de $[f_1[g/X]]_v = [f_1]_{v'}$, c'est donc $[\neg f_1]_{v'}$.
- Si $f = f_1 \diamond f_2$ avec la propriété vraie pour f_1 et f_2, alors $[(f_1 \diamond f_2)[g/X]]_v = [f_1[g/X] \diamond f_2[g/X]]_v$ se lit dans la table de $\diamond$ à partir de $[f_1[g/X]]_v = [f_1]_{v'}$ et $[f_2[g/X]]_v = [f_2]_{v'}$: c'est donc $[f_1 \diamond f_2]_{v'}$.

□

Théorème 45.25

Soient $f, g, h \in \mathscr{F}$ et $X \in \mathscr{V}$.

Si $f \equiv g$, alors $f[h/X] \equiv g[h/X]$.

Preuve Soit v une valuation, on définit v' comme dans le lemme précédent avec $v'(X) = [h]_v$. On a alors $[f[h/X]]_v = [f]_{v'}$ et $[g[h/X]]_v = [g]_{v'}$. Or, par hypothèse, $[f]_{v'} = [g]_{v'}$.

On a ainsi l'égalité $[f[h/X]]_v = [g[h/X]]_v$ et donc $f[h/X] \equiv g[h/X]$.

□

Le fait de faire passer des équivalences sous des opérations, ici les connecteurs logiques, est traditionnellement appelé une congruence. On a ici le théorème suivant :

Théorème 45.26

Soient $f, g, h \in \mathscr{F}$ où $f \equiv g$.

On a :

- $\neg f \equiv \neg g$
- $\forall \diamond \in \{\wedge, \vee, \to, \leftrightarrow\}$, $f \diamond h \equiv g \diamond h$ et $h \diamond f \equiv h \diamond g$.

Preuve Soit v une valuation. Par hypothèse $[f]_v = [g]_v$. Or $[\neg f]_v$ ne dépend que de $[f]_v$, et $[f \diamond h]_v$ (resp. $[h \diamond f]_v$) ne dépend que de $[f]_v$ et $[h]_v$: on obtient donc les mêmes valeurs en remplaçant f par g.

□

45.2.6 Conséquence logique

Définition 45.27

Soit $f, g \in \mathscr{F}$, on dit que f est **conséquence logique** de g lorsque $\mathrm{Mod}(g) \subset \mathrm{Mod}(f)$.

Autrement dit, lorsque pour toute valuation v, $[g]_v = V$ implique $[f]_v = V$.

On note alors $g \vDash f$.

Si $\Gamma \subset \mathscr{F}$ on note $\Gamma \vDash f$ s'il existe $g_1, ..., g_n \in \Gamma$ tels que $g_1 \wedge ... \wedge g_n \vDash f$ (avec la convention qu'une conjonction vide, pour $n = 0$, vaut $\top$). On note aussi $g_1, ..., g_p \vDash f$ pour $\{g_1, ..., g_p\} \vDash f$.

Lemme 45.28

Soient $\Gamma, \Delta \subset \mathscr{F}$ et $f \in \mathscr{F}$.

Si $\Gamma \vDash f$ alors $\Gamma \cup \Delta \vDash f$.

Preuve Par définition, il existe $g_1, ..., g_n \in \Gamma$ tels que $g_1 \wedge ... \wedge g_n \vDash f$. Comme ces formules sont aussi dans $\Gamma \cup \Delta$, on a directement $\Gamma \cup \Delta \vDash f$.

On aura aussi besoin du fait suivant : si $g \vDash f$, alors pour toute formule h, $g \wedge h \vDash f$. En effet, soit v une valuation telle que $v \in \mathrm{Mod}(g \wedge h)$. On a donc $[g \wedge h]_v = V$ et par définition de la valeur de vérité d'une

conjonction, $[g]_v = [h]_v = V$. Or, comme $[g]_v = V$ et que $g \vDash f$ on a $[f]_v = V$. On vient de montrer $g \wedge h \vDash f$. Avec la commutativité et l'associativité de $\wedge$, on en déduit que si une conjonction de formules de Γ a f pour conséquence, il en est de même de toute conjonction contenant ces formules.

□

Théorème 45.29

Soient $f, g \in \mathscr{F}$. Les propriétés suivantes sont équivalentes :

1. $g \vDash f$
2. $g, \neg f \vDash \bot$
3. $g \to f$ est une tautologie

Preuve 1. $\Rightarrow$ 2. : Supposons que $g \vDash f$ et supposons qu'il existe une valuation v telle que $[g \wedge \neg f]_v = V$. On a donc $[g]_v = [\neg f]_v = V$ par l'argument usuel sur la table de la conjonction. Ainsi $[f]_v = F$ mais comme $g \vDash f$, on a $[f]_v = V$. Or, une formule a une unique valeur de vérité, donc on aboutit à une contradiction. Ainsi Mod $(g \wedge \neg f) = \emptyset$ et $g \wedge \neg f \vDash \bot$ vu que Mod $(\bot) = \emptyset$.

2. $\Rightarrow$ 3. Supposons que $g, \neg f \vDash \bot$. Une conjonction de formules parmi $g, \neg f$ a pour conséquence $\bot$, donc d'après le fait établi dans la preuve du lemme précédent, $g \wedge \neg f \vDash \bot$, c'est-à-dire Mod $(g \wedge \neg f) \subset$ Mod $(\bot) = \emptyset$. Soit v une valuation et supposons, par l'absurde, que $[g \to f]_v = F$. Alors, de par la table de $\to$, cela signifie nécessairement que $[g]_v = V$ et $[f]_v = F$. Donc $[g \wedge \neg f]_v = V$. Contradiction car on a trouvé un modèle de $g \wedge \neg f$.

3. $\Rightarrow$ 1. Supposons que $g \to f$ soit une tautologie et soit v une valuation telle que $[g]_v = V$. Comme $[g \to f]_v = V$ on a nécessairement $[f]_v = V$ par étude de la table de $\to$. Donc, on a bien $g \vDash f$.

□

Remarque 45.30 Le fait de se ramener à montrer $\Gamma, \neg f \vDash \bot$ pour prouver que $\Gamma \vDash f$ est un principe très important appelé la *réfutation*. Il est à la base de la méthode de *résolution*, introduite par le logicien John Alan Robinson, qui sert de base au langage de programmation logique Prolog.

Théorème 45.31

Si $f \vDash g$ et $g \vDash h$ alors $f \vDash h$.

Preuve Il s'agit juste de la transitivité de l'inclusion sur les modèles.

□

Remarque 45.32 Cette propriété, appelée *syllogisme hypothétique*, est, comme le *Modus Ponens* (si f et $f \to g$ sont vraies alors g l'est), un principe fondamental du raisonnement identifié très tôt dans l'histoire des mathématiques.

45.3 Formes normales

On a vu beaucoup d'équivalences logiques dans la partie précédente, elles permettent de réécrire une formule sous une forme plus pratique pour raisonner. On parle alors de **forme normale**.

Comme on a pu le voir, les opérateurs $\to$ et $\leftrightarrow$ peuvent s'exprimer avec les trois autres. On va donc considérer ici uniquement des formules avec les opérateurs $\vee, \wedge, \neg$.

45.3.1 Forme normale conjonctive et disjonctive

Définition 45.33

Une formule de la forme X ou $\neg X$ avec $X \in \mathscr{V}$ est appelée un **littéral**, on précise parfois positif ou négatif.

Une formule de la forme $l_1 \vee \ldots \vee l_n$ où les l_i sont des littéraux est appelée une **clause**. On parle de clause vide pour $\bot$.

Une formule de la forme $C_1 \wedge \ldots \wedge C_m$ où les C_j sont des clauses est appelée une **forme normale conjonctive** (FNC). On parle de FNC vide pour $\top$.

Exemple 45.34

- $A \wedge (\neg A \vee B) \wedge \bot$ est une FNC.

En vertu de l'associativité des connecteurs, on note de manière synthétique

$$\bigwedge_{i=1}^{n} \bigvee_{j=1}^{n_i} l_{i,j}$$

une FNC, où les $l_{i,j}$ sont des littéraux.

Définition 45.35

On définit de même les formes normales disjonctives (FND) comme les disjonctions de conjonctions de littéraux.

Ce sont les formules sous la forme :

$$\bigvee_{i=1}^{n} \bigwedge_{j=1}^{n_i} l_{i,j}$$

où les $l_{i,j}$ sont des littéraux.

45.3.2 Mise sous forme normale disjonctive

Soit $f \in \mathscr{F}$ dont les variables sont $X_1, \ldots, X_n$, on peut passer de sa table de vérité à une FND en posant :

$$\Phi(f) = \bigvee_{(b_1,\ldots,b_n) \in T_f^{-1}(\{V\})} \bigwedge_{i=1}^{n} l(X_i, b_i)$$

où $l(X_i, V) = X_i$ et $l(X_i, F) = \neg X_i$ (si $T_f^{-1}(\{V\})$ est vide, $\Phi(f)$ est la disjonction vide, c'est-à-dire $\bot$).

On a $\Phi(f) \equiv f$ car $\Phi(f)$ a été construite pour avoir la même table de vérité que f : pour une valuation ν, la conjonction $\bigwedge_{i=1}^{n} l(X_i, b_i)$ est vraie si et seulement si $\nu(X_i) = b_i$ pour tout i, donc $[\Phi(f)]_\nu = V$ si et seulement si $(\nu(X_1), \ldots, \nu(X_n)) \in T_f^{-1}(\{V\})$, c'est-à-dire si et seulement si $[f]_\nu = V$.

On remarque tout de suite que la taille de $\Phi(f)$ dépend du nombre de valeurs à V dans sa table de vérité. Pour une tautologie portant sur n variables, $\Phi(f)$ est ainsi une disjonction de 2^n conjonctions de n littéraux, de taille $\Theta(n2^n)$.

45.3.3 Mise sous forme normale conjonctive

Définition 45.36

Soit $f \in \mathscr{F}$, on définit la formule $\varphi(f)$ ainsi par induction structurelle :

- Si $X \in \mathscr{V}$, $\varphi(X) = X$
- Si $f \in \mathscr{F}$,
  - Soit $f = X \in \mathscr{V}$ et $\varphi(\neg X) = \neg X$
  - Soit $f = \neg f'$ et $\varphi(\neg f) = \varphi(f')$
  - Soit $f = g \wedge h$ et $\varphi(\neg f) = \varphi(\neg g) \vee \varphi(\neg h)$.
  - Soit $f = g \vee h$ et $\varphi(\neg f) = \varphi(\neg g) \wedge \varphi(\neg h)$.
- Si $f, f' \in \mathscr{F}$, $\varphi(f \vee f') = \varphi(f) \vee \varphi(f')$ et $\varphi(f \wedge f') = \varphi(f) \wedge \varphi(f')$.

Théorème 45.37

On a $\varphi(f) \Leftrightarrow f$ et dans $\varphi(f)$ les négations appartiennent toutes à des littéraux.

Preuve On remarque d'abord que φ est bien définie : chaque appel récursif porte sur une formule de taille strictement inférieure ($\neg f'$ est plus petite que $\neg\neg f'$, $\neg g$ et $\neg h$ sont plus petites que $\neg(g \wedge h)$, etc.).

On montre alors par récurrence forte sur la taille de f que $\varphi(f) \equiv f$ et que les négations de $\varphi(f)$ portent toutes sur des variables. Les cas $f = X$, $f = \neg X$, $f = g \vee h$ et $f = g \wedge h$ découlent directement de l'hypothèse de récurrence (et de la congruence pour l'équivalence). Pour $f = \neg\neg f'$, on utilise $\neg\neg f' \equiv f'$; pour $f = \neg(g \wedge h)$ (resp. $\neg(g \vee h)$), on utilise la loi de de Morgan $\neg(g \wedge h) \equiv \neg g \vee \neg h$ (resp. $\neg(g \vee h) \equiv \neg g \wedge \neg h$), ces équivalences entre formules étant obtenues à partir de celles entre variables par le théorème de substitution, puis l'hypothèse de récurrence appliquée à $\neg g$ et $\neg h$.

□

Il est possible de passer de FND à FNC à l'aide d'une négation et des lois de de Morgan. On en déduit donc une manière d'obtenir une FNC équivalente à f : on met $\neg\Phi(\neg f) \equiv f$ sous FNC en éliminant la négation.

Cependant, on peut obtenir une FNC plus directement par induction comme on peut le voir dans la définition suivante.

Définition 45.38

Soit f une formule dont les négations appartiennent toutes à des littéraux, on définit $\psi(f)$ par induction structurelle :

- $\psi(l) = l$ si l est un littéral
- $\psi(f \wedge g) = \psi(f) \wedge \psi(g)$
- Si $\psi(f) = f_1 \wedge f_2$ alors $\psi(f \vee g) = \psi(f_1 \vee g) \wedge \psi(f_2 \vee g)$.
- Si $\psi(f)$ n'est pas une conjonction et $\psi(g) = g_1 \wedge g_2$ alors $\psi(f \vee g) = \psi(f \vee g_1) \wedge \psi(f \vee g_2)$.
- Si ni $\psi(f)$ ni $\psi(g)$ ne sont des conjonctions alors $\psi(f \vee g) = \psi(f) \vee \psi(g)$.

Théorème 45.39

Soit $f \in \mathcal{F}$, $\psi(\varphi(f)) \equiv f$ et $\psi(\varphi(f))$ est en forme normale conjonctive.

Preuve *Esquisse.* Pour deux FNC a et b, on définit $D(a, b)$ par : $D(a_1 \wedge a_2, b) = D(a_1, b) \wedge D(a_2, b)$, sinon $D(a, b_1 \wedge b_2) = D(a, b_1) \wedge D(a, b_2)$, et sinon (deux clauses) $D(a, b) = a \vee b$. Par récurrence sur la taille de a plus celle de b, $D(a, b)$ est bien définie, est une FNC et $D(a, b) \equiv a \vee b$ par distributivité (et congruence).

On vérifie ensuite que $\psi(c) = c$ pour toute FNC c, puis, par récurrence sur la somme des tailles de $\psi(f)$ et $\psi(g)$, que $\psi(f \vee g) = D(\psi(f), \psi(g))$: par exemple si $\psi(f) = f_1 \wedge f_2$, alors $\psi(f_1) = f_1$ est plus petite que $\psi(f)$ et $\psi(f \vee g) = \psi(f_1 \vee g) \wedge \psi(f_2 \vee g) = D(f_1, \psi(g)) \wedge D(f_2, \psi(g)) = D(\psi(f), \psi(g))$. Ceci assure que ψ est bien définie, et une induction structurelle sur f montre alors que $\psi(f)$ est une FNC équivalente à f. On conclut avec le théorème précédent appliqué à $\varphi(f)$.

□

Remarque 45.40 Il est possible d'obtenir ainsi une formule de taille exponentielle.

L'exemple caractéristique est

$$\bigvee_{i=1}^{n} (X_{i,1} \wedge X_{i,2}) \equiv \bigwedge_{a_1 \in \{1,2\}, \ldots, a_n \in \{1,2\}} \left(X_{1,a_1} \vee \ldots \vee X_{n,a_n}\right)$$

où la FNC de droite comporte 2^n clauses.

On passe ainsi d'une formule de taille $O(n)$ à une formule de taille $O(2^n)$.

Il est possible d'implémenter directement cette méthode :

```
type formula = Var of string
    | And of formula * formula
    | Or of formula * formula
    | Not of formula

let rec elim_not f = match f with
    | Var s -> Var s
    | Not (Var s) -> Not (Var s)
    | Not (Not f) -> elim_not f
    | Not (Or (f1,f2)) -> And( elim_not(Not f1), elim_not(Not f2) )
    | Not (And (f1,f2)) -> Or( elim_not(Not f1), elim_not(Not f2) )
    | Or(f1, f2) -> Or(elim_not f1, elim_not f2)
    | And(f1, f2) -> And(elim_not f1, elim_not f2)

let rec distrib_or f = match f with
    | Var _ -> f
    | Not (Var _) -> f
    | And(f1,f2) -> And(distrib_or f1, distrib_or f2)
    | Or(f1,f2) -> begin
            match distrib_or f1 with
            | And(f11, f12) -> And(distrib_or (Or(f11,f2)),
                                   distrib_or (Or(f12,f2)))
            | f1' -> match distrib_or f2 with
                | And (f21, f22) -> And(distrib_or(Or(f1', f21)),
                                        distrib_or(Or(f1',f22)))
                | f2' -> Or(f1', f2')
            end
    | _ -> failwith "Formule avec négation en dehors des littéraux"

let fnc f = distrib_or (elim_not f)
```

45.4 Problème SAT

45.4.1 Définition

Un des problèmes les plus importants en logique et plus largement en informatique est le problème SAT :

Problème 45.41 — SAT

- **Entrée** : une formule f, le plus souvent en FNC.
- **Sortie** : un booléen indiquant si f est satisfiable.

45.4.2 k-SAT

Le problème suivant permet un plus grand contrôle sur les formules.

Problème 45.42 — k-SAT

- **Entrée** : une formule f en FNC dont les clauses comportent au plus k littéraux.
- **Sortie** : un booléen indiquant si f est satisfiable.

Naturellement, si on sait résoudre k-SAT pour tout k, alors on sait résoudre SAT vu que les formules sont finies.

45.4.3 1-SAT

Les formules à résoudre sont des conjonctions de littéraux vu que les clauses sont réduites à un littéral. Or, soit cette conjonction contient deux littéraux opposés auquel cas $f \equiv \bot$, ou alors elle est satisfiable en considérant $v(X) = V$ si X apparaît positivement et $v(X) = F$ sinon.

45.4.4 2-SAT

Pour résoudre 2-SAT on va construire un graphe en transformant les clauses en implications :

$$X \vee Y \Leftrightarrow \neg X \to Y \Leftrightarrow \neg Y \to X$$

$$\neg X \vee Y \Leftrightarrow X \to Y \Leftrightarrow \neg Y \to \neg X$$

$$\neg X \vee \neg Y \Leftrightarrow X \to \neg Y \Leftrightarrow Y \to \neg X$$

On peut alors construire un graphe où les sommets sont les littéraux apparaissant dans une formule ainsi que leur négation, et où l'on met un arc de l vers l' quand une clause est équivalente à $l \to l'$ (chaque clause donne ainsi deux arcs).

Un chemin dans ce graphe est une chaîne d'implications. On peut montrer que f est satisfiable si et seulement s'il n'existe aucune variable X telle qu'il y ait à la fois un chemin de X à $\neg X$ et un chemin de $\neg X$ à X (un chemin de X à $\neg X$ seul impose simplement $v(X) = F$). Or, pour déterminer cela, il suffit de calculer la fermeture transitive du graphe, par exemple avec des parcours successifs, ou avec Floyd-Warshall. Cela se fait donc en temps et en espace polynomial dans le nombre de variables. On peut même le faire en temps linéaire en calculant les composantes fortement connexes du graphe.

45.4.5 3-SAT

On peut montrer un théorème assurant que ce chapitre aura un nombre fini de paragraphes :

Théorème 45.43

Si on sait résoudre 3-SAT alors on sait résoudre k-SAT pour tout k.

✍ **Preuve** Une clause $l_1 \vee \ldots \vee l_p$ avec $p > 3$ est remplacée par $(l_1 \vee l_2 \vee Z) \wedge (\neg Z \vee l_3 \vee \ldots \vee l_p)$ où Z est une variable nouvelle. Les deux formules ne sont pas équivalentes mais sont *équisatisfiables* : un modèle de la clause initiale se prolonge en un modèle de la nouvelle formule (on pose $v(Z) = F$ si l_1 ou l_2 est vrai, $v(Z) = V$ sinon) et réciproquement un modèle de la nouvelle formule rend vrai l'un des l_i (si $v(Z) = V$, c'est l'un de $l_3, \ldots, l_p$, sinon l'un de l_1, l_2). La seconde clause est plus courte d'un littéral : en itérant, on obtient une formule de 3-SAT équisatisfiable, de taille polynomiale en celle de la formule initiale.

□

Cependant, comme on en reparlera dans le chapitre sur la NP-complétude, le problème 3-SAT est *a priori* très difficile et on a peu d'espoir de pouvoir le résoudre de manière générale autrement qu'en énumérant les 2^n valuations.

45.4.6 Algorithme de Quine

L'algorithme de Quine consiste à construire un arbre de décision associé à une formule f inductivement ainsi :

- si f est sans variable, $f \equiv \top$ ou $\bot$ et on renvoie une feuille avec cette valeur.
- sinon on considère une variable X quelconque dans f et on renvoie l'arbre dont la racine est étiquetée par X, a pour fils gauche l'arbre associé à $f[\top/X]$ et pour fils droit l'arbre associé à $f[\bot/X]$.

Si f est satisfiable, alors cet arbre a une feuille de valeur $\top$. Il suffit alors de lire les embranchements pour retrouver le modèle correspondant.

Si f est une tautologie, toutes ses feuilles sont étiquetées par $\top$.

On donne ici une implémentation rapide de cet algorithme :

```
type formule = Et of formule * formule
   | Ou of formule * formule | Non of formule
   | Vrai | Faux | Var of string

let rec var f =
    match f with
    | Ou(f1, f2)
    | Et(f1, f2) -> var f1 @ var f2
    | Non f' -> var f'
    | Var s -> [s]
    | _ -> []
```

```
let rec subst f s v =
    match f with
    | Ou(f1, f2) -> Ou(subst f1 s v, subst f2 s v)
    | Et(f1, f2) -> Et(subst f1 s v, subst f2 s v)
    | Non f' -> Non (subst f' s v)
    | Var t when s = t -> v
    | _ -> f


let rec simpl f =
    match f with
    | Et(f1, f2) -> begin
            match simpl f1, simpl f2 with
            | Vrai, f -> f
            | f, Vrai -> f
            | Faux, _ -> Faux
            | _, Faux -> Faux
            | f1', f2' -> Et(f1', f2')
            end
    | Ou(f1, f2) -> begin
            match simpl f1, simpl f2 with
            | Vrai, _ -> Vrai
            | _, Vrai -> Vrai
            | Faux, f -> f
            | f, Faux -> f
            | f1', f2' -> Ou(f1', f2')
             end
    | Non f -> begin
        match simpl f with
        | Vrai -> Faux
        | Faux -> Vrai
        | f' -> Non f'
        end
    | _ -> f

type quine_resultat = Valide | Invalide | Branchement of string * quine_resultat * quine_resultat

let rec quine f =
    match simpl f with
    | Vrai -> Valide
    | Faux -> Invalide
    | f' -> let s = List.hd (var f') in
        Branchement(s, quine (subst f' s Vrai), quine (subst f' s Faux))

let imp f1 f2 = Ou(Non f1, f2)

let ex = imp (Et(Et(imp (Var "s") (Ou(Var "b", Var "t")),
    imp (Ou(Var "b", Var "a"))  (Et(Var "r", Var "m"))), Non (Var "r")))
    (imp (Var "s") (Var "t"))

let b = quine ex
```

46

TRAVAUX PRATIQUES 46

Calcul propositionnel en C

DANS CE CHAPITRE

46.1 Formules en FNC
46.2 Valuation
46.3 Énumération

46.1 Formules en FNC

On considère les types suivants permettant de représenter une formule en forme normale conjonctive (FNC) :

```c
struct litteral {
    int ident;
    bool negatif;
};
typedef struct litteral litteral;

struct clause {
    int nlitteraux;
    litteral *litteraux;
};
typedef struct clause clause;

struct formule {
    int nclauses;
    clause *clauses;
};
typedef struct formule formule;
```

Un littéral est une variable, d'identifiant `ident`, éventuellement niée si `negatif` vaut `true`. Une clause est un tableau de `nlitteraux` littéraux (leur disjonction) et une formule est un tableau de `nclauses` clauses (leur conjonction).

On identifie ainsi l'ensemble des variables $\mathscr{V}$ à $\mathbb{N}$, ce qui revient à considérer que les variables se notent $X_0, X_1, \ldots$.

Question 1

Allouez et créez la formule

$$f = (X_0 \vee \neg X_1) \wedge (X_1 \vee \neg X_2 \vee \neg X_0) \wedge \neg X_2$$

Corrigé

On alloue la formule, son tableau de clauses, puis le tableau de littéraux de chaque clause. Deux petites fonctions évitent de répéter les mêmes lignes.

```c
// alloue la clause i de f, avec n littéraux (non initialisés)
void alloue_clause(formule *f, int i, int n)
{
    f->clauses[i].nlitteraux = n;
    f->clauses[i].litteraux = malloc(n * sizeof(litteral));
}

void fixe_litteral(formule *f, int i, int j, int ident, bool negatif)
{
    f->clauses[i].litteraux[j].ident = ident;
    f->clauses[i].litteraux[j].negatif = negatif;
}

formule *exemple(void)
{
    formule *f = malloc(sizeof(formule));
    f->nclauses = 3;
    f->clauses = malloc(3 * sizeof(clause));
    alloue_clause(f, 0, 2);            // X0 ∨ ¬X1
    fixe_litteral(f, 0, 0, 0, false);
    fixe_litteral(f, 0, 1, 1, true);
    alloue_clause(f, 1, 3);            // X1 ∨ ¬X2 ∨ ¬X0
    fixe_litteral(f, 1, 0, 1, false);
    fixe_litteral(f, 1, 1, 2, true);
    fixe_litteral(f, 1, 2, 0, true);
    alloue_clause(f, 2, 1);            // ¬X2
    fixe_litteral(f, 2, 0, 2, true);
    return f;
}
```

Question 2

Écrire une fonction

```c
void libere_formule(formule *f);
```

qui libère toute la mémoire d'une formule `f` (on considère que `f` lui-même est un pointeur vers le tas).

Corrigé

```c
void libere_formule(formule *f)
{
    for(int i = 0; i < f->nclauses; i++) {
        free(f->clauses[i].litteraux);
    }
    free(f->clauses);
    free(f);
}
```

On libère dans l'ordre inverse des allocations : une fois `f->clauses` libéré, on ne peut plus accéder aux tableaux de littéraux.

Question 3

Écrire une fonction

```c
int num_variables(formule *f);
```

qui renvoie le plus petit entier n tel que les variables présentes dans `f` sont de numéro $< n$.

Corrigé

```
int num_variables(formule *f)
{
    int n = -1;
    for(int i = 0; i < f->nclauses; i++) {
        for(int j = 0; j < f->clauses[i].nlitteraux; j++) {
            if (f->clauses[i].litteraux[j].ident > n)
                n = f->clauses[i].litteraux[j].ident;
        }
    }
    return n+1;
}
```

On renvoie 0 pour une formule sans variable. Pour l'exemple, on obtient 3.

Question 4

Écrire une fonction

```
void affiche_formule(formule *f);
```

qui affiche une formule.

✍

Corrigé

On note `~` la négation, `|` la disjonction et `&` la conjonction.

```
void affiche_formule(formule *f)
{
    for(int i = 0; i < f->nclauses; i++) {
        if (i > 0)
            printf(" & ");
        printf("(");
        for(int j = 0; j < f->clauses[i].nlitteraux; j++) {
            if (j > 0)
                printf(" | ");
            litteral l = f->clauses[i].litteraux[j];
            printf("%sX%d", l.negatif ? "~" : "", l.ident);
        }
        printf(")");
    }
    printf("\n");
}
```

Pour l'exemple, on obtient `(X0 | ~X1) & (X1 | ~X2 | ~X0) & (~X2)`.

46.2 Valuation

Si on sait que les variables présentes dans `f` sont d'indice $< n$, on peut représenter une valuation pour `f` comme un tableau `bool *v` de taille `n` tel que `v[i] = true` si et seulement si $v(X_i) = 1$.

Question 5

Écrire des fonctions

```
bool evalue_litteral(litteral *l, bool *valuation);
bool evalue_clause(clause *c, bool *valuation);
bool evalue(formule *f, bool *valuation);
```

telles que l'appel à `evalue(f, v)` renvoie la valeur de $[f]_v$.

Corrigé

```
bool evalue_litteral(litteral *l, bool *valuation)
{
    if (l->negatif)
        return !valuation[l->ident];
    return valuation[l->ident];
}

bool evalue_clause(clause *c, bool *valuation)
{
    bool v = false;
    for(int i = 0; i < c->nlitteraux; i++)
        v = v || evalue_litteral(&c->litteraux[i], valuation);
    return v;
}

bool evalue(formule *f, bool *valuation)
{
    bool v = true;
    for(int i = 0; i < f->nclauses; i++) {
        v = v && evalue_clause(&f->clauses[i], valuation);
    }
    return v;
}
```

Conformément aux conventions sur la FNC, une clause vide (disjonction vide) est fausse et une formule sans clause (conjonction vide) est vraie. Grâce à l'évaluation paresseuse de `||` et `&&`, l'évaluation s'arrête dès qu'un littéral vrai ou une clause fausse est rencontré.

46.3 Énumération

Pour énumérer les valuations sur n variables, on va identifier un tableau `bool *v` de taille n avec le nombre écrit en binaire $N(v) = \sum_{i=0}^{n-1} v[i]2^i$.

Question 6

Écrire une fonction

```
bool suivante(int n, bool *valuation);
```

telle que `suivante(n, v)`, où `v` est un tableau de `n` valeurs, place dans `v` le nombre $N(v)+1$ et renvoie `true` si $N(v) < 2^n - 1$. Si $N(v) = 2^n - 1$, la fonction renvoie `false`.

Corrigé

On ajoute 1 en binaire : les 1 de poids faible deviennent des 0 et le premier 0 devient un 1. S'il n'y a pas de 0, on était à $2^n - 1$.

```
bool suivante(int n, bool *valuation)
{
    int j = 0;
    while(j < n && valuation[j]) {
        j++;
    }
    if (j == n)
        return false;
    for(int i = 0; i < j; i++)
        valuation[i] = false;
    valuation[j] = true;
    return true;
}
```

À partir du tableau `v` rempli de `false` (et donc $N(v) = 0$) et en utilisant la fonction précédente, on peut énumérer toutes les valuations facilement.

Par exemple, la table de vérité de la formule `f` de la question 1 s'obtient avec :

```
bool v[3] = { false, false, false };
do {
    printf("%d-%d-%d | %d\n", v[0], v[1], v[2], evalue(f, v));
} while(suivante(3, v));
```

qui affiche les valeurs de X_0, X_1, X_2 puis celle de f :

```
0-0-0 | 1
1-0-0 | 1
0-1-0 | 0
1-1-0 | 1
0-0-1 | 0
1-0-1 | 0
0-1-1 | 0
1-1-1 | 0
```

Question 7

Écrire une fonction

```
bool tautologie(formule *f);
```

qui indique si `f` est une tautologie.

Corrigé

On teste les 2^n valuations, en s'arrêtant à la première qui rend `f` fausse. On n'oublie pas de libérer la valuation avant de renvoyer le résultat.

```
bool *valuation_nulle(int n)
{
    bool *v = malloc(n * sizeof(bool) + 1); // + 1 : évite un malloc(0)
    for(int i = 0; i < n; i++)
        v[i] = false;
    return v;
}

bool tautologie(formule *f)
{
    int n = num_variables(f);
    bool *v = valuation_nulle(n);
    bool resultat = true;
    do {
        if (!evalue(f, v)) {
            resultat = false;
            break;
        }
    } while(suivante(n, v));
    free(v);
    return resultat;
}
```

Cette fonction est en $O(2^n \mid f \mid)$ où $\mid f \mid$ est le nombre total de littéraux de `f`.

Question 8

Écrire une fonction

```
bool *satisfiable(formule *f);
```

qui renvoie une valuation modèle de `f` si `f` est satisfiable et qui renvoie `NULL` si `f` n'est pas satisfiable.

Corrigé

```
bool *satisfiable(formule *f)
{
    int n = num_variables(f);
    bool *v = valuation_nulle(n);
    do {
        if (evalue(f, v))
            return v;
    } while(suivante(n, v));
    free(v);
    return NULL;
}
```

La valuation renvoyée est allouée sur le tas : c'est à l'appelant de la libérer. Pour l'exemple, on obtient la valuation nulle.

Question 9

Écrire une fonction

```
bool consequence(formule *f, formule *g);
```

qui indique si `f` est conséquence logique de `g`.

Corrigé

f est conséquence logique de g ($g \models f$) si et seulement si tout modèle de g est un modèle de f. On énumère donc les valuations sur les variables des deux formules, à la recherche d'un contre-exemple.

```
int max(int a, int b)
{
    return a > b ? a : b;
}

bool consequence(formule *f, formule *g)
{
    int n = max(num_variables(f), num_variables(g));
    bool *v = valuation_nulle(n);
    bool resultat = true;
    do {
        if (evalue(g, v) && !evalue(f, v)) {
            resultat = false;
            break;
        }
    } while(suivante(n, v));
    free(v);
    return resultat;
}
```

Par exemple, $\neg X_2$ est conséquence de la formule f de la question 1, mais pas l'inverse ; et toute formule est conséquence d'une formule insatisfiable comme $X_0 \wedge \neg X_0$.

47

TRAVAUX PRATIQUES 47

SAT-solver

DANS CE CHAPITRE

47.1 OCaml : lexer et parser
47.2 Résolution en C

On va réaliser dans ce TP un SAT-solver en deux parties :

- un lexer/parser en OCaml qui affiche une formule en forme normale conjonctive (FNC) dans une syntaxe très simple ;
- un solveur en C.

On reprend les définitions du chapitre Calcul propositionnel : un **littéral** est une variable ou la négation d'une variable, une **clause** est une disjonction de littéraux et une **FNC** est une conjonction de clauses. Le problème SAT consiste à déterminer si une formule, ici en FNC, est satisfiable.

47.1 OCaml : lexer et parser

47.1.1 Formules

On considère un type de formules logiques :

```
type formule =
      Var of string
    | Non of formule
    | Ou of formule * formule
    | Et of formule * formule
```

Question 1 Écrire une fonction permettant d'afficher une telle formule.

Question 2 Écrire une fonction qui renvoie la liste des variables présentes dans la formule, sans répétitions.

Question 3 Écrire une fonction qui vérifie si une formule est en forme normale conjonctive avec associativité à droite.

Note : on dira donc ici que $A \wedge (B \wedge C)$ est en FNC mais pas $(A \wedge B) \wedge C$. De même, on demande que les clauses soient de la forme $l_1 \vee (l_2 \vee (\ldots \vee l_k))$ où les l_i sont des littéraux.

47.1.2 Forme normale conjonctive

On va représenter une formule en forme normale conjonctive sous la forme d'un `int list list` (la liste des clauses, chaque clause étant la liste de ses littéraux) où on considère une énumération $v_1, \ldots, v_n$ des variables et où un littéral v_i est représenté par l'entier i et un littéral $\neg v_i$ par l'entier $-i$.

Question 4 En considérant l'énumération des variables donnée par votre fonction précédente et en supposant que la formule est en FNC, afficher la formule sous forme textuelle dans le format :

```
nombredeclauses
nombredelitclause1 lit lit ... lit
...
nombredelitclausen lit lit ... lit
```

Par exemple, $A \wedge (B \vee \neg C)$ dans l'énumération A, B, C deviendra :

```
2
1 1
2 2 -3
```

Remarque 47.1 C'est une version simplifiée du format DIMACS, utilisé par les SAT-solveurs réels, où chaque clause est terminée par un `0` au lieu d'être précédée de sa longueur.

47.1.3 Lexer

On va considérer une représentation infixe parenthésée des formules en utilisant `/\` pour et, `\/` pour ou, `~` pour non, où on ignore les espaces et sauts de ligne et où les parenthèses `(` et `)` servent à regrouper une sous-formule.

On définit donc le type `lexeme`

```
type lexeme =
      LNon | LOu | LEt | LLpar | LRpar
    | LVar of string
```

et, pour manipuler un flot de caractères dans une chaîne :

```
type lexbuf = {
    contenu : string;
    mutable curseur : int
}

let lexbuf_init s = { contenu = s; curseur = 0 }

let lexbuf_regard l j = l.contenu.[l.curseur + j]
let lexbuf_avance l k = l.curseur <- l.curseur + k
let lexbuf_fini l = l.curseur >= String.length l.contenu
exception FinLexbuf
let lexbuf_recup l =
    if lexbuf_fini l
    then raise FinLexbuf;
    let c = lexbuf_regard l 0 in
    lexbuf_avance l 1;
    c

exception ErreurLexicale
```

Question 5 Écrire une fonction `lexer : string -> lexeme list` qui découpe une chaîne en lexèmes. En cas d'erreur, on produira l'exception `ErreurLexicale`.

47.1.4 Parser

Question 6 Écrire une fonction `parser : lexeme list -> formule * lexeme list` qui lit une formule depuis un flot de lexèmes et renvoie les lexèmes non utilisés.

En cas d'erreur de syntaxe, on produira l'exception `ErreurSyntaxique`.

Pour que les formules lues soient en FNC au sens de la question 3, on lira `A /\ B /\ C` comme $A \wedge (B \wedge C)$, et de même pour `\/`. On pourra supposer que `/\` et `\/` ne sont jamais mélangés sans parenthèses, comme dans l'exemple ci-dessous.

47.1.5 Le frontend

On pourra ainsi utiliser le code suivant

```
let main =
    let s = read_line () in
    let lex = lexer s in
    let f, q = parser lex in
    assert(q = []);
    assert(est_en_cnf f);
    affiche_cnf (cnf f)
```

pour lire une formule et l'afficher, où `cnf` calcule la représentation par un `int list list` et `affiche_cnf` réalise l'affichage de la question 4.

Dans la suite, on va écrire un programme `resout` et, si le programme OCaml est `parser`, on pourra exécuter dans le shell :

```
echo "A /\ (B \/ ~C) /\ (~A \/ ~B)" | ./parser | ./resout
```

pour enchaîner les opérations. Ici, `./parser` affiche :

```
3
1 1
2 2 -3
2 -1 -2
```

Question 7 Écrire des jeux de tests.

47.2 Résolution en C

47.2.1 Lecture des données

Question 8 Écrire un programme en C qui lit une formule en FNC depuis l'entrée standard.

On pourra la stocker sous la forme d'un `int**` et du nombre de clauses. Pour les clauses, on pourra utiliser une sentinelle `0` pour terminer les listes.

47.2.2 Résolution par retour sur trace

Question 9 Écrire un programme qui cherche si la formule est satisfiable par retour sur trace.

On pourra s'inspirer de l'algorithme de Quine : on affecte les variables une par une et on revient en arrière dès qu'une clause a tous ses littéraux faux.



> **Remarque 47.2** Les SAT-solveurs réels reposent sur l'algorithme DPLL, qui ajoute deux règles à ce retour sur trace : si une clause n'a plus qu'un littéral non affecté et aucun littéral vrai, ce littéral doit être rendu vrai (**propagation unitaire**) ; si une variable n'apparaît que positivement (ou que négativement) dans les clauses non encore satisfaites, on peut lui donner la valeur qui rend ces littéraux vrais (**littéral pur**). On pourra les ajouter pour accélérer la recherche.

47.2.3 Première parallélisation

Si la formule a $n + p$ variables, on peut lancer 2^n threads en parallèle avec chaque valuation des n premières variables pour explorer les 2^p complétions de valuations sur les variables restantes.

Question 10 Implémenter cette stratégie. La charge est-elle bien répartie ?

47.2.4 Parallélisation dynamique

Question 11 Mettre en place un paradigme producteur-consommateur pour paralléliser le problème plus efficacement.

PARTIE VII

Bases de données

48 Bases de données 707

CHAPITRE 48

Bases de données

DANS CE CHAPITRE

48.1 SELECT
48.2 Produit cartésien et jointures
48.3 Fonctions et requêtes imbriquées
48.4 Agrégation
48.5 TP

Les figures de ce chapitre utilisent toutes la même table T, d'attributs `a`, `b` et `c`, où les valeurs de `c` sont représentées par des couleurs.

48.1 SELECT

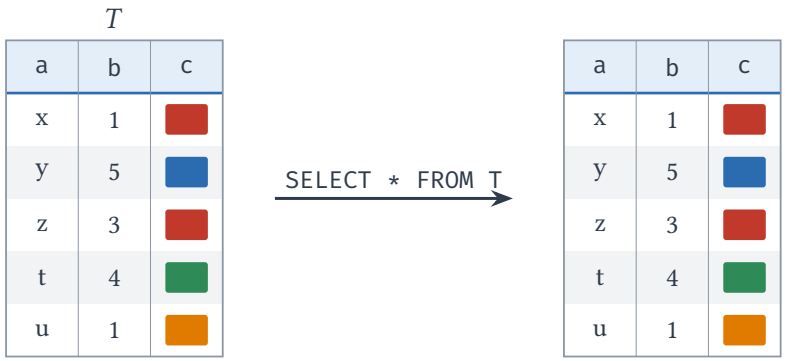


Figure 48.1 — SELECT * FROM T

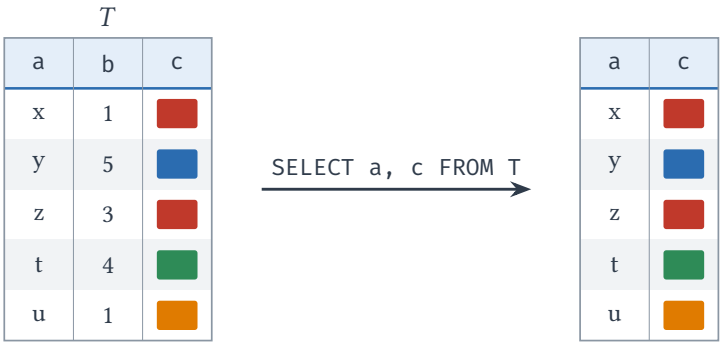


Figure 48.2 — SELECT a, c FROM T



Figure 48.3 — SELECT a AS d, c FROM T



Figure 48.4 — SELECT * FROM T WHERE c = rouge

48.2 Produit cartésien et jointures

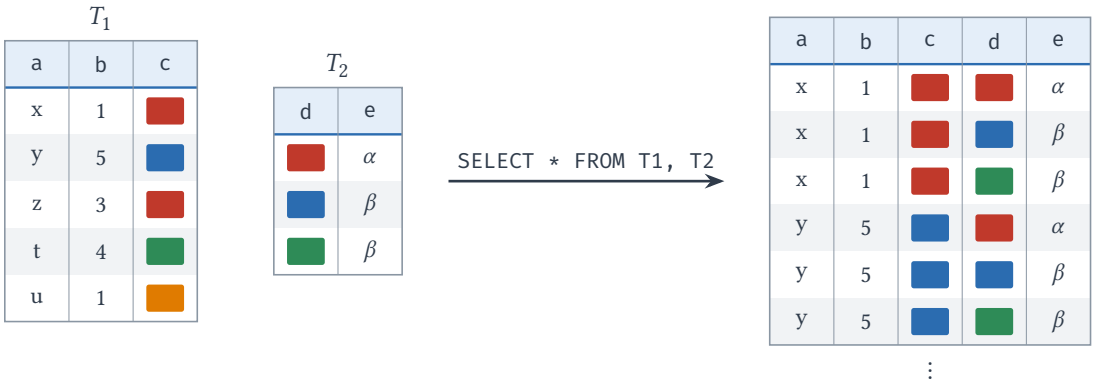


Figure 48.5 — SELECT * FROM T1, T2

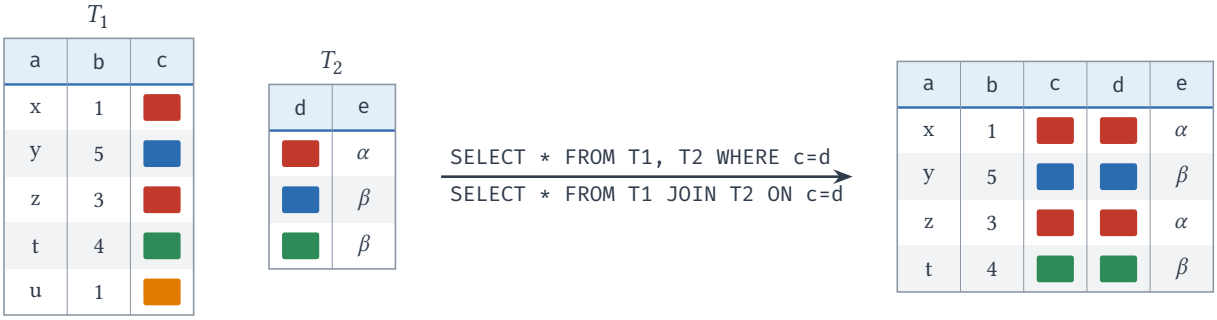


Figure 48.6 — SELECT * FROM T1 JOIN T2 ON c=d

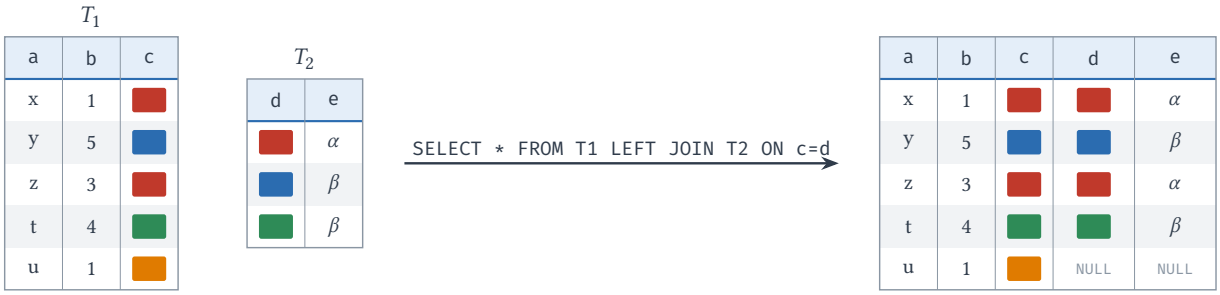


Figure 48.7 — SELECT * FROM T1 LEFT JOIN T2 ON c=d

48.3 Fonctions et requêtes imbriquées



Figure 48.8 — SELECT COUNT(*) FROM T



Figure 48.9 — SELECT COUNT(*) AS n FROM T

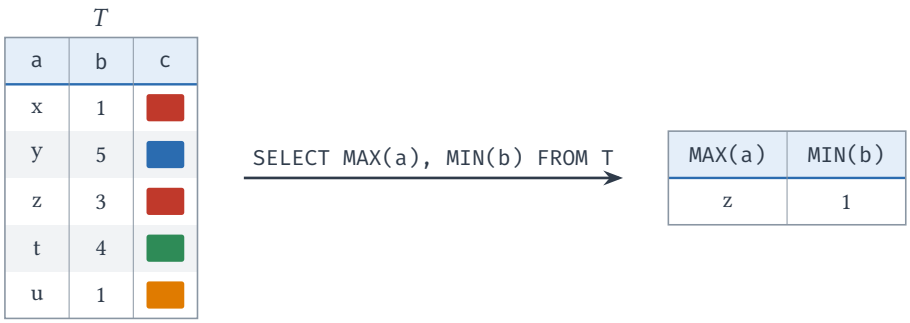


Figure 48.10 — SELECT MAX(a), MIN(b) FROM T

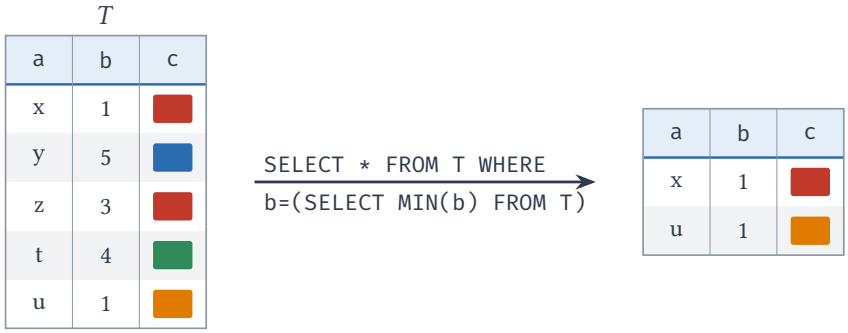


Figure 48.11 — SELECT * FROM T WHERE b=(SELECT MIN(b) FROM T)

48.4 Agrégation

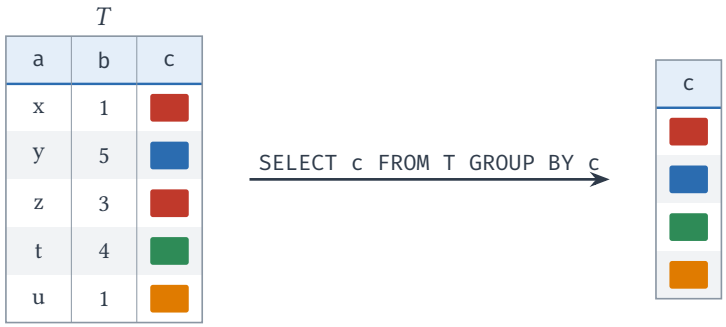


Figure 48.12 — SELECT c FROM T GROUP BY c



Figure 48.13 — SELECT c, COUNT(c) FROM T GROUP BY c

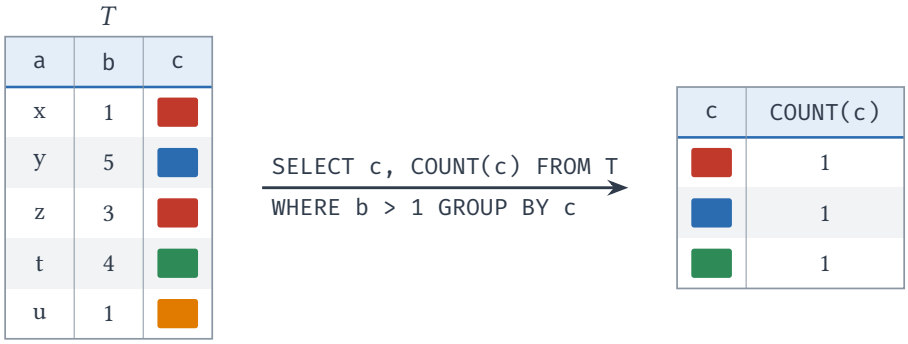


Figure 48.14 — SELECT c, COUNT(c) FROM T WHERE b > 1 GROUP BY c

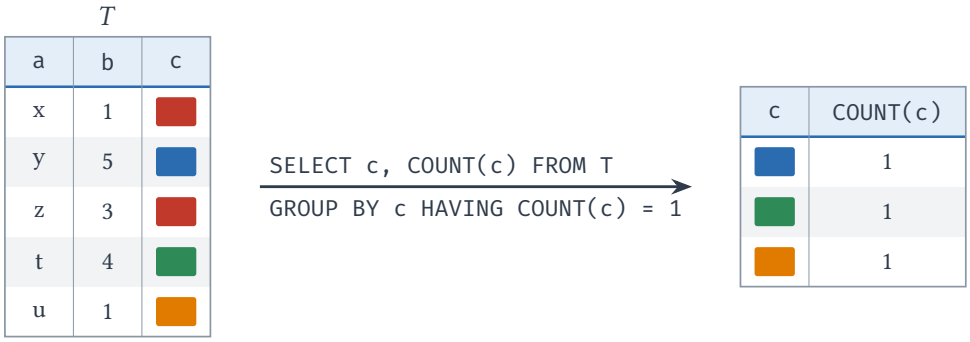


Figure 48.15 — SELECT c, COUNT(c) FROM T GROUP BY c HAVING COUNT(c) = 1

48.5 TP

Allez sur le site http://sql.classeprepa.net et identifiez-vous avec le couple utilisateur/mot de passe suivant : mp2i/mp2i.

48.5.1 Musique

- Sélectionnez `tp26_musique` dans la barre de navigation à gauche de l'écran.
- Cliquez sur l'onglet `SQL`. On vous demande dans la suite de taper des requêtes et d'appuyer sur `Exécuter` pour les voir s'exécuter. Après la première requête, on peut faire `Éditer en ligne` pour aller plus vite.
- On vous recommande de copier les réponses dans un fichier `tp26.sql` pour en garder une trace.

La base de données `tp26_musique` comporte quatre tables dont le schéma est donné dans la figure suivante :

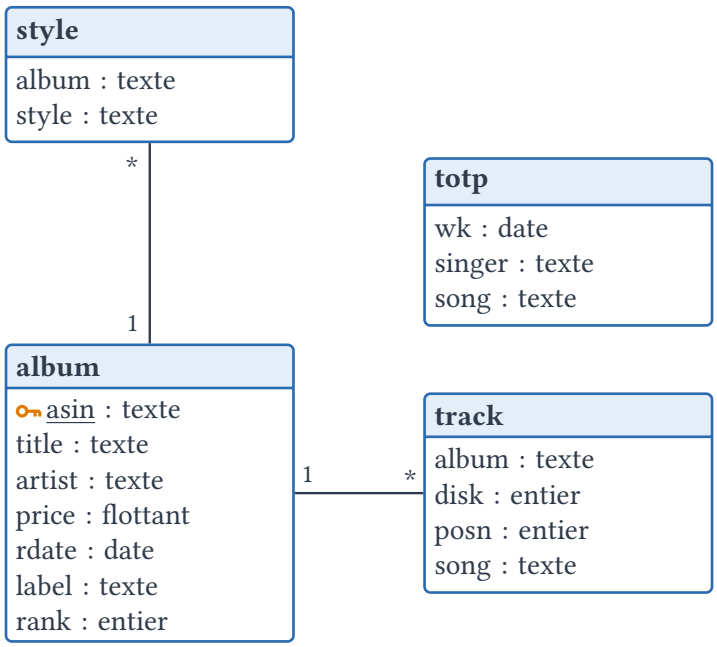


Figure 48.16 — Schéma de la base tp26_musique

48.5.2 Premières requêtes

L'attribut `asin` de `album` est la clé primaire qui sert de clé étrangère `album` dans `track` et `style`.

Question 1

Cherchez tous les albums de `'U2'`.

Ici, les dates sont au format texte `'yyyy-mm-dd'` qui permet d'effectuer des comparaisons directement avec `<`, `<=`, `>`, `>=`.

Par exemple, `'1981-08-05'` est le 5 août 1981.

Question 2

Cherchez tous les albums parus pendant l'année 1991.

Les données sont issues d'un célèbre vendeur en ligne, à un instant précis dans le passé, c'est en ce sens qu'il faut interpréter l'attribut `price` (c'est le prix en dollars) et l'attribut `rank` (c'est le rang dans les ventes de disques).

L'attribut `price` n'est pas toujours renseigné. Dans ce cas-là, il prend la valeur spéciale `NULL`. Pour tester si une valeur est `NULL` on utilise l'opérateur spécial `IS NULL` (et `IS NOT NULL` pour le test contraire) : une comparaison `price = NULL` n'est jamais vraie.

Question 3

Cherchez tous les albums dont le prix est renseigné.

Question 4

Trouvez le titre du 5e album le plus cher **et uniquement de celui-ci**.

Question 5

Affichez une table contenant pour chaque chanson, son nom et le nom de l'album qui la contient.

L'attribut `disk` indique le numéro du disque sur l'album, la plupart en contiennent un seul, et `posn` le numéro de la piste sur ce disque.

Question 6

Affichez une table contenant les titres des albums ayant au moins deux disques.

On fera en sorte qu'il n'y ait pas de titres répétés.

La table `totp` contient des chansons ayant apparu dans le classement mensuel des meilleures ventes. Apparemment, les tables `track` et `totp` ne sont pas associées.

Question 7

À l'aide de `LEFT JOIN` déterminez s'il y a des éléments de `totp` dont l'attribut `song` n'est pas présent dans `track`. Même question en inversant les tables.

Est-il pour autant possible de considérer que `song` est une clé primaire d'une de ces deux tables et que l'autre est une clé étrangère ?

Fonctions

On peut appliquer des fonctions sur un ou plusieurs attributs, ou expressions, pour obtenir des valeurs. Ce faisant, on va déduire une table à une ligne contenant les résultats.

Par exemple :

```
SELECT COUNT(*) FROM album
```

va renvoyer le nombre de lignes de la table `album`.

Les fonctions au programme sont :

- `COUNT` qui permet de trouver la taille des résultats. Attention, `COUNT(attr)` ne compte pas les valeurs distinctes : il compte les lignes où `attr` n'est pas `NULL` (pour les valeurs distinctes, on écrit `COUNT(DISTINCT attr)`). Pour compter les lignes, on utilisera `COUNT(*)`.

- `MAX` et `MIN` qui renvoient le maximum ou le minimum d'un attribut ou d'une expression.
- `SUM` qui fait la somme des valeurs d'un attribut ou d'une expression sur les lignes.
- `AVG` qui calcule la moyenne.

Les valeurs `NULL` sont ignorées par toutes ces fonctions, sauf `COUNT(*)`.

Question 8

En une seule requête, calculer le prix moyen d'un album et le plus petit rang.

Requêtes imbriquées

Attention, la requête

```
SELECT title, MIN(rank) FROM album
```

n'a aucune raison de renvoyer le titre de l'album de plus petit rang. En fait, avec un gestionnaire peu permissif, la requête précédente est invalide. (SQLite l'accepte et renvoie, par exception, une ligne réalisant le minimum, mais ce comportement n'est pas standard.)

Une manière de réaliser cela est d'utiliser une requête imbriquée comme :

```
SELECT title, rank FROM album WHERE rank = (SELECT MIN(rank) FROM album)
```

Ici, comme la sous-requête `SELECT MIN(rank) FROM album` renvoie un nombre, on peut tester cela dans la première requête.

On peut noter qu'il est possible de réaliser une telle requête avec une jointure externe à gauche comme

```
SELECT A1.title, A1.rank
FROM album AS A1 LEFT JOIN album AS A2
ON A2.rank < A1.rank
WHERE A2.rank IS NULL
```

mais cette requête n'est pas très naturelle.

Question 9

Expliquez cette requête et pourquoi elle répond à la question.

Que se passe-t-il pour les albums dont le rang n'est pas renseigné (`rank` à `NULL`) ? Comment corriger la requête ?

Question 10

Déterminer en une requête les albums ayant le plus de pistes sur un de leurs disques.

Agrégation

Pour grouper les chansons dans la table `track` selon l'album qui les contient on peut utiliser la requête :

```
SELECT album FROM track GROUP BY album
```

On obtient alors une ligne pour chaque valeur de `album` présente dans la table. On parle d'agrégat ou de paquet. Cela équivaut ici à

```
SELECT DISTINCT album FROM track
```

Une règle importante en SQL est qu'il est interdit de projeter des attributs qui ne sont pas compris dans le `GROUP BY` (en dehors d'une fonction d'agrégation). En effet, il y a ambiguïté sur la valeur qu'on pourrait choisir pour ces attributs.

Là où les `GROUP BY` sont très utiles c'est qu'on peut calculer des fonctions sur chaque agrégat, qui elles sont non ambiguës.

Ici la piste de numéro maximum sur un album indique le nombre de pistes de l'album (pour un album à un seul disque ; sinon, c'est le nombre de pistes de son plus long disque) :

```
SELECT album, MAX(posn) FROM track GROUP BY album
```

Question 11

Déterminer le numéro moyen d'une piste d'un album, tout disque compris.

Si on souhaite obtenir avec un `GROUP BY` les albums ayant au moins 10 pistes, on pourrait être tenté d'écrire

```
SELECT album FROM track GROUP BY album WHERE MAX(posn) >= 10
```

Cette requête est invalide. Le `WHERE` doit être écrit avant le `GROUP BY`, et surtout il s'effectue avant le regroupement : il filtre les lignes de la table, et ne peut donc pas porter sur une fonction d'agrégation comme `MAX(posn)`. `SQL` fournit une syntaxe permettant de filtrer sur des conditions dans les agrégats : `HAVING`. Ainsi, on écrira :

```
SELECT album FROM track GROUP BY album HAVING MAX(posn) >= 10
```

On aimerait aussi avoir le nom de l'album, pour cela, il y a plusieurs solutions, une des solutions les plus simples est d'utiliser `MAX(title)` sur un agrégat : tous les titres d'un même agrégat sont égaux, `MAX` sert uniquement à en choisir un de manière non ambiguë.

```
SELECT MAX(title) FROM track JOIN album
ON track.album = asin
GROUP BY track.album
HAVING MAX(posn) >= 10
```

On peut le faire avec une requête imbriquée dans un `JOIN` interne qui ne va donc sélectionner que les albums présents dans les deux tables. Ici, le renommage est obligatoire pour que la requête soit valide.

```
SELECT title
FROM album
JOIN (SELECT album FROM track
    GROUP BY album HAVING MAX(posn) >= 10) AS T
ON asin = T.album
```

On pourrait aussi utiliser une requête imbriquée et l'opérateur `IN` hors programme pour tester la présence d'une valeur dans une table de résultat.

```
SELECT title FROM album
WHERE asin IN (SELECT album FROM track
    GROUP BY album HAVING MAX(posn) >= 10)
```

Question 12

Trouver les noms des albums ayant au moins deux disques.

48.5.3 Films

On considère maintenant la base de données `tp26_film`. Ses tables sont décrites dans le diagramme :

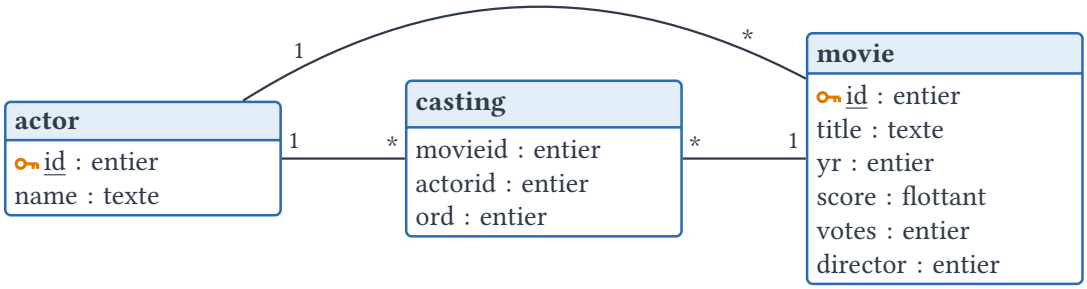


Figure 48.17 — Schéma de la base tp26_film

Quelques précisions :

- Les clés `id` permettent d'identifier de manière unique un film (`movie`) ou un acteur (`actor`).
- L'attribut `director` fait référence à un id d'`actor` qui est également réalisateur de ce film (un pur réalisateur peut ne pas être un vrai acteur mais être présent dans la table `actor` quand même).
- L'entier `ord` indique l'ordre dans lequel sont énumérés les acteurs dans le casting d'un film : du premier nom de valeur 1 (le plus important) au dernier.
- `score` est le score des internautes sur le film.
- `votes` est le nombre de votes.
- `yr` est un entier à quatre chiffres comme 1977.

On va rajouter un opérateur qui est pratique pour le TP mais pas au programme : `LIKE` qui permet de comparer des textes avec un joker, le caractère `%`, qui signifie n'importe quelle suite de caractères (éventuellement vide).

Ainsi,

```
SELECT title FROM movie WHERE title LIKE '%, The'
```

va sélectionner les films dont le nom finit par `, The`.

Question 13

Déterminer les noms des acteurs ayant joué dans des films dont le nom commence par Star.

Question 14

Déterminer le film qui a le plus d'acteurs au casting.

Question 15

Déterminer les noms des acteurs ayant tourné dans au moins 5 films tout en étant toujours premier au casting.

Question 16

Déterminer les noms des acteurs qui n'ont tourné qu'avec un seul réalisateur.

Question 17

Déterminer les noms des acteurs qui n'ont tourné que dans des films qu'ils ont réalisés.

PARTIE VIII

Problèmes

49	Problème	DM - Enveloppe convexe	719
50	Problème	Dungeons and Diagrams	731
51	Problème	Dungeons and Diagrams - les 64 grilles	759
52	Problème	Jeux et nombres	783

49

PROBLÈME 49

DM - Enveloppe convexe

DANS CE CHAPITRE

49.1 **Préliminaires**
49.2 **Algorithme du paquet cadeau**
49.3 **Intermède : piles d'entiers**
49.4 **Algorithme de balayage**

Problème qui reprend un sujet de tronc commun de l'X en l'adaptant au C. Donné en DM 1 en MP2I 2021-2022, puis en DM 2 en MP2I 2022-2023.

> **Remarque 49.1** À rendre sur feuille pour la rentrée des vacances de la Noël. Les programmes pourront être donnés sous forme imprimée plutôt qu'écrits à la main. Vous êtes très fortement encouragé à faire un programme complet réalisant le sujet : en testant toutes vos fonctions !
>
> On pourra notamment faire une interface graphique avec raylib.

Ce sujet a pour objectif de calculer des enveloppes convexes de nuages de points dans le plan affine, un grand classique en géométrie algorithmique. On dit qu'un ensemble $C \subset \mathbb{R}^2$ est convexe si et seulement si pour toute paire de points $p, q \in C$, le segment de droite $[p, q]$ est inclus dans C. L'enveloppe convexe d'un ensemble $P \subset \mathbb{R}^2$, notée $Conv(P)$, est le plus petit convexe contenant P. Dans le cas où P est un ensemble fini (appelé nuage de points), le bord de $Conv(P)$ est un polygone convexe dont les sommets appartiennent à P, comme illustré dans la figure 1.

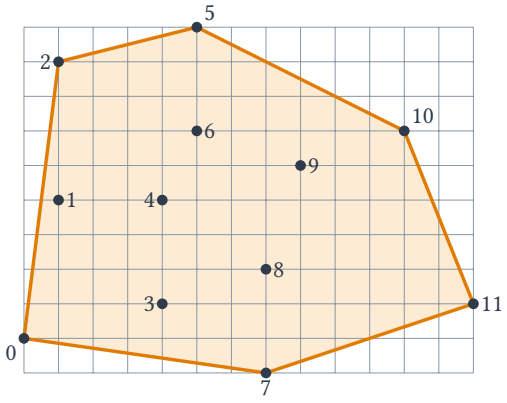


Figure 49.1 — Figure 1 : un nuage de points, numérotés de 0 à 11, et le bord de son enveloppe convexe.

Le calcul de l'enveloppe convexe d'un nuage de points est un problème fondamental en informatique, qui trouve des applications dans de nombreux domaines comme :

- la robotique, par exemple pour l'accélération de la détection de collisions dans le cadre de

- le traitement d'images et la vision, par exemple pour la détection d'objets convexes (comme des plaques minéralogiques de voiture) dans des scènes 2d,
- l'informatique graphique, par exemple pour l'accélération du rendu de scènes 3d par lancer de rayons,
- la théorie des jeux, par exemple pour déterminer l'existence d'équilibres de Nash,
- la vérification formelle, par exemple pour déterminer si une variable risque de dépasser sa capacité de stockage ou d'atteindre un ensemble de valeurs interdites lors de l'exécution d'une boucle dans un programme, et bien d'autres encore.

Dans ce sujet nous allons écrire deux algorithmes de calcul du bord de l'enveloppe convexe d'un nuage de points P dans le plan affine. Le premier, dit algorithme du paquet cadeau, consiste à envelopper le nuage de points P progressivement en faisant pivoter une droite tout autour. Le deuxième, dit de balayage, consiste à balayer le plan horizontalement avec une droite verticale, tout en maintenant au fur et à mesure l'enveloppe convexe de la partie du nuage située à gauche de cette droite verticale. Les deux algorithmes sont illustrés respectivement dans les figures 3 et 4.

Le temps d'exécution du premier algorithme est majoré par une constante fois nm, celui du deuxième par une constante fois $n \log n$, où n désigne le nombre total de points de P et m le nombre de points de P appartenant au bord de $Conv(P)$. Rappelons que le temps d'exécution d'un programme A (fonction ou procédure) est le nombre d'opérations élémentaires (comparaisons, additions, soustractions, multiplications, divisions, affectations, etc.) nécessaires à l'exécution de A. Sauf mention contraire dans l'énoncé du sujet, le candidat n'aura pas à justifier des temps de calcul de ses programmes. Toutefois, il devra veiller à ce que ces derniers ne dépassent pas les bornes prescrites.

On va représenter les points par des tableaux de deux entiers `{ x, y }`.

Dans toute la suite on supposera que le nuage de points P est de taille $n \geq 3$ et en position générale, c'est-à-dire qu'il ne contient pas 3 points distincts alignés. Ces hypothèses vont permettre de simplifier les calculs en ignorant les cas pathologiques, comme par exemple la présence de 3 points alignés sur le bord de l'enveloppe convexe. Nos programmes prendront en entrée un nuage de points P dont les coordonnées sont stockées dans un tableau `tab` de tableaux de 2 entiers, comme dans l'exemple ci-dessous qui contient les coordonnées du nuage de points de la figure 1 :

```
int tab[][2] = {
    { 0, 0 }, { 1, 4 }, { 1, 8 }, { 4, 1 }, { 4, 4 }, { 5, 9 },
    { 5, 6 }, { 7, -1 }, { 7, 2 }, { 8, 5 }, { 11, 6 }, { 13, 1 }
};
```

Précisons que les coordonnées, supposées entières, sont données dans une base orthonormée du plan, orientée dans le sens direct.

49.1 Préliminaires

Question 1 Écrire une fonction `int plus_bas(int tab[][2], int n)` qui prend en paramètre le tableau `tab` de n points et qui renvoie l'indice j du point le plus bas (c'est-à-dire de plus petite ordonnée) parmi les points du nuage P. En cas d'égalité, votre fonction devra renvoyer l'indice du point de plus petite abscisse parmi les points les plus bas.

Corrigé

```
int plus_bas(int tab[][2], int n)
{
    int i = 0;

    for (int j = 1; j < n; j++)
    {
        if (tab[j][1] < tab[i][1]
```

```
            || (tab[j][1] == tab[i][1] && tab[j][0] < tab[i][0]))
            i = j;
    }

    return i;
}
```

Question 2 Prouver la correction de cette fonction.

Corrigé

On note y_i l'ordonnée de `tab[i]` et x_i son abscisse, et on note $\underset{lex}{\leq}$ l'ordre lexicographique sur les couples (y, x) : le point cherché est l'indice i tel que (y_i, x_i) soit minimal pour cet ordre.

On a l'invariant de boucle $P(i, j) = \forall k \in [\![0, j-1]\!], (y_i, x_i) \underset{lex}{\leq} (y_k, x_k)$.

- **Initialisation** : en début de boucle, j vaut 1, i vaut 0 et $P(0, 1)$ est vérifié directement.
- **Hérédité** : si l'invariant $P(i, j)$ est vérifié en début de boucle, alors
  - soit $(y_i, x_i) \underset{lex}{\leq} (y_j, x_j)$, i ne change pas et on a bien $P(i, j+1)$;
  - soit $(y_j, x_j) \underset{lex}{<} (y_i, x_i)$ (c'est exactement la condition du `if`) et alors on passe à $P(j, j+1)$ en fin de boucle : pour $k \in [\![0, j]\!]$, soit $k < j$ et ainsi $(y_k, x_k) \underset{lex}{\geq} (y_i, x_i) \underset{lex}{>} (y_j, x_j)$, soit $k = j$. Dans tous les cas $(y_k, x_k) \underset{lex}{\geq} (y_j, x_j)$.

L'invariant est donc valide et ainsi en sortie de boucle on a

$$P(i, n) = \forall k \in [\![0, n-1]\!], (y_i, x_i) \underset{lex}{\leq} (y_k, x_k)$$

c'est-à-dire que p_i est le plus bas et, parmi les plus bas, celui de plus petite abscisse. La terminaison est immédiate (boucle `for` à $n-1$ itérations).

Sur le tableau exemple précédent, le résultat de la fonction doit être l'indice 7.

Dans la suite nous aurons besoin d'effectuer un seul type de test géométrique : celui de l'orientation.

Définition 49.2 — 1

Étant donnés trois points p_i, p_j, p_k du nuage P, distincts ou non, le test d'orientation renvoie $+1$ si la séquence (p_i, p_j, p_k) est orientée positivement, -1 si elle est orientée négativement, et 0 si les trois points sont alignés (c'est-à-dire si deux au moins sont égaux, d'après l'hypothèse de position générale).

Pour déterminer l'orientation de (p_i, p_j, p_k), il suffit de calculer l'aire signée du triangle, comme illustré sur la figure 2. Cette aire vaut

$$\frac{1}{2}\big((y_k - y_i)(x_j - x_i) - (x_k - x_i)(y_j - y_i)\big)$$

où p_a a pour coordonnées (x_a, y_a). **Comme on travaille avec des entiers, on évitera de faire la division par 2 qui peut aboutir à 0 et ne préserve pas le signe strict.**

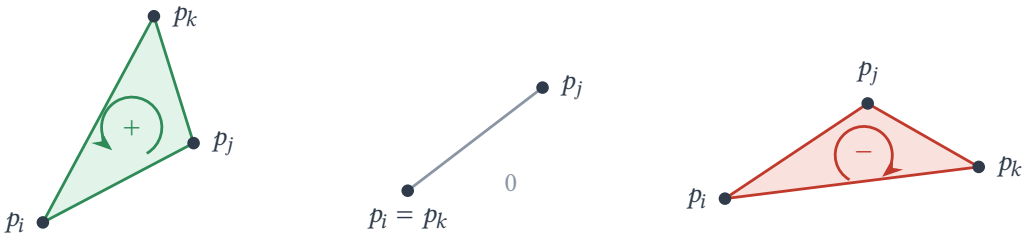


Figure 49.2 — Figure 2 : test d'orientation sur la séquence (p_i, p_j, p_k) : positif à gauche, nul au centre, négatif à droite.

Question 3 Sur le tableau exemple précédent, donner le résultat du test d'orientation pour les choix d'indices suivants :

1. $i = 0, j = 3, k = 4$,
2. $i = 8, j = 9, k = 10$.

Corrigé

1. Positif.
2. Négatif.

Question 4 Écrire une fonction `int orient(int tab[][2], int i, int j, int k)` qui prend en paramètres le tableau `tab` et trois indices, potentiellement égaux, et qui renvoie le résultat (-1, 0 ou $+1$) du test d'orientation sur la séquence (p_i, p_j, p_k) de points de P.

Corrigé

```
int orient(int tab[][2], int i, int j, int k)
{
    int a = (tab[k][1]-tab[i][1])*(tab[j][0]-tab[i][0])
          - (tab[k][0]-tab[i][0])*(tab[j][1]-tab[i][1]);

    if (a < 0)
        return -1;
    if (a > 0)
        return 1;
    return 0;
}
```

49.2 Algorithme du paquet cadeau

Cet algorithme a été proposé par R. Jarvis en 1973. Il consiste à envelopper peu à peu le nuage de points P dans une sorte de paquet cadeau, qui à la fin du processus est exactement le bord de $Conv(P)$. On commence par insérer le point de plus petite ordonnée (celui d'indice 7 dans l'exemple de la figure 1) dans le paquet cadeau, puis à chaque étape de la procédure on sélectionne le prochain point du nuage P à insérer.

La procédure de sélection fonctionne comme suit. Soit p_i le dernier point inséré dans le paquet cadeau à cet instant. Par exemple, $i = 10$ dans l'exemple de la figure 3.

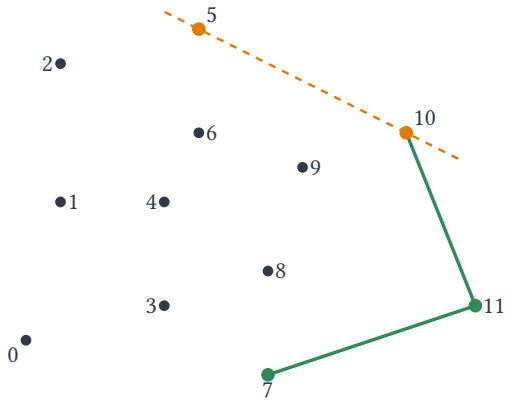


Figure 49.3 — Figure 3 : mise à jour du paquet cadeau après insertion du point p_{10}.

Considérons la relation binaire $\preccurlyeq$ définie sur l'ensemble $P \setminus \{p_i\}$ par :

$$p_j \preccurlyeq p_k \Leftrightarrow \texttt{orient(tab, i, j, k)} \leq 0$$

Question 5 Justifier brièvement le fait que $\preccurlyeq$ est une relation d'ordre total sur l'ensemble $P \setminus \{p_i\}$.

Corrigé

C'est une question de maths ça...

Si p, q, r sont des points, on note ici $A(p, q, r)$ l'aire signée du triangle donnée par la formule du sujet. On a $q \preccurlyeq r \Leftrightarrow A(p_i, q, r) \leq 0$.

La relation $\preccurlyeq$ est :

- **réflexive** : soit q un point, on a directement $A(p_i, q, q) = 0$ donc $q \preccurlyeq q$.
- **antisymétrique** : soit $q = (x, y), r = (x', y')$ des points tels que $q \preccurlyeq r$ et $r \preccurlyeq q$. On a $A(p_i, q, r) = \frac{1}{2}((y' - y_i)(x - x_i) - (x' - x_i)(y - y_i)) = -\frac{1}{2}((y - y_i)(x' - x_i) - (x - x_i)(y' - y_i)) = -A(p_i, r, q)$.

  Donc $A(p_i, q, r) \leq 0$ car $q \preccurlyeq r$ et $A(p_i, q, r) = -A(p_i, r, q) \geq 0$ car $r \preccurlyeq q$, donc $A(p_i, q, r) = 0$. Ainsi le triangle est d'aire nulle, donc il est plat, comme les points ne peuvent être alignés par hypothèse sur P, c'est forcément que $q = r$.
- **totale** : soit q, r des points, si on n'a pas $q \preccurlyeq r$, on a $A(p_i, q, r) > 0$ et donc $A(p_i, r, q) = -A(p_i, q, r) < 0$ donc on a bien $r \preccurlyeq q$. Dans tous les cas, soit $q \preccurlyeq r$, soit $r \preccurlyeq q$: la relation est bien totale.
- **transitive** : *assez compliqué en fait.* Ici, on peut supposer par construction que p_i est sur le bord de l'enveloppe convexe de P.

  On considère, par l'absurde, trois points p_j, p_k, p_l de $P \setminus \{p_i\}$ distincts tels que $p_j \preccurlyeq p_k$, $p_k \preccurlyeq p_l$ et $p_l \preccurlyeq p_j$ (si deux de ces points sont égaux, la transitivité découle de la réflexivité et de l'antisymétrie). Comme les trois points ne sont pas alignés, il est possible d'exprimer p_i comme un barycentre de ceux-ci : il existe a, b, c tels que $a + b + c = 1$ et $x_i = ax_j + bx_k + cx_l$, $y_i = ay_j + by_k + cy_l$.

  On montre alors par le calcul que $A(p_i, p_j, p_k) = cA(p_l, p_j, p_k)$, $A(p_i, p_k, p_l) = aA(p_l, p_j, p_k)$ et $A(p_i, p_l, p_j) = bA(p_l, p_j, p_k)$. Les trois aires de gauche sont strictement négatives (position générale) et $A(p_l, p_j, p_k) \neq 0$, donc a, b, c sont non nuls et de même signe ; comme $a + b + c = 1$, on a $a, b, c > 0$ et ainsi p_i est un point strictement intérieur au triangle de sommets p_j, p_k, p_l : il ne peut appartenir au bord de l'enveloppe convexe. Contradiction.

Ainsi, le prochain point à insérer (le point d'indice 5 dans la figure 3) est l'élément maximum pour la relation d'ordre $\preccurlyeq$. Il peut se calculer en temps linéaire (c'est-à-dire majoré par une constante fois n) par une simple itération sur les points de $P \setminus \{p_i\}$.

Question 6 Écrire une réalisation de la procédure. Elle prendra la forme d'une fonction

```
int prochain_point(int tab[][2], int n, int i);
```

qui prend en paramètre le tableau `tab` de n points ainsi que l'indice i du point inséré en dernier dans le paquet cadeau, et qui renvoie l'indice du prochain point à insérer. Le temps d'exécution de votre fonction doit être majoré par une constante fois n, pour tous n et i. La constante doit être indépendante de n et i, et on ne demande pas de la préciser.

Corrigé

```
int prochain_point(int tab[][2], int n, int i)
{
    int m = 0;

    if (i == 0) m = 1;
    for (int j = 1; j < n; j++)
    {
        if (j != i && orient(tab, i, m, j) <= 0)
            m = j;
    }

    return m;
}
```

Question 7 Décrire à la main le déroulement de la fonction `prochain_point` sur l'exemple de la figure 3. Plus précisément, indiquer la séquence des points de $P \setminus \{p_{10}\}$ considérés et la valeur de l'indice du maximum à chaque itération.

Corrigé

- Au départ $m = 0$.
- Pour $j = 1$, on a $\mathit{orient}(10, 0, 1) \leq 0$, donc $m = j = 1$.
- Pour $j = 2$, on a $\mathit{orient}(10, 1, 2) \leq 0$, donc $m = j = 2$.
- Pour $j = 3$ et 4, le test est > 0 donc rien ne change.
- Pour $j = 5$, on a $\mathit{orient}(10, 2, 5) \leq 0$, donc $m = j = 5$.
- Pour les valeurs suivantes de j ($6, 7, 8, 9, 11$; $j = 10$ est sauté), le test est > 0 donc rien ne change.

La fonction renvoie donc 5.

On peut maintenant combiner la fonction `prochain_point` avec la fonction `plus_bas` de la question 1 pour calculer le bord de l'enveloppe convexe de P. On commence par insérer le point p_i d'ordonnée la plus basse, puis on itère le processus de mise à jour du paquet cadeau jusqu'à ce que le prochain point à insérer soit de nouveau p_i. À ce moment-là on renvoie le paquet cadeau comme résultat sans insérer p_i une seconde fois.

Un détail technique : comme la taille du paquet cadeau augmente peu à peu lors du processus, et qu'à la fin elle peut être petite par rapport au nombre n de points de P, nous stockerons les indices des points du paquet cadeau dans un tableau de n entiers en renvoyant le nombre de points de l'enveloppe.

Par exemple, sur le nuage de la figure 1, on placera $7, 11, 10, 5, 2, 0$ dans les cases 0 à 5 du tableau d'enveloppe, sans toucher aux cases 6 à 11 de ce tableau, et on renverra la valeur 6 correspondant au nombre de points.

Question 8 Écrire une fonction

```
int conv_jarvis(int tab[][2], int n, int *env);
```

qui prend en paramètre le tableau `tab` de taille n représentant le nuage P, un tableau de même taille `env` et qui place dans `env` les indices du bord de l'enveloppe convexe de P, sans doublon, et renvoie le nombre de points

qui le constituent. Le temps d'exécution de votre fonction doit être majoré par une constante fois nm, où m est le nombre de points de P situés sur le bord de $Conv(P)$ (c'est la valeur renvoyée par la fonction).

Corrigé

```
int conv_jarvis(int tab[][2], int n, int *env)
{
    int taille = 1;
    env[0] = plus_bas(tab, n);

    int next = prochain_point(tab, n, env[0]);

    while (next != env[0])
    {
        env[taille] = next;
        taille += 1;
        next = prochain_point(tab, n, next);
    }

    return taille;
}
```

Question 9 Justifier brièvement le temps d'exécution de l'algorithme du paquet cadeau.

Corrigé

Si l'enveloppe convexe des n points en contient m, on effectue m appels à `prochain_point` qui fait chacun de l'ordre de n opérations élémentaires. Donc le coût de la boucle est de l'ordre de mn.

Le coût de l'initialisation avec `plus_bas` est de l'ordre de n.

Donc `conv_jarvis` est de coût temporel de l'ordre de mn, on notera qu'il est en $O(mn)$.

49.3 Intermède : piles d'entiers

Dans la suite nous aurons besoin d'utiliser des piles d'entiers, dont on donne la définition ci-dessous :

Définition 49.3 — 2

Une pile d'entiers est une structure de données permettant de stocker des entiers et d'effectuer les opérations suivantes en temps constant (indépendant de la taille de la pile) :

- créer une nouvelle pile vide,
- déterminer si la pile est vide,
- insérer un entier au sommet de la pile,
- déterminer la valeur de l'entier au sommet de la pile,
- retirer l'entier au sommet de la pile,
- détruire une pile existante.

Nous supposerons fournies les fonctions suivantes, qui réalisent les opérations ci-dessus et s'exécutent chacune en temps constant :

- `stack new_stack(void)`, qui ne prend pas d'argument et renvoie une pile vide,
- `bool is_empty(stack s)`, qui prend une pile `s` en argument et renvoie la valeur `true` ou `false` suivant que `s` est vide ou non,
- `void push(int i, stack s)`, qui prend un entier `i` et une pile `s` en argument, insère `i` au sommet de `s` (c'est-à-dire à la fin de la liste), et ne renvoie rien,
- `int top(stack s)`, qui prend une pile `s` (supposée non vide) en argument et renvoie la valeur de l'entier au sommet de `s` (c'est-à-dire le dernier élément empilé),

- `int pop(stack s)`, qui prend une pile `s` (supposée non vide) en argument, supprime l'entier au sommet de `s` et renvoie sa valeur,
- `void destroy_stack(stack s)`, qui libère l'espace utilisé par la pile `s`.

Dans la suite il est demandé aux candidats de manipuler les piles uniquement au travers de ces fonctions, sans aucune hypothèse sur la représentation effective des piles en mémoire.

Si on écrit

```
stack s = new_stack();
for (int i = 0; i < 3; i++)
{
    push(i, s);
    printf("%d\n", top(s));
}

for (int i = 0; i < 3; i++)
    printf("%d\n", pop(s));
destroy_stack(s);
```

on va alors afficher 0 1 2 à mesure qu'on empile, puis 2 1 0 en dépilant.

On vous donne ici une implémentation des piles d'entiers en C qui fonctionne tant qu'on empile moins de `N` éléments où `N` est une constante qu'on peut adapter à la pile considérée (le fameux **stack overflow** si on la dépasse !).

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <assert.h>

int N = 100;
struct stack_t {
    int *elements;
    int size;
};
typedef struct stack_t *stack;

stack new_stack(void)
{
    stack s = malloc(sizeof(struct stack_t));
    s->elements = malloc(sizeof(int)*N);
    s->size = 0;
    return s;
}

bool is_empty(stack s)
{
    return s->size == 0;
}

void push(int i, stack s)
{
    assert(s->size < N);
    s->elements[s->size] = i;
    s->size = s->size+1;
}

int top(stack s)
{
    assert(s->size != 0);
    return s->elements[s->size-1];
}

int pop(stack s)
{
```

```
    int i = top(s);
    s->size = s->size - 1;
    return i;
}

void destroy_stack(stack s)
{
    free(s->elements);
    free(s);
}

int main(void)
{
    stack s = new_stack();
    for (int i = 0; i < 3; i++)
    {
        push(i, s);
        printf("%d\n", top(s));
    }

    for (int i = 0; i < 3; i++)
        printf("%d\n", pop(s));

    destroy_stack(s);
}
```

49.4 Algorithme de balayage

Cet algorithme a été proposé par R. Graham en 1972. Nous allons écrire la variante (plus simple) proposée par A. Andrew quelques années plus tard.

La première étape consiste à trier les n points du nuage P par ordre croissant d'abscisse, en conservant tous les points de même abscisse dans un ordre arbitraire.

Question 10 Parmi les algorithmes de tri que vous connaissez, mentionnez-en un qui a un temps d'exécution majoré par une constante fois $n \log n$ sur les entrées de taille n.

Corrigé

Le tri fusion.

À partir de maintenant, on supposera que les points fournis en entrée sont triés par abscisse croissante, comme c'est le cas dans l'exemple du tableau `tab` donné au début du sujet.

L'idée de l'algorithme est de balayer le nuage de points horizontalement de gauche à droite par une droite verticale, tout en mettant à jour l'enveloppe convexe des points de P situés à gauche de cette droite, comme illustré dans la figure 4.

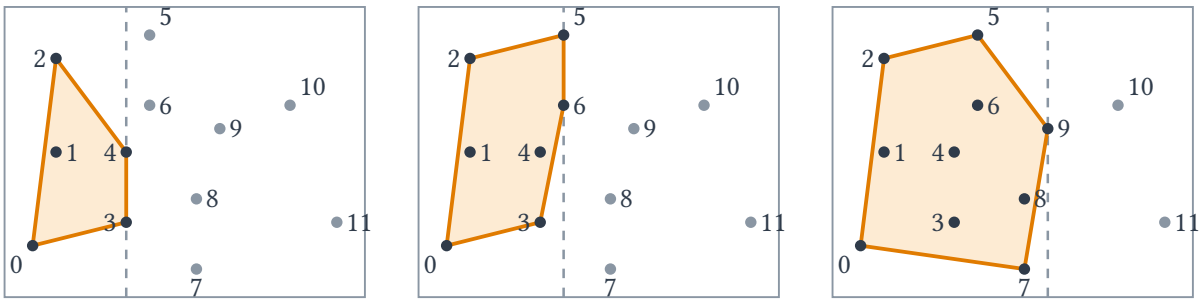


Figure 49.4 — Figure 4 : diverses étapes dans la procédure de balayage. La droite de balayage est en tirets.

Plus précisément, l'algorithme visite chaque point de P une fois, par ordre croissant d'abscisse (donc par ordre croissant d'indice dans le tableau `tab` car celui-ci est trié). À chaque nouveau point p_i visité, il met à jour le

bord de l'enveloppe convexe du sous-nuage $\{p_0, ..., p_i\}$ situé à gauche de p_i. On remarque que les points p_0 et p_i sont sur ce bord, et on appelle enveloppe supérieure la partie du bord de $Conv\{p_0, ..., p_i\}$ située au-dessus de la droite passant par p_0 et p_i (p_0 et p_i compris), et enveloppe inférieure la partie du bord de $Conv\{p_0, ..., p_i\}$ située au-dessous (p_0 et p_i compris). Le bord de $Conv\{p_0, ..., p_i\}$ est donc constitué de l'union de ces deux enveloppes, après suppression des doublons de p_0 et p_i.

Par exemple, dans le cas du nuage P de la figure 4 gauche, le sous-nuage $\{p_0, p_1, p_2, p_3, p_4\}$ a pour enveloppe supérieure la séquence (p_0, p_2, p_4) et pour enveloppe inférieure la séquence (p_0, p_3, p_4), le bord de son enveloppe convexe étant donné par la séquence (p_0, p_3, p_4, p_2).

Informatiquement, les indices des sommets des enveloppes inférieure et supérieure seront stockés dans deux piles d'entiers séparées, *ei* (pour enveloppe inférieure) et *es* (pour enveloppe supérieure).

La mise à jour de l'enveloppe supérieure est illustrée dans la figure 5 : tant que le point visité (p_9 dans ce cas) et les deux points dont les indices sont situés au sommet de la pile *es* (dans l'ordre : p_8 et p_5) forment une séquence (p_9, p_8, p_5) d'orientation négative (voir la définition 1 pour rappel de l'orientation), on dépile l'indice situé au sommet de *es* (8 dans ce cas). On poursuit ce processus d'élimination jusqu'à ce que l'orientation devienne positive ou qu'il ne reste plus qu'un seul indice dans la pile. L'indice du point visité (p_9 dans ce cas) est alors inséré au sommet de *es*. La mise à jour de l'enveloppe inférieure s'opère de manière symétrique.

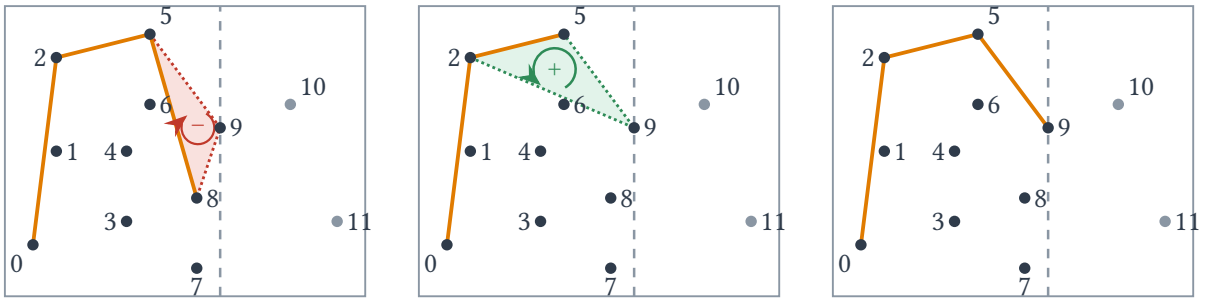


Figure 49.5 — Figure 5 : mise à jour de l'enveloppe supérieure lors de la visite du point p_9.

Question 11 Écrire une fonction

```
void maj_es(int tab[][2], stack es, int i);
```

qui prend en paramètre le tableau `tab` ainsi que la pile `es`, non vide, et l'indice *i* du point à visiter, et qui met à jour l'enveloppe supérieure du sous-nuage. Le temps d'exécution de votre fonction doit être majoré par une constante fois *i*.

Corrigé

```
void maj_es(int tab[][2], stack es, int i)
{
    int j = pop(es);
    while (!is_empty(es) && orient(tab, i, j, top(es)) <= 0)
    {
        j = pop(es);
    }
    push(j, es);
    push(i, es);
}
```

Question 12 Écrire une fonction

```
void maj_ei(int tab[][2], stack ei, int i);
```

qui effectue la mise à jour de l'enveloppe inférieure, avec le même temps d'exécution.

Corrigé

```
void maj_ei(int tab[][2], stack ei, int i)
{
    int j = pop(ei);
    while (!is_empty(ei) && orient(tab, i, j, top(ei)) >= 0)
    {
        j = pop(ei);
    }
    push(j, ei);
    push(i, ei);
}
```

Question 13 Écrire maintenant une fonction

```
stack conv_graham(int tab[][2], int n);
```

qui prend en paramètre le tableau `tab` de taille n représentant le nuage P, et qui effectue le balayage des points de P comme décrit précédemment. On supposera le tableau `tab` déjà trié par ordre croissant d'abscisse. La pile renvoyée devra contenir les indices des sommets du bord de $Conv(P)$ triés dans l'ordre positif d'orientation, à commencer par le point p_0.

Corrigé

```
stack conv_graham(int tab[][2], int n)
{
    stack es = new_stack();
    stack ei = new_stack();

    push(0, es);
    push(0, ei);

    for (int i = 1; i < n; i++)
    {
        maj_es(tab, es, i);
        maj_ei(tab, ei, i);
    }

    pop(es); // dernier sommet en commun
    while (!is_empty(es))
        push(pop(es), ei);
    pop(ei); // premier sommet en commun

    destroy_stack(es);
    return ei;
}
```

Par exemple, sur le nuage de la figure 1, le résultat de la fonction `conv_graham` doit être la pile `env` contenant la suite d'indices 0, 7, 11, 10, 5, 2 dans cet ordre, l'indice 0 se trouvant au fond de la pile `env` et l'indice 2 au sommet de `env`.

Question 14 Analyser brièvement le temps d'exécution de l'algorithme de balayage décrit précédemment, en supposant une fois encore que les points du nuage fourni en entrée sont déjà triés par abscisse croissante. En déduire que le temps d'exécution total de l'algorithme de Graham-Andrew est bien majoré par une constante fois $n \log n$.

Corrigé

Les seules opérations significatives ici sont les opérations manipulant les piles. L'algorithme effectue de l'ordre de n opérations élémentaires car chaque point est ajouté et retiré au plus une fois dans chaque pile (chaque tour de la boucle `while` de `maj_es` ou `maj_ei` dépile un indice qui avait été empilé auparavant).

Si on considère le tri initial, on effectue donc de l'ordre de $n \log n$ opérations élémentaires. On dira que l'algorithme est en $O(n \log n)$.

> **Remarque 49.4** Pour un nuage de points proche de son enveloppe convexe, cet algorithme est bien plus efficace que celui de Jarvis. Cas dégénéré pour Jarvis : n points sur un cercle, donc qui sont tous sur le bord de leur enveloppe convexe. Jarvis sera quadratique (en n^2) alors que Graham est quasi-linéaire (en $n \log n$).

50

PROBLÈME 50

Dungeons and Diagrams

DANS CE CHAPITRE

50.1 **Présentation du problème et de la résolution**
50.2 **Opérations bit à bit en C**
50.3 **Squelette du backtracking**
50.4 **Implémentation des différentes vérifications**
50.5 **Exemples**

Pour résoudre ce problème, on partira du fichier d_and_d.c.

50.1 Présentation du problème et de la résolution

50.1.1 D&D

Dungeons and Diagrams (D&D) est un casse-tête type *Sudoku* inventé par Zach Barth et présenté initialement sur son site en version papier et, plus récemment, au sein du jeu *Last Call BBS* de *Zachtronics*.

Une grille de D&D est la donnée d'une grille rectangulaire sur laquelle figurent des monstres 👹 et des trésors 💰, ainsi que des indications chiffrées sur les colonnes et les lignes. Le but est de remplir certaines cases vides par des murs en respectant les règles suivantes :

- Le nombre de murs sur une ligne (resp. une colonne) correspond au nombre associé à cette ligne (resp. cette colonne).
- Tous les monstres sont dans des impasses et toutes les impasses contiennent des monstres. Une impasse est une case ayant exactement une case voisine libre.
- Chaque trésor est dans une *salle au trésor* qui est un carré sans mur de taille 3x3 ne contenant aucun monstre et un seul trésor et qui, de plus, est bordé de murs sauf en exactement une case qui doit être libre.
- En dehors des salles au trésor, les couloirs ont une largeur d'une case : il n'y a pas de carré 2x2 de cases libres.
- Le graphe induit par la grille avec des connexions directes, mais pas en diagonale, est connexe.

On suppose que les grilles admettent une unique solution.

Ainsi, par exemple, une grille 6x6 et son unique solution :

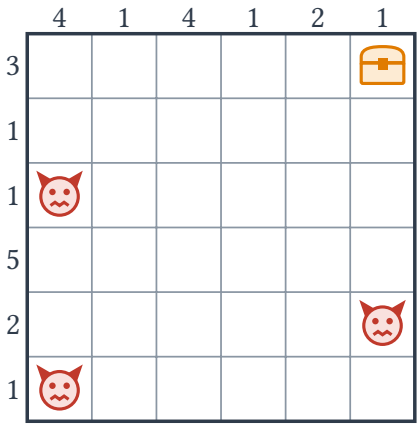


Figure 50.1 — Une grille 6x6 : monstres, trésor et nombres de murs attendus par ligne et par colonne.

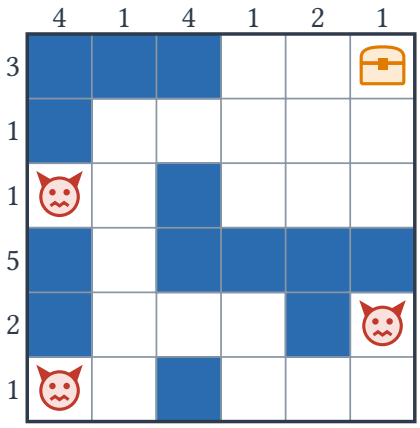


Figure 50.2 — Son unique solution (murs en bleu).

Question 1 Résoudre manuellement les 9 problèmes du PDF donné en lien au-dessus.

Corrigé

On peut vérifier ses solutions avec le programme écrit dans la suite de ce problème. Les solutions des 9 grilles de la section Exemples sont données à la fin de cette section.

Question 2 Que peut-on dire du graphe d'une solution pour un problème sans trésor ?

Corrigé

C'est un graphe connexe dont les sommets de degré 1 (les feuilles) sont exactement les cases contenant des monstres, tous les autres sommets étant de degré au moins 2, et qui ne contient aucun carré 2x2 de cases libres.

Ce n'est pas forcément un arbre : un couloir peut faire le tour d'un bloc de murs et former un cycle. C'est le cas pour la grille 3 de la liste des 64 grilles, dont la solution contient un couloir circulaire.

Si on considère une grille satisfaisant les contraintes de lignes et de colonnes, une fois une salle au trésor identifiée, il est possible de la remplir et de placer un monstre à son unique entrée.

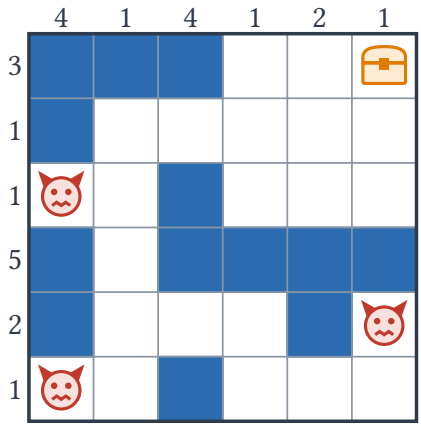


Figure 50.3 — Son unique solution (murs en bleu).

⇒

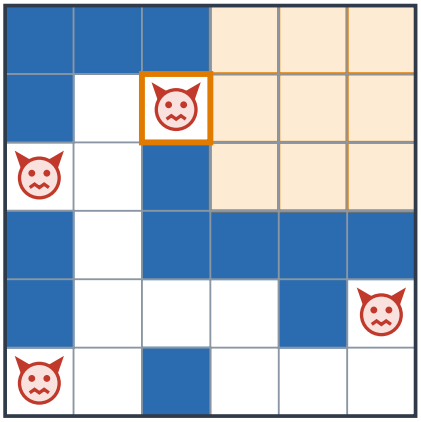


Figure 50.4 — La salle au trésor remplie de murs (en orange) et un monstre placé à son unique entrée (encadré).

On en déduit ainsi une méthode de vérification des solutions :

- tester qu'il y a le bon nombre de murs dans chaque ligne et chaque colonne
- vérifier que chaque trésor est dans une salle au trésor conforme
- remplir les salles au trésor et placer un monstre à leur entrée
- vérifier que le graphe est connexe, que ses feuilles sont exactement les cases contenant des monstres et qu'il ne contient pas de carré 2x2 de cases libres.

En pratique, il n'est pas nécessaire de vérifier la dernière condition avec un parcours de graphe, on peut se contenter de vérifier que le graphe est *localement* conforme :

- chaque case avec un monstre est de degré 1
- chaque case libre est de degré au moins 2
- il n'existe pas de carré de cases libres

Question 3 Pourquoi cette vérification n'est pas suffisante *a priori* ?

Bonus : trouver une grille pour laquelle cette approximation ne permet pas d'obtenir la solution.

Corrigé

Ces conditions sont locales : elles ne garantissent pas que le graphe est connexe. Plusieurs composantes connexes vérifiant chacune les conditions locales seraient acceptées.

Bonus : avec les indications de lignes 0, 8, 0, 8, 0, 8, 0, 8, des 4 sur toutes les colonnes et des monstres aux deux extrémités des lignes paires, les lignes sont entièrement forcées et on obtient :

```
M......M
########
M......M
########
M......M
########
M......M
########
```

Cette grille vérifie toutes les conditions locales (et le programme final l'affiche comme solution) alors que le graphe a quatre composantes connexes : la grille n'a en fait aucune solution.

Sur les 64 grilles de *Last Call BBS*, l'approximation donne cependant à chaque fois l'unique solution.

Compte tenu de la vérification de la présence d'une unique entrée dans une salle au trésor, on peut réordonner les vérifications :

- tester qu'il y a le bon nombre de murs dans chaque ligne et chaque colonne
- vérifier que les cases contenant des monstres sont de degré 1
- vérifier que les cases libres sont de degré au moins 2
- vérifier que chaque trésor est dans une salle au trésor conforme
- remplir les salles au trésor
- vérifier que la grille résultante ne contient pas de carré de cases libres

50.1.2 Représentation des plateaux en 64 bits

On va résoudre ce casse-tête par *backtracking* en C. Les grilles étant de taille 8x8, on peut ainsi représenter une information booléenne sur chaque case à l'aide d'entiers non signés sur 64 bits : chaque bit est associé à une case.

On choisit ici l'ordre suivant :

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
…							

Figure 50.5 — Numérotation des cases : la case i correspond au bit a_i.

qui correspond au mot binaire $a_{63}…a_0$ sur 64 bits avec a_i représentant l'information sur la case i.

On va associer trois entiers à une grille :

- le plateau (*board*) où un booléen vrai indique la présence d'un mur
- les monstres où un booléen vrai indique la présence d'un monstre
- les trésors où un booléen vrai indique la présence d'un trésor

Par exemple, pour la grille :

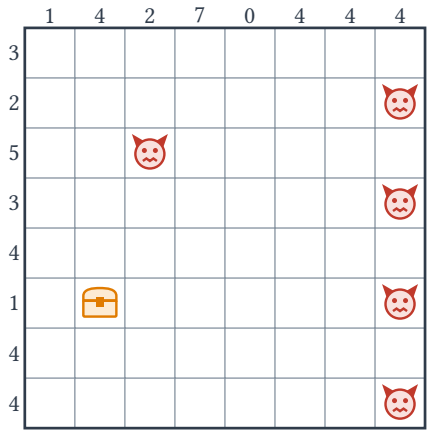


Figure 50.6 — Grille dont les monstres occupent les cases 15, 18, 31, 47 et 63.

on a les monstres dans les positions 15, 18, 31, 47 et 63. Cela correspond alors à l'entier s'écrivant en binaire

1000000000000000100000000000000010000000000001001000000000000000

et qui vaut 9223512776490909696 en décimal ainsi que `0x8000800080048000` en hexadécimal.

On pourra écrire en C :

```c
// au début du fichier
#include <stdint.h>
typedef uint64_t board;

// puis
board monsters = 0x8000800080048000;
```

On pourrait considérer une représentation *développée* sous la forme d'un tableau de 64 booléens. Dans ce cas-là, on obtiendrait la valeur suivante en C :

```c
// au début du fichier
#include <stdbool.h>
typedef bool map[64];

// puis

map monsters = {
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 1,
    0, 0, 1, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 1,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 1,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 1,
};
```

Question 4 Écrire une fonction de signature

```c
board map_to_board(map m);
```

qui renvoie la représentation entière associée à un tableau.

Corrigé

On construit l'entier en partant du bit de poids fort `m[63]` (schéma de Horner) :

```
board map_to_board(map m)
{
    board b = 0;
    // m[63] must be the most significant bit: iterate backwards
    for(int i = 63; i >= 0; i--)
    {
        b = b * 2;
        if (m[i])
            b = b + 1;
    }
    return b;
}
```

Question 5 Écrire une fonction de signature

```
void board_to_map(board b, map m);
```

qui prend en entrée une représentation entière `b`, un tableau `m` de 64 cases et place dans `m` le tableau de booléens déduit de `b`.

Corrigé

```
void board_to_map(board b, map m)
{
    for(int i = 0; i < 64; i++)
    {
        m[i] = b % 2 == 1;
        b = b / 2;
    }
}
```

Question 6 Faire des tests pour vérifier que ces deux fonctions sont correctes. Notamment, qu'elles sont réciproques l'une de l'autre.

Corrigé

```
#include <assert.h>

void test_map_board(void)
{
    map monsters = {
        0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 1,
        0, 0, 1, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 1,
        0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 1,
        0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 1,
    };
    assert(map_to_board(monsters) == 0x8000800080048000);

    map m;
    board b = 0x123456789abcdef1;
    for (int k = 0; k < 1000; k++)
    {
        board_to_map(b, m);
        assert(map_to_board(m) == b);
        // nombres pseudo-aléatoires (générateur congruentiel)
        b = b * 6364136223846793005 + 1442695040888963407;
    }
}
```

> Comme `board_to_map` est injective sur les tableaux de booléens et que `map_to_board(board_to_map(b)) = b` pour tout `b`, les deux fonctions sont bien réciproques.

50.1.3 Représentation du problème

Pour représenter le problème, on utilise le type suivant en C :

```
typedef char constraint[8];

struct problem {
    board monsters;
    board treasures;
    constraint rows;
    constraint columns;
};

typedef struct problem problem;
```

Pour définir un problème, on peut le faire directement comme ici :

```
problem p = {
    0x8000800080048000,
    0x20000000000,
    { 3,2,5,3,4,1,4,4 },
    { 1,4,2,7,0,4,4,4 }
};
```

Mais par lisibilité, on peut préciser les noms des champs comme ici :

```
problem p = {
    .monsters = 0x8000800080048000,
    .treasures = 0x20000000000,
    .rows = { 3,2,5,3,4,1,4,4 },
    .columns = { 1,4,2,7,0,4,4,4 }
};
```

Cette dernière syntaxe a l'avantage de permettre d'omettre n'importe quel champ, qui est alors initialisé à 0. On peut ainsi utiliser les fonctions précédemment définies pour présenter le problème de manière plus lisible dans le code source :

```
problem p = {
    .rows = { 3,2,5,3,4,1,4,4 },
    .columns = { 1,4,2,7,0,4,4,4 }
};

map monsters = {
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 1,
    0, 0, 1, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 1,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 1,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 1,
};

map treasures = {
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 1, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
```

```
    0, 0, 0, 0, 0, 0, 0, 0,
};
// ou plus rapidement
map treasure = { [41] = 1 }; // met 1 à l'indice 41 et 0 ailleurs

p.monsters = map_to_board(monsters);
p.treasures = map_to_board(treasures);
```

50.1.4 Choix pour le backtracking

On va effectuer des remplissages incrémentaux de la grille pour résoudre ce problème par backtracking. On reprend donc la terminologie de plateau et de coups permettant de passer d'un plateau à un autre.

Une majeure partie des conditions de résolution portant sur des grilles complètes, on pourrait se contenter de vérifier uniquement les grilles remplies. En faisant cela, on obtient un arbre déraisonnablement grand.

Par exemple, avec comme coup le fait, pour une position donnée, de choisir ou pas de placer un mur, on obtient un arbre binaire ayant 2^{64} feuilles.

On va plutôt considérer les coups consistant à placer directement k murs sur une ligne dont on sait qu'elle doit en contenir k.

Question 7 Justifier que si les indications de contraintes sur les lignes sont données par les nombres $l_1, \ldots, l_8$ alors on obtient ainsi un arbre de backtracking ayant

$$\prod_{k=1}^{8} \binom{8}{l_k}$$

feuilles.

Dans le cas où $l_1 = \ldots = l_8 = 4$, comparer ce nombre avec le nombre obtenu en effectuant des mouvements case par case.

✍ **Corrigé**

L'arbre a 8 niveaux : au niveau k, on choisit la ligne k. Un nœud de profondeur $k-1$ a exactement $\binom{8}{l_k}$ fils, un par choix des l_k cases de murs parmi les 8 cases de la ligne, indépendamment des lignes précédentes. Par récurrence sur k, il y a donc $\prod_{j=1}^{k}\binom{8}{l_j}$ nœuds de profondeur k, et les feuilles sont les nœuds de profondeur 8.

Pour $l_k = 4$, on obtient $\binom{8}{4}^8 = 70^8 \approx 5,8 \cdot 10^{14}$ feuilles contre $2^{64} \approx 1,8 \cdot 10^{19}$, soit environ 32 000 fois moins.

Même si ce nombre est beaucoup plus petit que le précédent, cela reste prohibitif pour faire une recherche exhaustive.

On va donc adopter une détection des nœuds pour lesquels on est certain de ne pas pouvoir résoudre le problème.

Tout d'abord, on restreint les lignes à celles qui ne placent pas un mur sur la même case qu'un monstre ou un trésor.

Ensuite, si on a un nœud correspondant à avoir rempli les k premières lignes, on vérifie :

- que le nombre de murs dans une colonne n'est pas strictement supérieur à la contrainte (**column overload**)
- qu'aucun monstre n'est sur une case de degré 0 (**trapped monsters**)
- qu'aucune case libre sur les k premières lignes n'est de degré ≤ 1 (**no deadend upto row**)

On a indiqué entre parenthèses les noms des fonctions qu'on programmera dans la suite pour réaliser ces tests.

50.2 Opérations bit à bit en C

Dans ce paragraphe, on notera `0b10101` une valeur en binaire en `C`. Cette notation, calquée sur la notation hexadécimale, n'est pas valide en C standard (avant C23), mais rendra les exemples plus lisibles. De même, dans les formules, on notera 10101_2 un nombre écrit en binaire.

On va étudier ici des opérateurs opérant directement sur l'écriture binaire des entiers. Aucun de ces opérateurs n'est explicitement au programme, cependant, leur usage permet, selon les contextes, d'avoir un code plus efficace et plus lisible.

50.2.1 Opérateurs de décalage

On utilise l'opération `x >> k` pour décaler l'écriture de `x` de k bits vers la droite, cela revient donc à réaliser une division entière par 2^k (pour un entier non signé).

De même, on utilise l'opération `x << k` pour décaler son écriture de k bits vers la gauche *en rajoutant des 0.*

Exemple :

- `123 >> 3` vaut 15 car `123 = 0b1111011` donc `123 >> 3 = 0b1111 = 15`
- `123 << 2` vaut 492 car `123 << 2 = 0b111101100 = 492`

Pour calculer 2^k on peut ainsi écrire `1 << k` mais cela pose un problème dès que $k \geq 31$ parce que `1` est un `int` (sur 32 bits en général) : le calcul provoque un dépassement de capacité, ce qui est un comportement indéfini en C (en pratique, on obtient un résultat faux). Pour $k \leq 63$, on peut régler ce problème en utilisant `1ULL << k` car `1ULL` est la constante `1` en `unsigned long long`, type d'au moins 64 bits (`1UL` convient sous Linux et macOS, où `unsigned long` fait 64 bits, mais pas sous Windows).

> Cependant, comme ce problème n'est pas propre aux constantes, on préfèrera utiliser une coercition comme ici : `(uint64_t)x << k` qui fonctionne quand `x` est un `int`.

On donne dans le code fourni une fonction

```
uint64_t bit(int pos)
{
    return (uint64_t) 1 << pos;
}
```

afin de pouvoir écrire directement `bit(n)` pour calculer 2^n avec $n \in [\![0, 63]\!]$.

50.2.2 Opérateurs logiques

Il est possible de réaliser les opérations logiques connues sur les booléens en les appliquant *bit à bit.* Pour cela, on utilise :

- `a & b` pour réaliser un *et* logique entre chaque bit. Ainsi, en écrivant abusivement en binaire, on a `0b10110 & 0b11010` qui vaut `0b10010`.
- `a | b` pour réaliser un *ou* logique entre chaque bit. Ainsi, on a `0b10110 | 0b11010` qui vaut `0b11110`.
- `a ^ b` pour réaliser un *ou exclusif* logique entre chaque bit. Ainsi, on a `0b10110 ^ 0b11010` qui vaut `0b01100`.
- `~a` pour réaliser la *négation* logique de chaque bit. Ainsi, sur 5 bits, on a `~0b10101` qui vaut `0b01010`.

Une manière classique d'utiliser `&` est de réaliser des masquages : on stocke dans une variable `mask` les bits que l'on veut étudier et on écrit `x & mask` pour ne conserver que ceux-ci.

L'opération `|` permet de réaliser efficacement une addition d'entiers dont les bits sont disjoints. En effet, comme il n'y a pas de retenue dans ce cas-là, il suffit de superposer les écritures avec un *ou.*

> **Remarque 50.1** Ces opérations utilisent de simples portes logiques et sont très rapides à exécuter sur un processeur.
>
> Notons que le compilateur les privilégie souvent. Par exemple, pour tester la parité, `x % 2 == 0` sera traduit par un test de nullité de `x & 1` et les divisions/multiplications par 2^k sont remplacées par des décalages.

50.2.3 Popcount

On va avoir besoin de compter les murs ou les cases libres, pour cela on utilise une opération appelée `popcount` (pour *population count*) qui permet, étant donné un entier non signé `x` sur 64 bits de renvoyer le nombre $p(x) \in [\![0, 64]\!]$ de 1 dans son écriture binaire.

Dans la suite, on utilisera cette fonction sur un plateau pour compter les murs ou sur sa négation pour compter les cases libres.

Calcul naïf

Pour réaliser un calcul naïf, on va juste itérer sur les bits en faisant des tests de parité et des divisions par 2.

Question 8 Écrire une fonction de signature

```
int popcount_slow(uint64_t x);
```

qui effectue ce calcul.

Corrigé

```
int popcount_slow(uint64_t v)
{
    int s = 0;
    while(v != 0)
    {
        if (v % 2 == 1)
            s = s+1;
        v = v/2;
    }
    return s;
}
```

On pourrait penser que ce code est long en raison de la boucle, mais en fait, celle-ci sera sûrement déroulée par le compilateur et le problème vient des branchements induits par les tests de parité.

Calcul par masquage et additions

Soit $p > 0$ et x, y deux entiers tels que $x, y \leq p < 2^p$. On a alors $x + y \leq 2p < 2^{p+1}$ ainsi, si $z = x2^p + y$, en considérant le masque $m = 2^p - 1$, on peut calculer $x + y$ ainsi :

```
(z & m) + ((z & (m << p)) >> p)
```

Cette remarque a priori anodine permet de réaliser un calcul efficace de `popcount`. En effet, on souhaite effectuer en place la somme de tous les bits d'un entier.

Si `x` est un entier sur 8 bits, on peut se servir du procédé de masquage pour réaliser plusieurs additions d'un coup à l'aide des masques

$$m_1 = 01010101_2 \quad m_2 = 00110011_2 \quad m_3 = 00001111_2$$

Par exemple, si $x = 173 = 10101101_2$, on va avoir :

z	m	p	zm	$(z(m << p)) >> p$	$zm + (z(m << p)) >> p$
10101101	01010101	1	00000101	01010100	01011001
01011001	00110011	2	00010001	00010010	00100011
00100011	00001111	4	00000011	00000010	00000101

Ici, les couleurs servent à identifier les sous-mots sur lesquels portent les additions.

Le dernier résultat est alors le nombre de 1 présents dans x : $00000101_2 = 5$.

On en déduit le programme C suivant :

```c
unsigned char popcount_byte(unsigned char x)
{
    unsigned char m1 = 0x55; // 0b01010101
    unsigned char m2 = 0x33; // 0b00110011
    unsigned char m3 = 0x0f; // 0b00001111

    x = (x&m1) + ((x&(m1<<1))>>1);
    x = (x&m2) + ((x&(m2<<2))>>2);
    x = (x&m3) + ((x&(m3<<4))>>4);

    return x;
}
```

Question 9 En déduire une fonction de signature

```c
int popcount(uint64_t x);
```

qui calcule le popcount d'un entier sur 64 bits en utilisant cette méthode. On peut se contenter de renvoyer un `int` car on sait que la valeur renvoyée est dans $[\![0, 64]\!]$.

Corrigé

Il faut $\log_2 64 = 6$ étapes, avec des blocs de taille 1, 2, 4, 8, 16 puis 32 bits :

```c
int popcount(uint64_t v)
{
    uint64_t mask_1 = 0x5555555555555555;
    uint64_t mask_2 = 0x3333333333333333;
    uint64_t mask_3 = 0x0f0f0f0f0f0f0f0f;
    uint64_t mask_4 = 0x00ff00ff00ff00ff;
    uint64_t mask_5 = 0x0000ffff0000ffff;
    uint64_t mask_6 = 0x00000000ffffffff;

    v = (v & mask_1) + ((v & (mask_1 << 1)) >> 1);
    v = (v & mask_2) + ((v & (mask_2 << 2)) >> 2);
    v = (v & mask_3) + ((v & (mask_3 << 4)) >> 4);
    v = (v & mask_4) + ((v & (mask_4 << 8)) >> 8);
    v = (v & mask_5) + ((v & (mask_5 << 16)) >> 16);
    v = (v & mask_6) + ((v & (mask_6 << 32)) >> 32);

    return v;
}
```

Question 10 Combien cette fonction fait-elle d'opérations élémentaires ?

✍

Corrigé

Les masques décalés `mask_i << k` sont des constantes calculées à la compilation. Chaque étape fait donc deux `&`, un décalage `>>` et une addition, soit 4 opérations, et la fonction fait $6 \times 4 = 24$ opérations élémentaires, sans aucun branchement, quel que soit `x`. La version naïve fait jusqu'à 64 tours de boucle, chacun avec un test.

Instruction assembleur

Il existe une fonction `__builtin_popcountll` (extension de `gcc` et `clang`) permettant de réaliser un popcount grâce à une instruction assembleur quand elle est disponible sur l'architecture visée, ou avec le code qu'on vient de voir quand ce n'est pas le cas.

Si votre processeur dispose de cette instruction (processeur x86-64), vous pouvez compiler avec le flag `-mpopcnt` pour l'utiliser. Cela permettra d'améliorer les performances.

50.3 Squelette du backtracking

La fonction récursive `solve` réalisant le backtracking a la structure suivante :

```
void solve(problem *p, int row, board b)
{
    // teste si b est une impasse et sort

    // b est une feuille de l'arbre
    if(row == 8)
    {
        // teste et sort si ce n'est pas une solution

        printf("Solution %lx\n", b);
        print_board(p, b);
        return;
    }

    // pour chaque mouvement move valide dans
    // la rangée row
        solve(p, row+1, b+move);
}
```

Question 11 Écrire une fonction de signature

```
void print_board(problem *p, board b);
```

qui permet d'afficher le plateau avec un affichage comme celui-ci :

```
###...##
#...#...
M.#.#...
#...#T..
###.####
.....#.M
M#M#.#.#
####...#
```

où `#` est un mur, `M` est un monstre, `T` un trésor et `.` est une case libre.

Corrigé

```
void print_board(problem *p, board b)
{
    for (int i = 0; i < 64; i++)
    {
        if (((b >> i) & 1) != 0)
            putchar('#');
        else if (((p->monsters >> i) & 1) != 0)
            putchar('M');
        else if (((p->treasures >> i) & 1) != 0)
            putchar('T');
        else putchar('.');

        if (i % 8 == 7) printf("\n");
    }
}
```

Afin de réaliser la boucle sur les mouvements, on peut itérer sur les 256 placements possibles de murs, ne garder que ceux qui ont le bon nombre de murs et qui ne se superposent pas aux monstres et aux trésors.

Question 12 Écrire le test permettant de s'assurer que `move` n'est pas en conflit avec les monstres et les trésors du problème `p`.

Corrigé

`move` est le placement de murs déjà décalé sur la ligne `row` :

```
// ensure there's no overlap with walls/monsters/treasures
if ((move & (p->monsters | p->treasures)) != 0)
    continue;
```

Question 13 Écrire la boucle en utilisant `popcount` pour tester qu'il y a le bon nombre de murs.

Corrigé

```
for(int i = 0; i < 256; i++)
{
    if (popcount(i) == p->rows[row])
    {
        uint64_t move = (uint64_t)i << (8*row);

        // ensure there's no overlap with walls/monsters/treasures
        if ((move & (p->monsters | p->treasures)) != 0)
            continue;

        solve(p, row+1, b + move);
    }
}
```

La fonction précédente va refaire continuellement les mêmes calculs de `popcount`.

Question 14 Dans le programme donné, il y a une variable globale

```
int enum_popcount_byte[256];
```

Rajouter les instructions dans `main` pour que `enum_popcount_byte[i]` contienne la valeur `popcount(i)` et en déduire une version plus efficace de la boucle précédente.

Corrigé

Dans `main`, avant les appels à `solve` :

```
for(int i = 0; i < 256; i++)
    enum_popcount_byte[i] = popcount(i);
```

puis, dans `solve`, on remplace le test `popcount(i) == p->rows[row]` par `enum_popcount_byte[i] == p->rows[row]`.

Remarque 50.2 On peut réduire les branchements en précalculant les tableaux de placements pour chaque nombre de murs (c'est ce que font les tableaux `enum_byte` et `enum_byte_idx` du corrigé complet).

50.4 Implémentation des différentes vérifications

Toutes les fonctions `test_...` ci-dessous renvoient `true` lorsqu'elles détectent un problème, c'est-à-dire lorsque le plateau doit être rejeté : c'est la convention utilisée par la fonction `solve` du fichier fourni (`if (test_...(p, b)) return;`).

50.4.1 Tests pour les colonnes

Il n'est pas nécessaire de tester si les lignes ont le bon nombre de murs, car on a restreint le backtracking pour l'assurer.

Question 15 Déterminer un masque correspondant à la première colonne.

En déduire, à l'aide de décalages, une fonction de signature

```
uint64_t mask_column(int i)
```

renvoyant un masque correspondant à la colonne `i`.

Corrigé

La première colonne est formée des cases $0, 8, 16, \ldots, 56$, soit un bit à 1 dans chaque octet : `0x0101010101010101`.

```
uint64_t mask_column(int i)
{
    return (uint64_t)0x0101010101010101 << i;
}
```

Question 16 En déduire des fonctions de signature

```
bool test_columns_overload(problem *p, board b);
bool test_columns_underload(problem *p, board b);
```

testant respectivement si une colonne a trop ou pas assez de murs dans le plateau.

Corrigé

```
bool test_columns_overload(problem *p, board b)
{
    for(int col = 0; col < 8; col++)
        if (popcount(b & mask_column(col)) > p->columns[col])
            return true;
    return false;
}

bool test_columns_underload(problem *p, board b)
{
    for(int col = 0; col < 8; col++)
        if (popcount(b & mask_column(col)) < p->columns[col])
            return true;
    return false;
}
```

50.4.2 Impasses

Question 17 Écrire une fonction de signature

```
bool empty(problem *p, board b, int pos);
```

indiquant s'il y a une case libre dans le plateau `b` à la position `pos`.

Attention, une case contenant un monstre ou un trésor n'est pas libre.

Corrigé

```
bool empty(problem *p, board b, int pos)
{
    board fb = b | p->monsters | p->treasures;

    return (fb & bit(pos)) == 0;
}
```

Question 18 Écrire des fonctions de signature

```
bool treasure(problem *p, int pos);
bool monster(problem *p, int pos);
```

déterminant respectivement s'il y a un trésor ou un monstre à la position `pos`.

Corrigé

```
bool treasure(problem *p, int pos)
{
    return p->treasures & bit(pos);
}

bool monster(problem *p, int pos)
{
    return p->monsters & bit(pos);
}
```

Question 19 Écrire une fonction de signature

```
uint64_t neighbours(int pos);
```

qui renvoie le masque correspondant aux cases voisines de la position `pos`. Ce masque contiendra ainsi de `2` à `4` bits à 1 selon les positions (`2` aux coins, `3` sur les bords stricts et `4` à l'intérieur du plateau).

Corrigé

```
uint64_t neighbours(int pos)
{
    uint64_t mask = 0;
    if (pos % 8 != 0)
        mask |= bit(pos - 1);
    if (pos % 8 != 7)
        mask |= bit(pos + 1);
    if (pos > 7)
        mask |= bit(pos - 8);
    if (pos < 56)
        mask |= bit(pos + 8);
    return mask;
}
```

Question 20 En déduire une fonction de signature

```
int count_non_wall(problem *p, board b, int pos);
```

qui compte le nombre de cases qui ne sont pas des murs dans le voisinage de `pos`.

On pensera bien à prendre en compte la négation du plateau pour ne pas avoir besoin de faire de cas selon qu'on soit à l'intérieur ou sur un bord.

Corrigé

```
int count_non_wall(problem *p, board b, int pos)
{
    uint64_t mask = neighbours(pos);
    return popcount(~b & mask);
}
```

Maintenant qu'on peut compter le degré d'une position dans le graphe, on peut réaliser les tests présentés plus haut.

Question 21 Écrire une fonction de signature

```
bool test_deadends(problem *p, board b);
```

qui renvoie `true` si, dans le plateau `b`, un monstre n'est pas dans une impasse ou une case libre est une impasse.

Corrigé

```
bool test_deadends(problem *p, board b)
{
    for(int i = 0; i < 64; i++)
    {
        // Empty cell must not be a dead-end
        if (empty(p,b,i) && count_non_wall(p, b, i) <= 1)
            return true;
        // Monsters must be a dead-end
        if (monster(p,i) && count_non_wall(p, b, i) != 1)
            return true;
    }

    return false;
}
```

Question 22 Écrire une fonction de signature

```
bool test_no_deadend_upto_row(problem *p, board b, int row);
```

qui renvoie `true` si une case libre est dans une impasse dans les rangées d'indice `< row`.

Corrigé

Les cases de la rangée `row` ne sont pas encore remplies et comptent comme libres : une case libre des rangées précédentes qui est déjà une impasse le restera.

```
bool test_no_deadend_upto_row(problem *p, board b, int row)
{
    for(int i = 0; i < 8*row; i++)
    {
        if (empty(p,b,i) && count_non_wall(p, b, i) <= 1)
            return true;
    }
    return false;
}
```

Question 23 Écrire une fonction de signature

```
bool test_trapped_monster(problem *p, board b);
```

qui renvoie `true` si un monstre est piégé par des murs.

Corrigé

```
bool test_trapped_monster(problem *p, board b)
{
    for(int i = 0; i < 64; i++)
        if (monster(p,i) && count_non_wall(p, b, i) == 0)
            return true;
    return false;
}
```

Carrés libres

Question 24 Déterminer un masque correspondant à un carré de quatre cases commençant à la position 0.

Corrigé

Les cases 0, 1, 8 et 9, soit `0x0303`.

Question 25 En déduire une fonction de signature

```
bool test_empty_squares(problem *p, board b);
```

qui renvoie `true` s'il y a un carré de quatre cases libres dans le plateau `b`.

Corrigé

On décale le masque sur les 7 premières colonnes de chacune des 7 premières lignes ; en fin de ligne, on décale une fois de plus pour passer de la colonne 6 à la colonne 0 de la ligne suivante. Le plateau `b` reçu contient aussi les monstres et les salles au trésor remplies.

```
bool test_empty_squares(problem *p, board b)
{
    uint64_t mask_square = 0x0303;
    for(int i = 0; i < 7; i++)
    {
        for(int j = 0; j < 7; j++)
        {
            if((b & mask_square) == 0)
                return true;
            mask_square *= 2;
        }
        mask_square *= 2;
    }

    return false;
}
```

Remarque 50.3 À partir d'ici, vous pouvez résoudre tous les problèmes ne comportant pas de trésors.

Salles au trésor

Pour chaque trésor, on va appliquer la méthode suivante :

- à l'aide d'un masque correspondant à un carré 3x3, identifier une possible salle au trésor autour du trésor (pas la peine de toutes les chercher, il ne peut y en avoir qu'une)
- à l'aide d'un masque correspondant à la bordure de la salle, en prenant garde au fait qu'elle n'est pas nécessairement à l'intérieur du plateau, déterminer l'unique ouverture de la salle
- remplir la salle au trésor de murs

Si jamais on n'arrive pas à trouver une salle valide, on sait que le plateau est insoluble.

Question 26 Écrire une fonction de signature

```
bool test_treasure_rooms(problem *p, board *b);
```

qui renvoie `true` si un trésor n'est pas dans une salle valide, et sinon remplit toutes les salles par des murs à l'aide du pointeur `b`.

Corrigé

Le plateau `*b` contient les murs et les monstres, ce qui exclut les salles contenant un monstre. Le carré 3x3 de coin supérieur gauche (s_x, s_y) correspond au masque `0x070707` décalé de $s_x + 8s_y$, à condition que $s_x, s_y \leq 5$.

```
bool test_treasure_rooms(problem *p, board *b)
{
    for(int i = 0; i < 64; i++)
    {
        if(treasure(p,i))
        {
            int x = i % 8;
            int y = i / 8;
            int minx = x - 2;
            if (minx < 0) minx = 0;
            int miny = y - 2;
            if (miny < 0) miny = 0;
            // the room must fit in the board: top-left corner (sx, sy)
            // with sx, sy <= 5, otherwise the shift wraps to the next row
            int maxx = x < 5 ? x : 5;
            int maxy = y < 5 ? y : 5;
            // starting point of the treasure room
            //    xxx
            //    xTx
            //    xxx
            bool found = false;
            uint64_t room = 0;
            int sx, sy;

            for(sy = miny; sy <= maxy; sy++)
            {
                for(sx = minx; sx <= maxx; sx++)
                {
                    room = (uint64_t)0x070707  << (sx + 8*sy);
                    if (popcount(~(*b) & room) == 9)
                    {
                        found = true;
                        break;
                    }
                }
                if(found) break;
            }

            if (!found) return true;
            uint64_t border = 0;
            if (sy > 0)
                border |= (uint64_t)0x07 << (sx+8*(sy-1));
            if (sy+2 < 7)
                border |= (uint64_t)0x07 << (sx+8*(sy+3));
            if (sx > 0)
                border |= (uint64_t)0x010101 << (sx-1+8*sy);
            if (sx+2 < 7)
                border |= (uint64_t)0x010101 << (sx+3+8*sy);

            int exit = popcount(~(*b) & border);
            if (exit != 1)
                return true;

            *b = *b | room;
        }
    }

    return false;
}
```

Remarque 50.4 Vous pouvez maintenant résoudre tous les problèmes. En tenant compte de toutes les optimisations dans les remarques et en compilant en `-O3`, il est possible de résoudre les 64 problèmes de *Last Call BBS* en moins de 500 ms !

50.5 Exemples

Les exemples de problèmes présents dans le fichier fourni sont reproduits ici dans le même ordre.

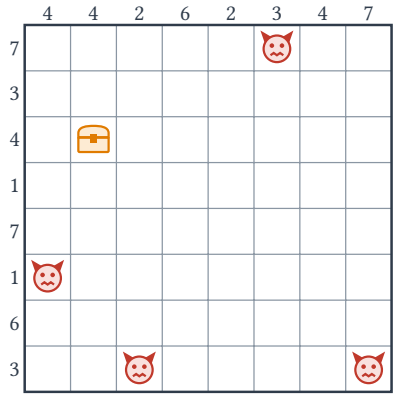


Figure 50.7 — Exemple 1 (problème p1 du fichier d_and_d.c).

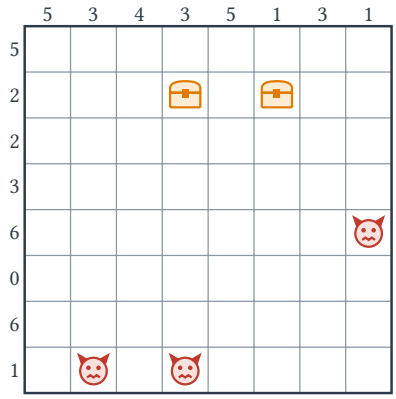


Figure 50.8 — Exemple 2 (problème p2 du fichier d_and_d.c).

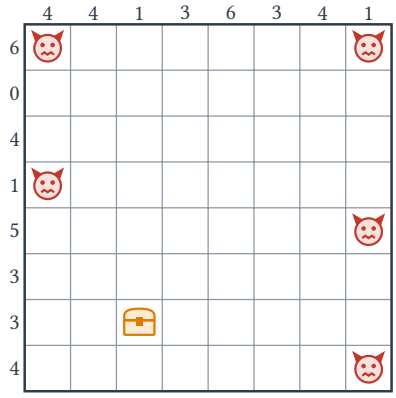


Figure 50.9 — Exemple 3 (problème p3 du fichier d_and_d.c).

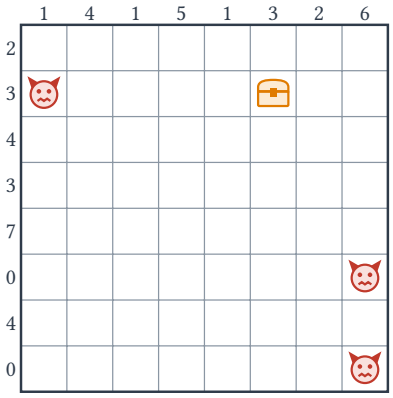


Figure 50.10 — Exemple 4 (problème p4 du fichier d_and_d.c).

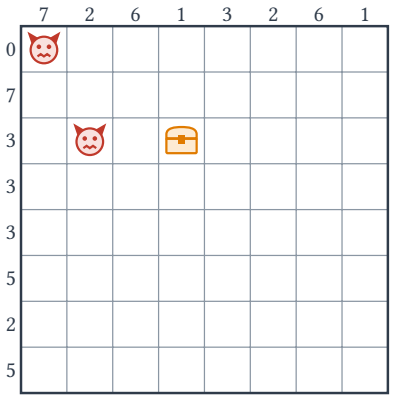


Figure 50.11 — Exemple 5 (problème p5 du fichier d_and_d.c).

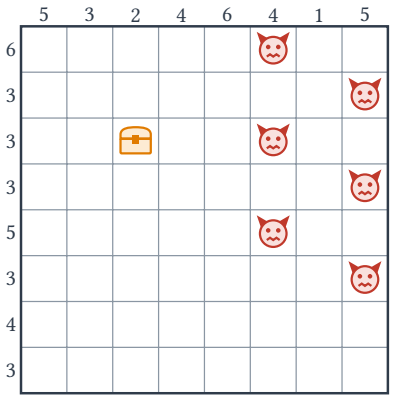


Figure 50.12 — Exemple 6 (problème p6 du fichier d_and_d.c).

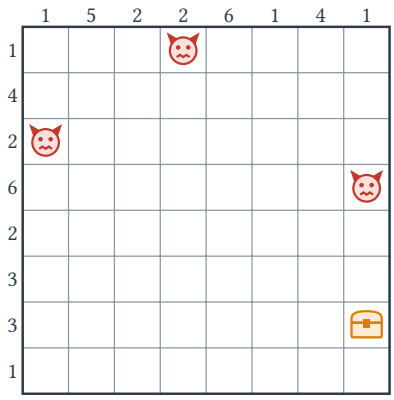


Figure 50.13 — Exemple 7 (problème p7 du fichier d_and_d.c).

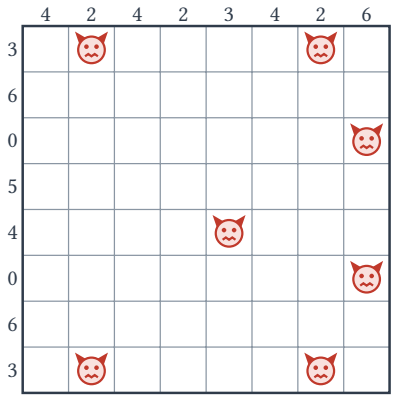


Figure 50.14 — Exemple 8 (problème p8 du fichier d_and_d.c).

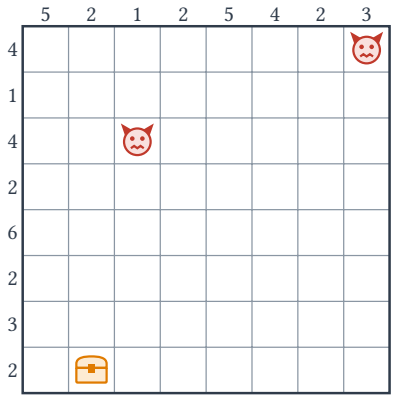


Figure 50.15 — Exemple 9 (problème p9 du fichier d_and_d.c).

Solutions des 9 exemples

Solutions obtenues avec le programme ci-dessous (`#` : mur, `M` : monstre, `T` : trésor), dans le même ordre :

```
Ex 1        Ex 2        Ex 3        Ex 4        Ex 5
######M##   #####...    M#######M   ...#...#    M.......
...#..##    #..T#T..    ........    M#.#.T.#    #######.
.T.#.###    #...#...    .#.##.#.    ##.#...#    #M#T..#.
.......#    #.....##    M#......    .....###    #.#...#.
######.#    #####.#M    ##.##.#M    .#######    #.#...#.
M......#    ........    #...#..#    .......M    #.#.###.
##.#.###    .######.    #.T.##..    .#.#.#.#    #...#...
##M#...M    .M#M....    #...###M    .......M    ###...##
```

```
Ex 6        Ex 7        Ex 8        Ex 9
#####M.#    ...M#...    #M#...M#    #...###M
#...##.M    .#.##.#.    #.#.####    ..#.....
#.T.#M.#    M#....#.    .......M    .#M#.#.#
#...##.M    ##.####M    .#.###.#    .....#.#
##.##M.#    ......##    .#.#M#.#    ##.###.#
...###.M    .##.#...    .......M    #...#...
.###...#    .##.#..T    #.#.####    #...#.#.
.....###    ....#...    #M#...M#    #T..#...
```

50.5.1 Solution

Corrigé complet

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <stdbool.h>
#include <inttypes.h>

typedef uint64_t board;
typedef bool map[64];
typedef char constraint[8];

struct problem {
    board monsters;
    board treasures;
    constraint rows;
    constraint columns;
};
typedef struct problem problem;

problem p1 = {
    .monsters = 0x8400010000000020,
    .treasures = 0x20000,
    .rows = { 7,3,4,1,7,1,6,3 },
    .columns = { 4,4,2,6,2,3,4,7 }
};

problem p2 = {
    .monsters = 0xa00008000000000,
    .treasures = 0x2800,
    .rows = { 5,2,2,3,6,0,6,1 },
    .columns = { 5,3,4,3,5,1,3,1 }
};

problem p3 = {
    .monsters = 0x8000008001000081,
    .treasures = 0x4000000000000,
    .rows = { 6,0,4,1,5,3,3,4 },
    .columns = { 4,4,1,3,6,3,4,1 }
};

problem p4 = {
    .monsters = 0x8000800000000100,
    .treasures = 0x2000,
    .rows = { 2,3,4,3,7,0,4,0 },
    .columns = { 1,4,1,5,1,3,2,6 }
};

problem p5 = {
    .monsters = 0x20001,
    .treasures = 0x80000,
    .rows = { 0,7,3,3,3,5,2,5 },
    .columns = { 7,2,6,1,3,2,6,1 }
};
```

```
problem p6 = {
    .monsters = 0x802080208020,
    .treasures = 0x40000,
    .rows = { 6,3,3,3,5,3,4,3 },
    .columns = { 5,3,2,4,6,4,1,5 }
};

problem p7 = {
    .monsters = 0x80010008,
    .treasures = 0x80000000000000,
    .rows = { 1,4,2,6,2,3,3,1 },
    .columns = { 1,5,2,2,6,1,4,1 }
};

problem p8 = {
    .monsters = 0x4200801000800042,
    .treasures = 0x0,
    .rows = { 3,6,0,5,4,0,6,3 },
    .columns = { 4,2,4,2,3,4,2,6 }
};

problem p9 = {
    .monsters = 0x40080,
    .treasures = 0x2000000000000000,
    .rows = { 4,1,4,2,6,2,3,2 },
    .columns = { 5,2,1,2,5,4,2,3 }
};

uint64_t mask_column(int i)
{
    return (uint64_t)0x0101010101010101 << i;
}

board map_to_board(map m)
{
    board b = 0;
    // m[63] must be the most significant bit: iterate backwards
    for(int i = 63; i >= 0; i--)
    {
        b = b * 2;
        if (m[i])
            b = b + 1;
    }
    return b;
}

void board_to_map(board b, map m)
{
    for(int i = 0; i < 64; i++)
    {
        m[i] = b % 2 == 1;
        b = b / 2;
    }
}


int popcount_slow(uint64_t v)
{
    int s = 0;
    while(v != 0)
    {
        if (v % 2 == 1)
            s = s+1;
        v = v/2;
    }
    return s;
```

```
}

int popcount(uint64_t v)
{
    uint64_t mask_1 = 0x5555555555555555;
    uint64_t mask_2 = 0x3333333333333333;
    uint64_t mask_3 = 0x0f0f0f0f0f0f0f0f;
    uint64_t mask_4 = 0x00ff00ff00ff00ff;
    uint64_t mask_5 = 0x0000ffff0000ffff;
    uint64_t mask_6 = 0x00000000ffffffff;

    v = (v & mask_1) + ((v & (mask_1 << 1)) >> 1);
    v = (v & mask_2) + ((v & (mask_2 << 2)) >> 2);
    v = (v & mask_3) + ((v & (mask_3 << 4)) >> 4);
    v = (v & mask_4) + ((v & (mask_4 << 8)) >> 8);
    v = (v & mask_5) + ((v & (mask_5 << 16)) >> 16);
    v = (v & mask_6) + ((v & (mask_6 << 32)) >> 32);

    return v;
}

#define popcount __builtin_popcountll

int enum_popcount_byte[256];

void print_board(problem *p, board b)
{
    for (int i = 0; i < 64; i++)
    {
        if (((b >> i) & 1) != 0)
            putchar('#');
        else if (((p->monsters >> i) & 1) != 0)
            putchar('M');
        else if (((p->treasures >> i) & 1) != 0)
            putchar('T');
        else putchar('.');

        if (i % 8 == 7) printf("\n");
    }
}

uint64_t bit(int pos)
{
    return (uint64_t) 1 << pos;
}

uint64_t neighbours(int pos)
{
    uint64_t mask = 0;
    if (pos % 8 != 0)
        mask |= bit(pos - 1);
    if (pos % 8 != 7)
        mask |= bit(pos + 1);
    if (pos > 7)
        mask |= bit(pos - 8);
    if (pos < 56)
        mask |= bit(pos + 8);
    return mask;
}

int count_non_wall(problem *p, board b, int pos)
{
    uint64_t mask = neighbours(pos);
    return popcount(~b & mask);
}

bool empty(problem *p, board b, int pos)
```

```
{
    board fb = b | p->monsters | p->treasures;

    return (fb & bit(pos)) == 0;
}

bool treasure(problem *p, int pos)
{
    return p->treasures & bit(pos);
}

bool monster(problem *p, int pos)
{
    return p->monsters & bit(pos);
}

// Optional: precompute wall test
int enum_byte[] = { 0, 1, 2, 4, 8, 16, 32, 64, 128, 3, 5, 6, 9, 10, 12, 17, 18, 20, 24, 33,
34, 36,
    40, 48, 65, 66, 68, 72, 80, 96, 129, 130, 132, 136, 144, 160,
    192, 7, 11, 13, 14, 19, 21, 22, 25, 26, 28, 35, 37, 38, 41, 42,
    44, 49, 50, 52, 56, 67, 69, 70, 73, 74, 76, 81, 82, 84, 88, 97,
    98, 100, 104, 112, 131, 133, 134, 137, 138, 140, 145, 146, 148,
    152, 161, 162, 164, 168, 176, 193, 194, 196, 200, 208, 224, 15,
    23, 27, 29, 30, 39, 43, 45, 46, 51, 53, 54, 57, 58, 60, 71, 75,
    77, 78, 83, 85, 86, 89, 90, 92, 99, 101, 102, 105, 106, 108, 113,
    114, 116, 120, 135, 139, 141, 142, 147, 149, 150, 153, 154, 156,
    163, 165, 166, 169, 170, 172, 177, 178, 180, 184, 195, 197, 198,
    201, 202, 204, 209, 210, 212, 216, 225, 226, 228, 232, 240, 31,
    47, 55, 59, 61, 62, 79, 87, 91, 93, 94, 103, 107, 109, 110, 115,
    117, 118, 121, 122, 124, 143, 151, 155, 157, 158, 167, 171, 173,
    174, 179, 181, 182, 185, 186, 188, 199, 203, 205, 206, 211, 213,
    214, 217, 218, 220, 227, 229, 230, 233, 234, 236, 241, 242, 244,
    248, 63, 95, 111, 119, 123, 125, 126, 159, 175, 183, 187, 189,
    190, 207, 215, 219, 221, 222, 231, 235, 237, 238, 243, 245, 246,
    249, 250, 252, 127, 191, 223, 239, 247, 251, 253, 254, 255 };
int enum_byte_idx[] = { 0, 1, 9, 37, 93, 163, 219, 247, 255, 256 };

bool test_columns_overload(problem *p, board b)
{
    for(int col = 0; col < 8; col++)
        if (popcount(b & mask_column(col)) > p->columns[col])
            return true;
    return false;
}

bool test_columns_underload(problem *p, board b)
{
    for(int col = 0; col < 8; col++)
        if (popcount(b & mask_column(col)) < p->columns[col])
            return true;
    return false;
}

bool test_no_deadend_upto_row(problem *p, board b, int row)
{
    for(int i = 0; i < 8*row; i++)
    {
        if (empty(p,b,i) && count_non_wall(p, b, i) <= 1)
            return true;
    }
    return false;
}

bool test_trapped_monster(problem *p, board b)
{
    for(int i = 0; i < 64; i++)
```

```
        if (monster(p,i) && count_non_wall(p, b, i) == 0)
            return true;
    return false;
}

bool test_deadends(problem *p, board b)
{
    for(int i = 0; i < 64; i++)
    {
        // Empty cell must not be a dead-end
        if (empty(p,b,i) && count_non_wall(p, b, i) <= 1)
            return true;
        // Monsters must be a dead-end
        if (monster(p,i) && count_non_wall(p, b, i) != 1)
            return true;
    }

    return false;
}

bool test_treasure_rooms(problem *p, board *b)
{
    for(int i = 0; i < 64; i++)
    {
        if(treasure(p,i))
        {
            int x = i % 8;
            int y = i / 8;
            int minx = x - 2;
            if (minx < 0) minx = 0;
            int miny = y - 2;
            if (miny < 0) miny = 0;
            // the room must fit in the board: top-left corner (sx, sy)
            // with sx, sy <= 5, otherwise the shift wraps to the next row
            int maxx = x < 5 ? x : 5;
            int maxy = y < 5 ? y : 5;
            // starting point of the treasure room
            //    xxx
            //    xTx
            //    xxx
            bool found = false;
            uint64_t room = 0;
            int sx, sy;

            for(sy = miny; sy <= maxy; sy++)
            {
                for(sx = minx; sx <= maxx; sx++)
                {
                    room = (uint64_t)0x070707  << (sx + 8*sy);
                    if (popcount(~(*b) & room) == 9)
                    {
                        found = true;
                        break;
                    }
                }
                if(found) break;
            }

            if (!found) return true;
            uint64_t border = 0;
            if (sy > 0)
                border |= (uint64_t)0x07 << (sx+8*(sy-1));
            if (sy+2 < 7)
                border |= (uint64_t)0x07 << (sx+8*(sy+3));
            if (sx > 0)
                border |= (uint64_t)0x010101 << (sx-1+8*sy);
            if (sx+2 < 7)
```

```
                border |= (uint64_t)0x010101 << (sx+3+8*sy);

            int exit = popcount(~(*b) & border);
            if (exit != 1)
                return true;

            *b = *b | room;
        }
    }

    return false;
}

bool test_empty_squares(problem *p, board b)
{
    uint64_t mask_square = 0x0303;
    for(int i = 0; i < 7; i++)
    {
        for(int j = 0; j < 7; j++)
        {
            if((b & mask_square) == 0)
                return true;
            mask_square *= 2;
        }
        mask_square *= 2;
    }

    return false;
}

void solve(problem *p, int row, board b)
{
    // Unsolvable from this board
    if (test_columns_overload(p, b))
        return;
    if (test_no_deadend_upto_row(p, b, row))
        return;
    if (test_trapped_monster(p, b))
        return;

    // Board is filled
    if(row == 8)
    {
        // Validate columns count
        if (test_columns_underload(p, b))
            return;

        // Monsters are exactly the dead-ends
        if (test_deadends(p, b))
            return;

        // Full board with monsters
        board fb = b | p->monsters;

        if (test_treasure_rooms(p, &fb))
            return;
        // Every treasure room is now filled in fb

        if (test_empty_squares(p, fb))
            return;

        printf("Solution %" PRIx64 "\n", b);
        print_board(p, b);

        return;
    }
```

```
    /*
    // Slower because of branching
    for(int i = 0; i < 256; i++)
        if (enum_popcount_byte[i] == p->rows[row])
        {
        uint64_t move = (uint64_t)i << (8*row);

        // ensure there's no overlap with walls/monsters/treasures
        if ((move & (p->monsters | p->treasures)) != 0)
            continue;

        solve(p, row+1, b + move);

        }
    */

    int tgt = p->rows[row];
    for(int v = enum_byte_idx[tgt]; v < enum_byte_idx[tgt+1]; v++)
    {
        uint64_t move = (uint64_t)enum_byte[v] << (8*row);

        // ensure there's no overlap with walls/monsters/treasures
        if ((move & (p->monsters | p->treasures)) != 0)
            continue;

        solve(p, row+1, b + move);
    }
}

int main(void)
{
    /*
    map monsters = {
        0, 0, 1, 0, 1, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 1,
        1, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 1,
        1, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 1, 0, 1, 0, 0,
    };
    */

    map monsters = { [5] = 1 };
    problem test = {
       .rows = { 6, 2, 5, 3, 2, 5, 2, 6 },
       .columns = { 6, 2, 4, 3, 4, 4, 2, 6 }
    };
    test.monsters = map_to_board(monsters);

    for(int i = 0; i < 256; i++)
        enum_popcount_byte[i] = popcount(i);

    problem *problems[] = {
        &p1, &p2, &p3,
        &p4, &p5, &p6,
        &p7, &p8, &p9
    };

    for (int i = 0; i < 9; i++)
    {
        printf("Solving problem %d\n", i);
        print_board(problems[i], 0);
        solve(problems[i], 0, 0);
    }
}
```

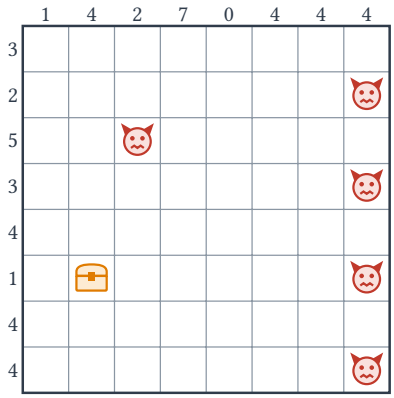
PROBLÈME 51

Dungeons and Diagrams - les 64 grilles

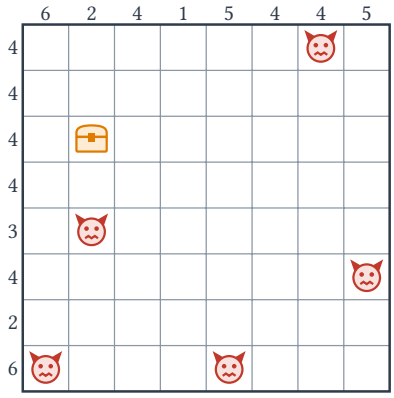
Voici les 64 grilles de *Dungeons and Diagrams* du jeu *Last Call BBS* (Zachtronics), dans l'ordre du fichier d_and_d.h où elles sont nommées `p00` à `p63`. Les règles et la résolution par backtracking sont présentées dans le problème Dungeons and Diagrams.

Remarque 51.1 Le fichier `d_and_d.h` contient aussi les définitions des types `board`, `map`, `constraint` et `problem` : pour l'utiliser avec le fichier d_and_d.c, il faut retirer ces définitions et les problèmes `p1` à `p9` de `d_and_d.c`, puis écrire `#include "d_and_d.h"` après les `#include` standards (`d_and_d.h` utilise `bool`, il faut donc inclure `stdbool.h` avant) et adapter le tableau `problems` de `main` (`&p00, ..., &p63`).

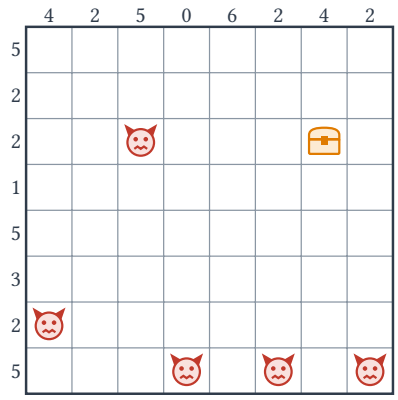
Grille 0



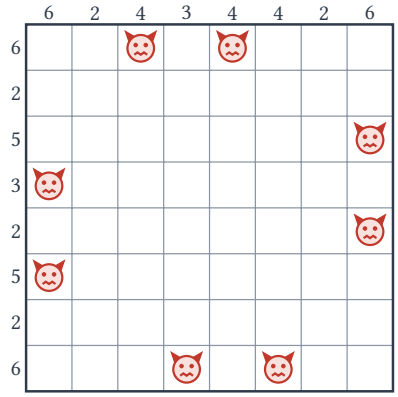
Grille 1



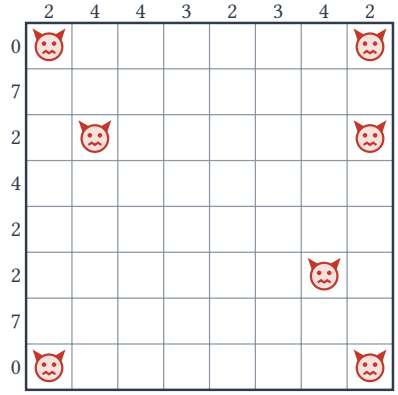
Grille 2



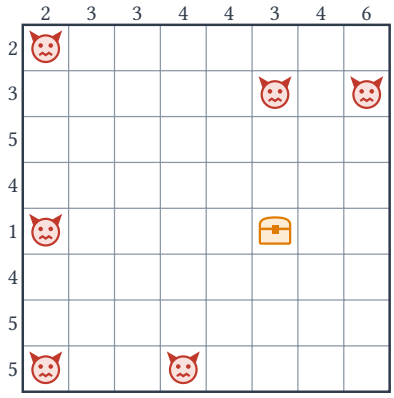
Grille 3



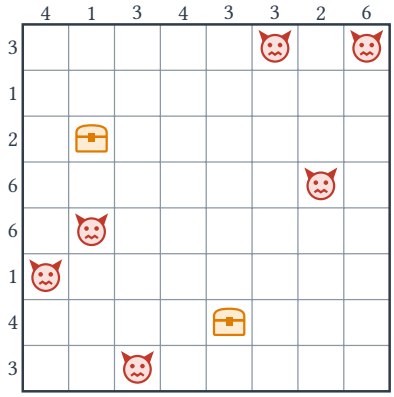
Grille 4



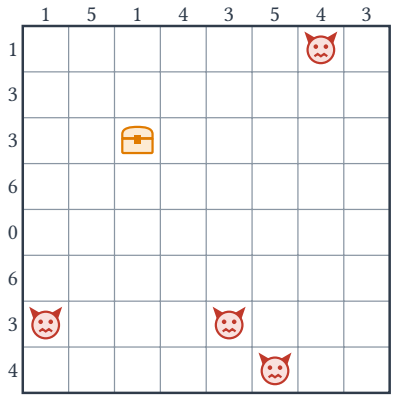
Grille 5



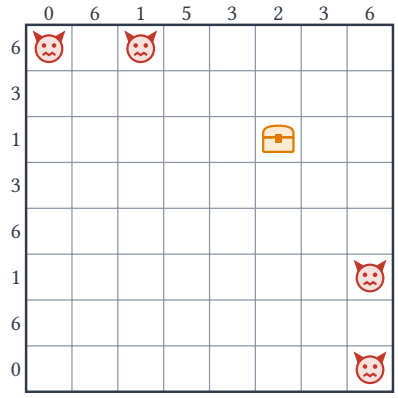
Grille 6



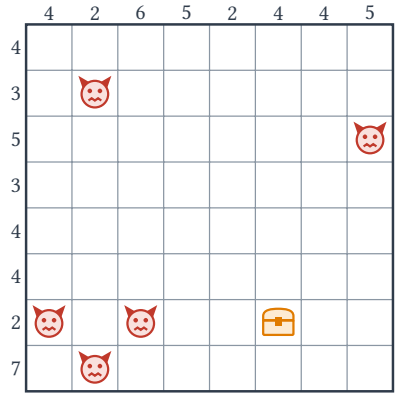
Grille 7



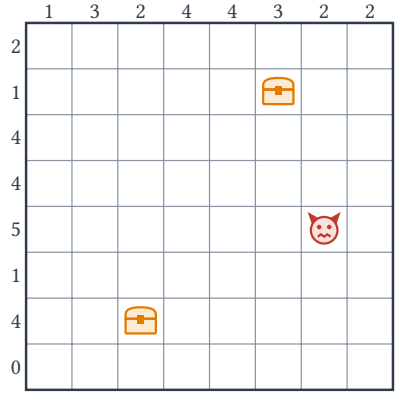
Grille 8



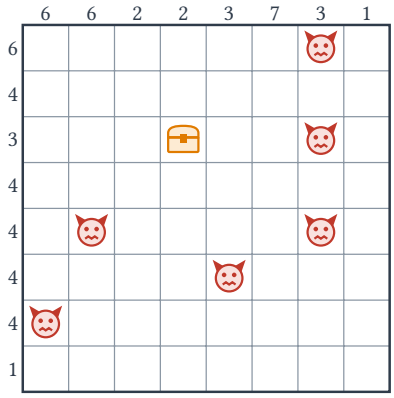
Grille 9



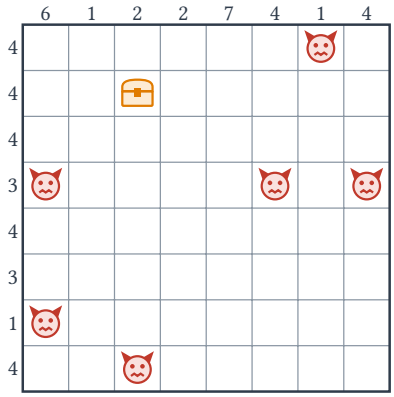
Grille 10



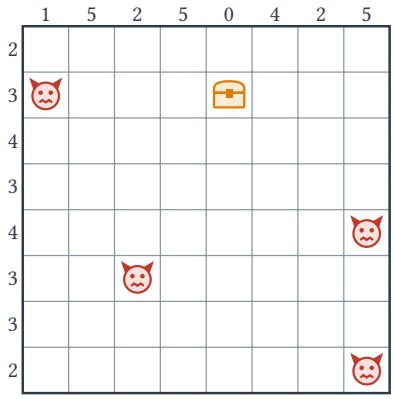
Grille 11



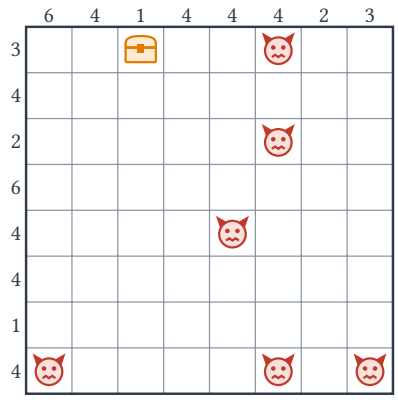
Grille 12



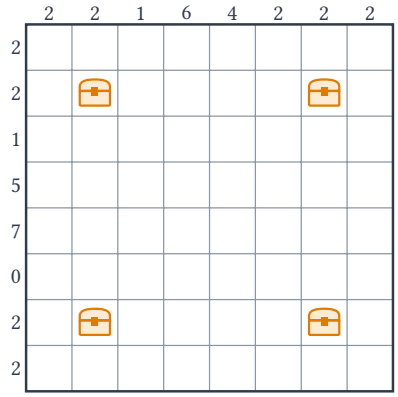
Grille 13



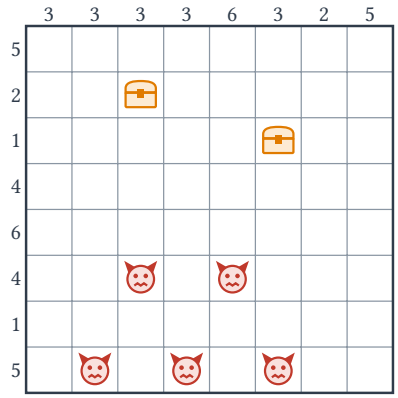
Grille 14



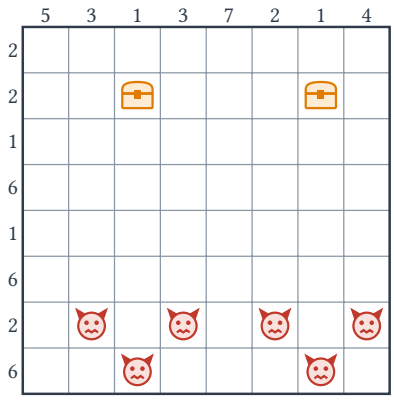
Grille 15



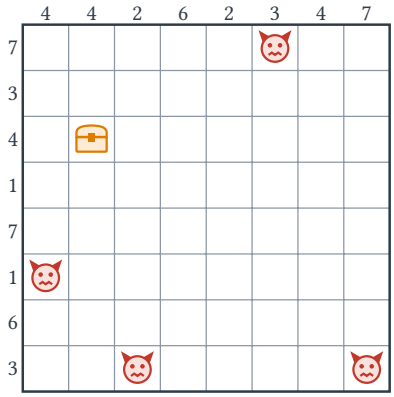
Grille 16



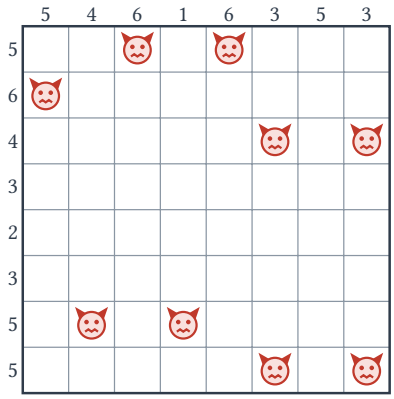
Grille 17



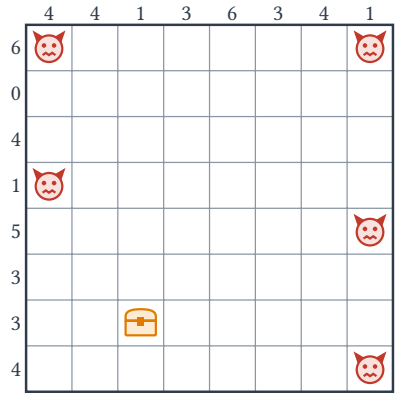
Grille 18



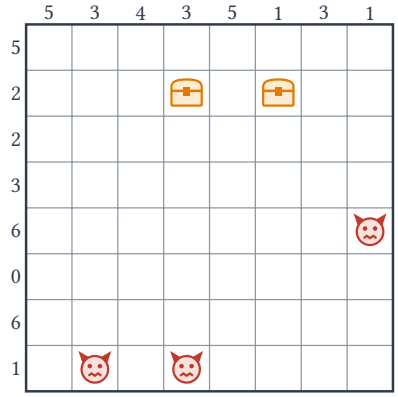
Grille 19



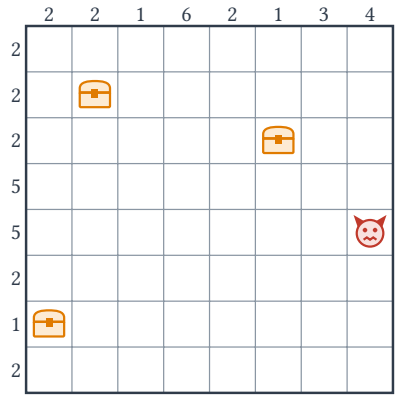
Grille 20



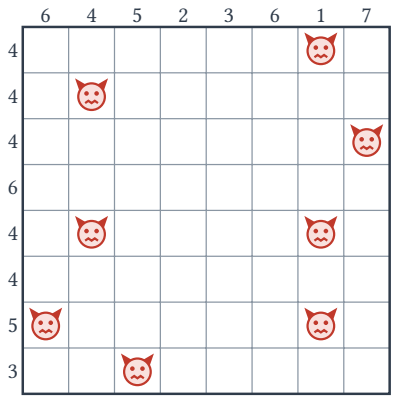
Grille 21



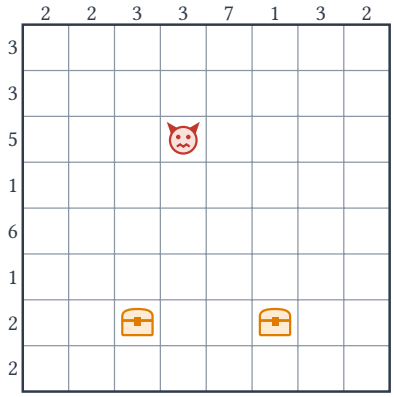
Grille 22



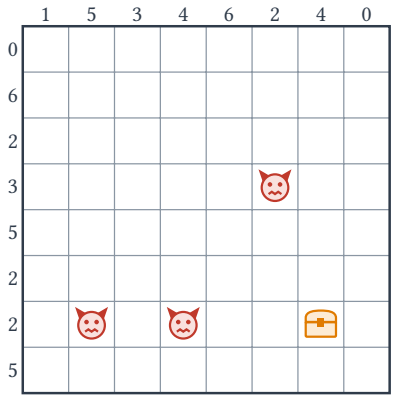
Grille 23



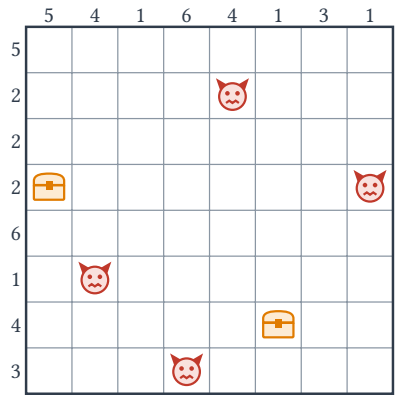
Grille 24



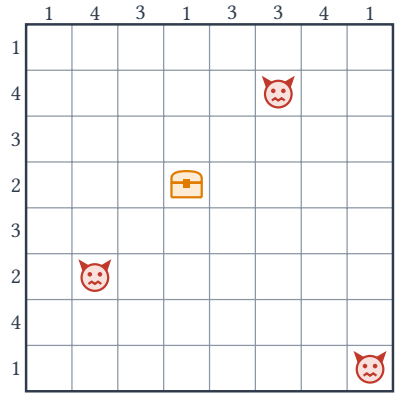
Grille 25



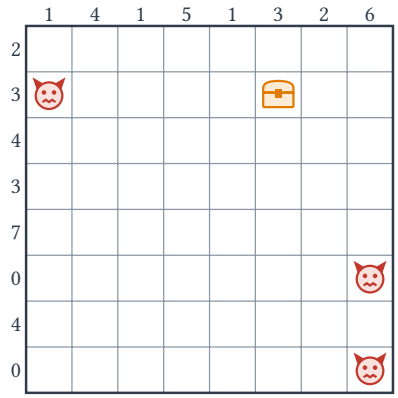
Grille 26



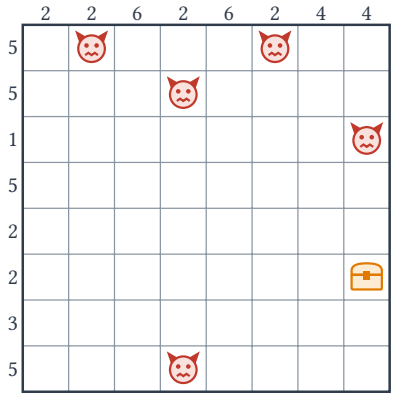
Grille 27



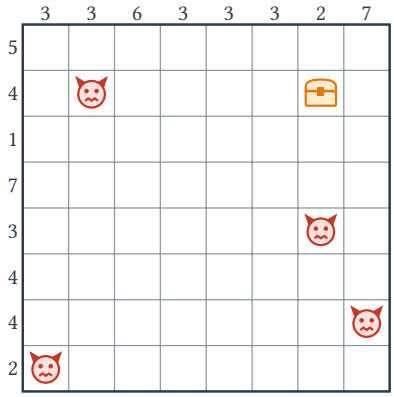
Grille 28



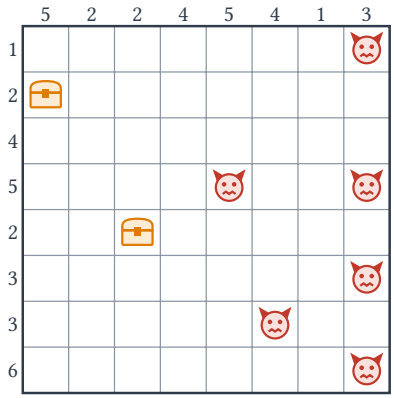
Grille 29



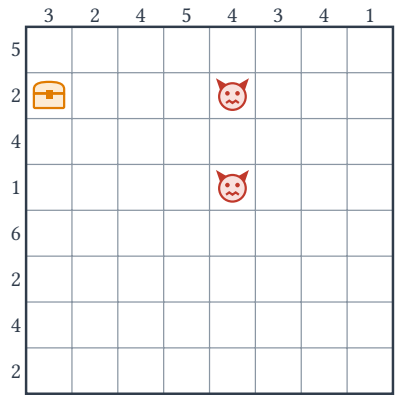
Grille 30



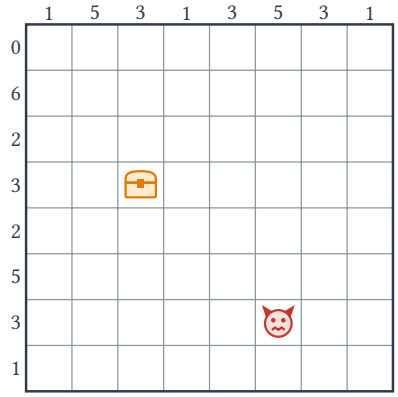
Grille 31



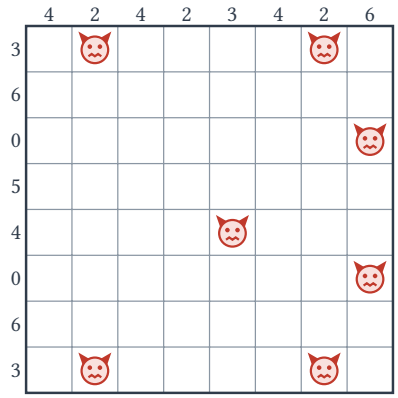
Grille 32



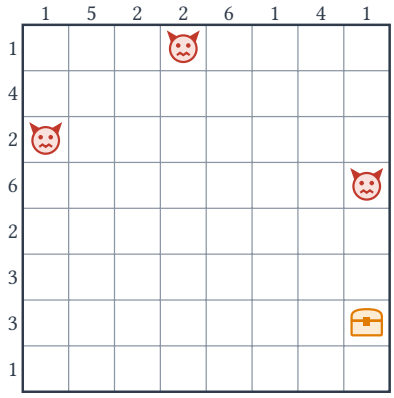
Grille 33



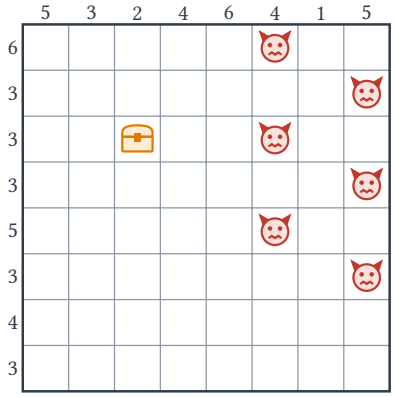
Grille 34



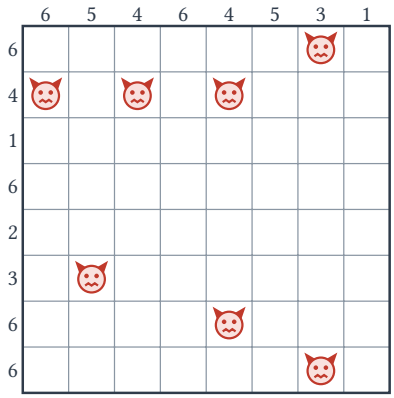
Grille 35



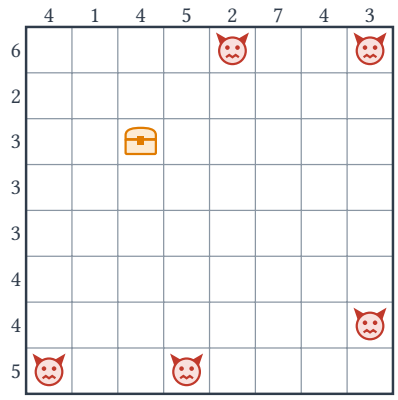
Grille 36



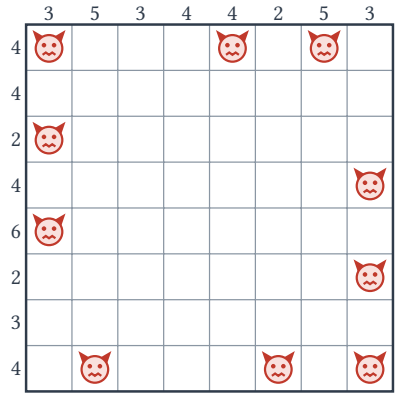
Grille 37



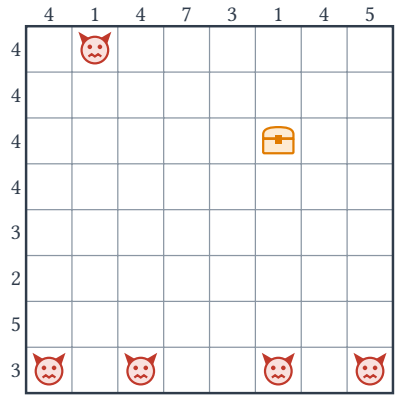
Grille 38



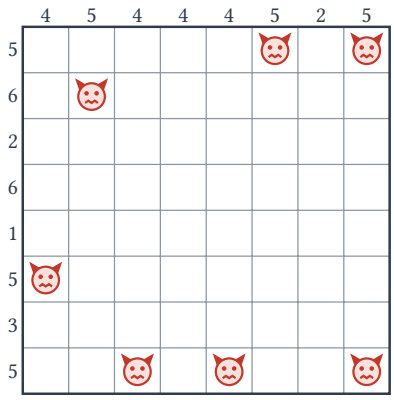
Grille 39



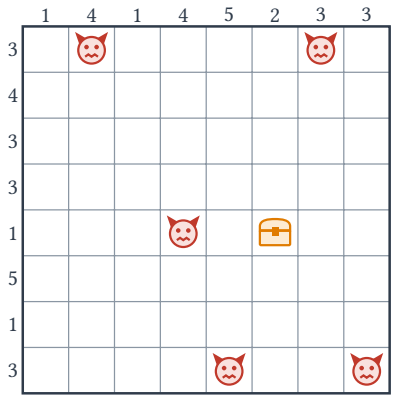
Grille 40



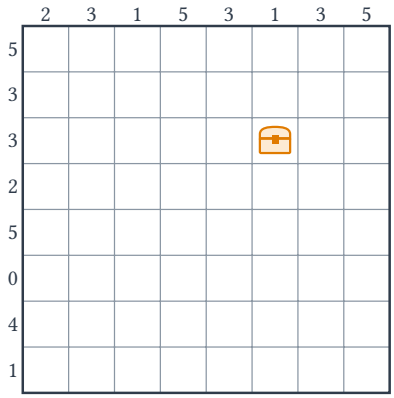
Grille 41



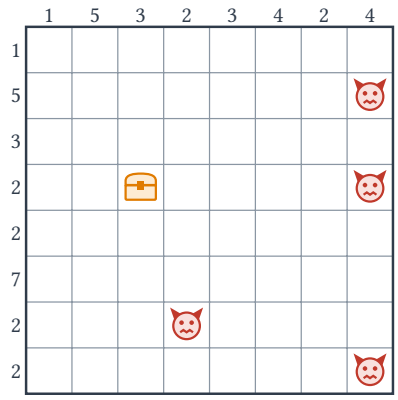
Grille 42



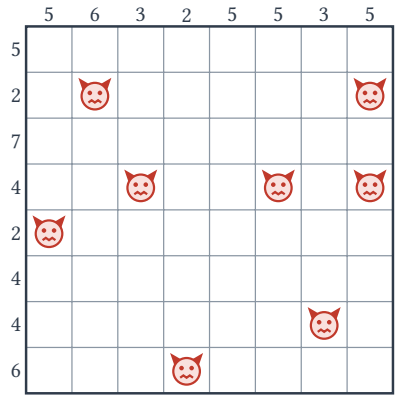
Grille 43



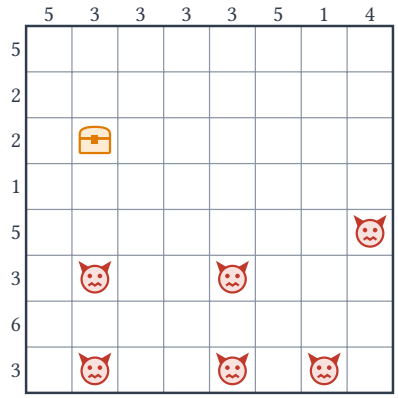
Grille 44



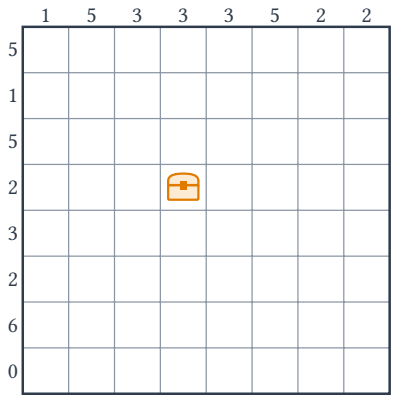
Grille 45



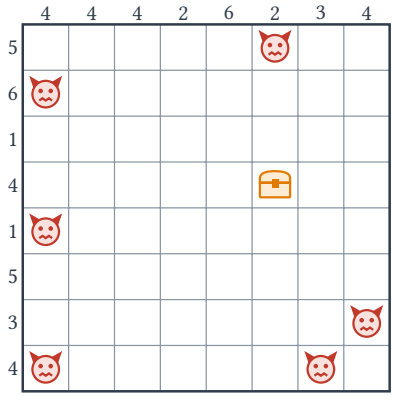
Grille 46



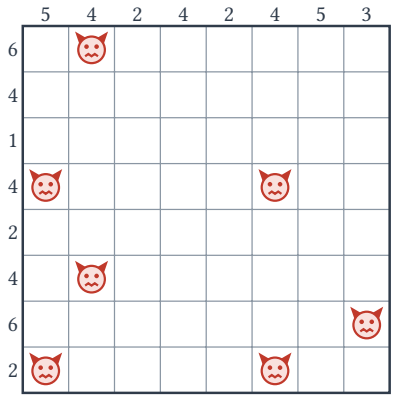
Grille 47



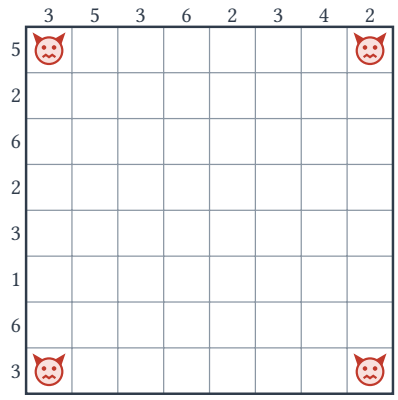
Grille 48



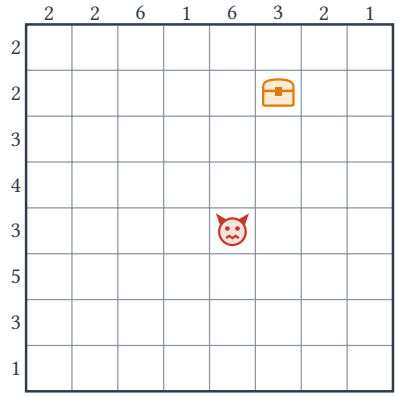
Grille 49



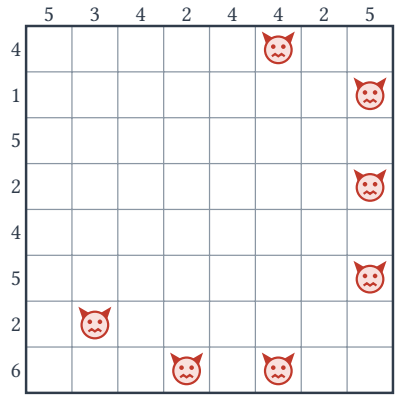
Grille 50



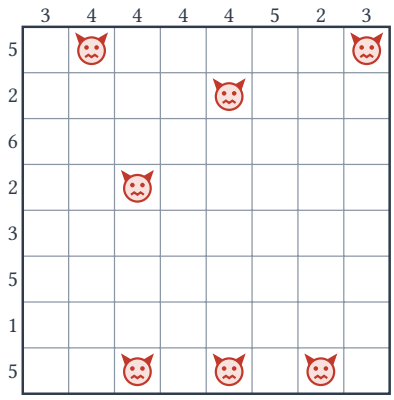
Grille 51



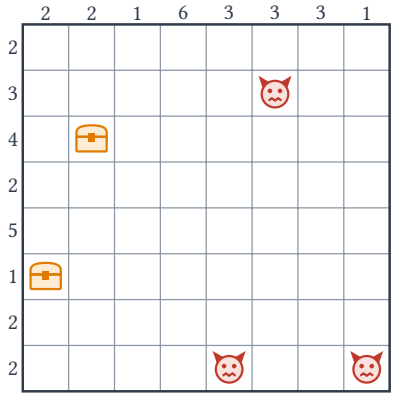
Grille 52



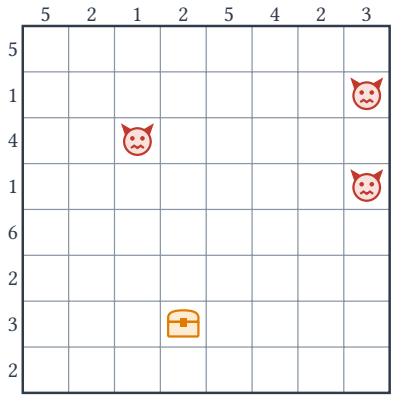
Grille 53



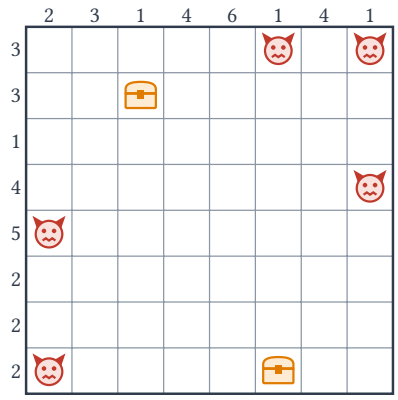
Grille 54



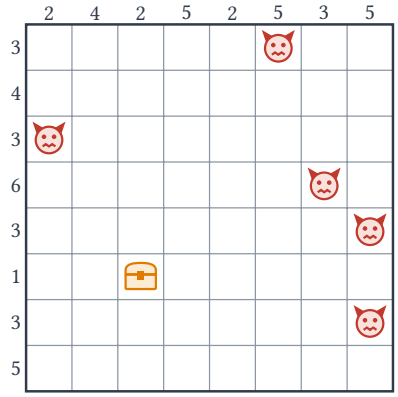
Grille 55



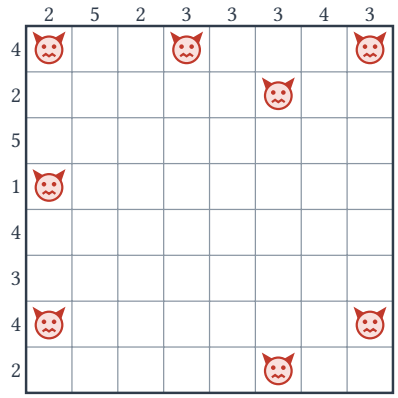
Grille 56



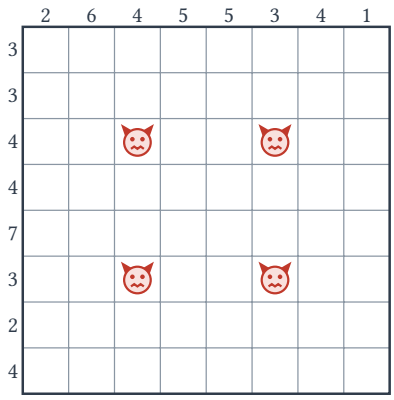
Grille 57



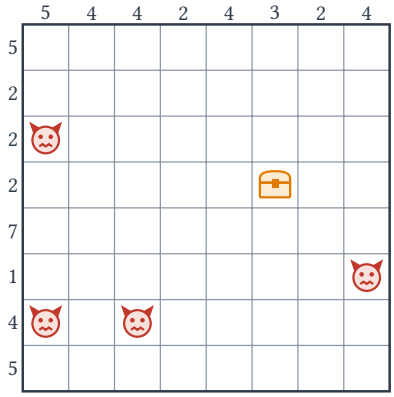
Grille 58



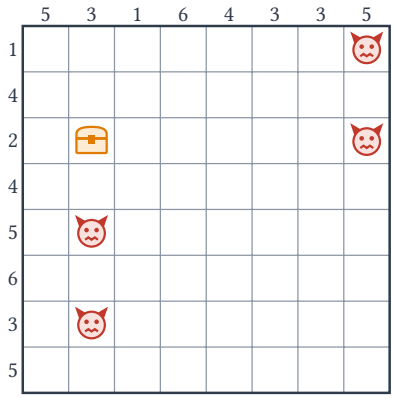
Grille 59



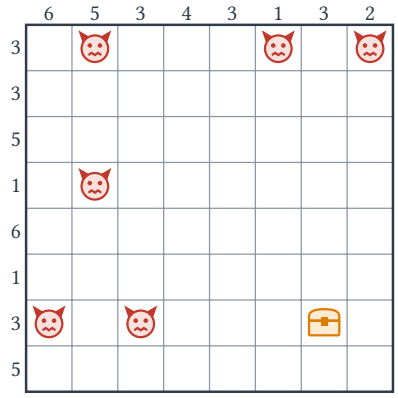
Grille 60



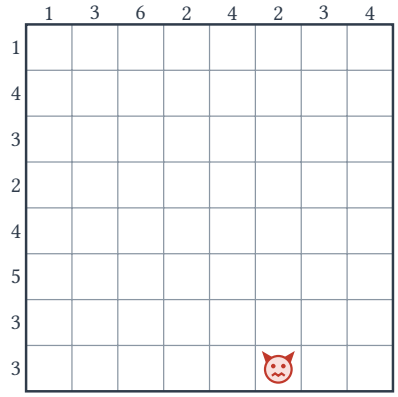
Grille 61



Grille 62



Grille 63



51.1 Solutions

Solutions des 64 grilles

Solutions obtenues avec le programme du problème (`#` : mur, `M` : monstre, `T` : trésor) ; chaque grille a une unique solution.

```
G 0          G 1          G 2          G 3          G 4
.....###     #...###M#    #...####     ###M#M###    M......M
.#.#...M     #...##.#     ..#.#...     #......#     ###.####
.#M#.###     #T..##.#     .#M.#.T.     #.####.M     .M#...#M
.###...M     ###.#...     ..#.....     M...##.#     .####...
...#.###     #M#...#.     #...####     #.#....M     .#....#.
.T.#...M     #.#.#.#M     #.#.#...     M.####.#     ...#.#M.
...#.###     ......##     M.#...#.     #......#     ####.###
####...M     M#####M##    ###M#M#M     ####M#M##    M......M

G 5          G 6          G 7          G 8          G 9
M##.....     ...###M#M    .....#M.     M#M#####     ####....
.#..#M#M     ...#....     .#...##.     .#.#...#     .M#..##.
...#####     .T...#.#     .#T..##.     .....T.#     .##.###M
#.##...#     ###.###M#    .#######.     .#.#...#     .....###
M....T.#     #M#.####     ........     .#.#####     #.##...#
#.##...#     M......#     ##.###.#     .#.....M     #.##...#
...#####     #.##T..#     M#.#M..#     .####.##     M.M#.T.#
M#.M####     #.M#...#     ...##M##     .......M     #M######

G 10         G 11         G 12         G 13         G 14
...##...     ######M.     #...##M#     ...#...#     #.T.#M.#
.#...T..     ##...##.     #.T.##.#     M#.#T..#     #...##.#
.####...     ##.T.#M.     #...##.#     ##.#...#     #...#M..
....####     ##...##.     M.###M.M     .....###     ##.####.
```

```
###..#M#    #M.###M.    #...##.#    .###.#.M    ##.#M#..
...#....    ##..M##.    #.#.#...    .#M..#.#    ##.#...#
..T####.    M##.##..    M.....#.    .###....    .....#..
........    .......#    ##M##...    .....##M    M###.M#M

G 15        G 16        G 17        G 18        G 19
...##...    #...####    #...#...    #####M##    ##M.M###
.T.##.T.    #.T.#...    #.T.#.T.    ...#..##    M##.####
...#....    ....#T..    ....#...    .T.#.###    .##.#M#M
##...###    .####...    .####.##    .......#    ..#.#.#.
####.###    .#####.#    .......#    ######.#    #.....#.
........    .#M#M#.#    ##.###.#    M......#    #.#.#...
.T.##.T.    .......#    #M.M#M.M    ##.#.###    #M#M##.#
...##...    #M#M#M##    ##M####M#    ##M#...M    #####M.M

G 20        G 21        G 22        G 23        G 24
M#######    #####...    ...#...#    #...##M#    #.....##
........    #..T#T..    .T.#...#    #M#..#.#    ..###...
.#.##.#.    #...#...    ...#.T.#    ####...M    .##M###.
M#......    #.....##    ##...###    ####.#.#    ....#...
##.##.#M    #####.#M    ####..#M    #M#..#M#    ###.#.##
#...#..#    ........    ...##...    ....####    ....#...
#.T.##..    .######.    T.....#.    M##.##M#    ..T##T..
#...###M    .M#M....    ...##...    ##M....#    ...##...

G 25        G 26        G 27        G 28        G 29
........    #####...    #.......    ...#...#    .M###M##
.######.    ...#M.#.    ..###M#.    M#.#.T.#    .##M#.##
.#....#.    ...##...    .#...##.    ##.#...#    ..#....M
...##M#.    T..#..#M    .#.T.#..    .....###    #...####
.#.####.    ##.#.###    .#...#.#    .#######    ..#.#...
.#..#...    #M......    .M#.#...    .......M    .##....T
.M#M#.T.    ##.##T..    .##.#.#.    .#.#.#.#    ..###...
#####...    ##.M#...    ......#M    .......M    #..M####

G 30        G 31        G 32        G 33        G 34
####...#    ......#M    ####...#    ........    #M#...M#
#M##..T#    T..##...    T..#M#..    .######.    #.#.####
.......#    ...###.#    ...####.    .#...#..    .......M
.#######    ####M#.M    ...#M...    .#T..#.#    .#.###.#
.##...M#    #.T....#    .######.    .#...#..    .#.#M#.#
....####    #...##.M    ..#...#.    .##.###.    .......M
#.#.##.M    #...#M.#    #.#.#.#.    ..#.#M#.    #.#.####
M.#....#    ######.M    #...#...    #.......    #M#...M#

G 35        G 36        G 37        G 38        G 39
...M#...    #####M.#    ######M.    ####M##M    M###M.M#
.#.##.#.    #...##.M    M#M#M##.    ...#.#..    ...##.##
M#....#.    #.T.#M.#    ...#....    ..T#.#.#    M#..#...
##.####M    #...##.M    ##.###.#    ...#.#.#    ###...#M
......##    ##.##M.#    #...#...    #.##....    M##.####
.##.#...    ...###.M    #M#...#.    #.#..##.    .....##M
.##.#..T    .###...#    #####M##.    #...###M    ##.#....
....#...    .....###    ######M.    M.#M####    #M.##M#M

G 40        G 41        G 42        G 43        G 44
#M##...#    #####M.M    #M#...M#    ...#####    .......#
#.##...#    #M####.#    ....####    .#.#...#    .#####.M
#.##.T.#    ..#....#    .#.##...    .#.#.T.#    .#...#.#
...#.###    .##.####    .#.##...    ...#...#    .#T..#.M
.#....##    .......#    ...M#T..    .#.##.##    .#.....#
...##...    M#.###.#    .#.##.##    ........    .#######
#.###.#.    ##...#..    .#......    #.#.#.#.    ..#M#...
M.M###M#    ###M#M##M    ...#M##M    #.......    #.....#M

G 45        G 46        G 47        G 48        G 49
####...#    #####...    #####...    ###..M##    #M.#####
```

```
#M...#.M    ...#..#.    ......#.    M##.####    ##.#...#
###.####    .T.#.#..    .######..    ....#...    .....#..
##M.#M#M    .......#    ...T.#.#    #.###T..    M##.#M#.
M##.....    ###.##.M    .#...#.#    M...#...    ##......
.#..##.#    #M..M#.#    .#...#..    ###...##    #M.#.##.
...####M#   ###.##.#    .######.    ##..#..M    ####.##M
##.M####    #M..M#M#    ........    M..###M#    M....M##

G 50        G 51        G 52        G 53        G 54
M.#####M    #...#...    #...#M##    #M####.M    ##......
#..#....    ..#.#T..    ..#....M    ....M#.#    ...###M#.
##.#.###    .##.#...    .###.#.#    .#####.#    .T.####.
##......    ..#.##.#    ..#..#.M    .#M.#...    ...##...
...##.#.    #.#.M#..    #...##.#    .##...#.    ####..#.
.#......    ..#####.    ##.###.M    ..#####.    T....#..
.###.###    .##...#.    #M.....#    #.......    ...#...#
M###...M    ....#...    ###M#M##    ##M#M#M#    ...#M#.M

G 55        G 56        G 57        G 58        G 59
#...####    ...###M#M   ...##M.#    M###.##M    ###.....
..#....M    ..T##.#.    .#..##.#    ...#.M#.    ....###.
.#M#.#.#    ....#...    M##....#    ##.#.##.    .#M###M#.
.....#.M    ###...#M    ####.#M#    M#......    .####...
##.###.#    M#..####    ...#.##M    ...##.##    .#######
#...#...    ...##...    ..T#....    ##.....#    .#M##M..
#..T#.#.    ##......    ...#.##M    M##.##.M    .....##.
#...#...    M..##T..    ##...###    ....#M.#    ####....

G 60        G 61        G 62        G 63
###...##    ...#...M    #M..#M#M    ....#...
#...#...    ...#.###    ###.....    .##.#.#.
M.#.#...    .T.#..#M    ###.#.#.    ..#.#.#.
#...#T..    ##.##...    #M......    #.#.....
###.####    #M.###.#    ##.####.    ..##.#.#
.....#.M    ##.###.#    ...#....    .###.#.#
M#M#.#.#    #M..#..#    M#M#..T#    .##....#
####...#    ###...##    ####...#    ....#M##
```

PROBLÈME 52

Jeux et nombres

52.1 Hackenbush

Le jeu de *Hackenbush* est joué par deux joueurs **Plein** et **Pointillé** sur un plateau consistant en un ensemble de traits reliés, au moins en un point, à une ligne appelée le *sol*.

À tour de rôle, **Plein** efface un trait plein et **Pointillé** efface un trait en pointillé. Le premier joueur qui ne peut plus effacer de trait, car il n'y en a plus de son type, a perdu.

Quand on efface un trait, tous les traits qui ne sont plus reliés au sol disparaissent.

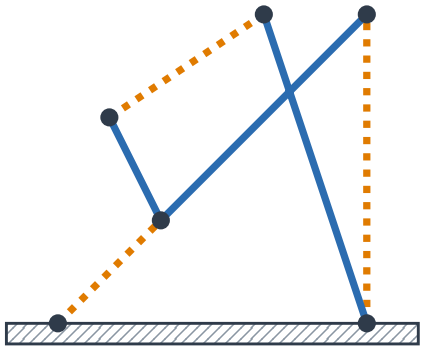
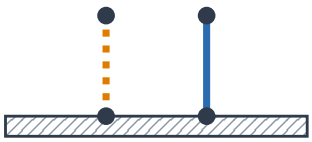


Figure 52.1 — Traits pleins (Plein) en bleu, traits en pointillé (Pointillé) en orange, sol hachuré.

On aimerait associer à chaque plateau un entier relatif $p \in \mathbb{Z}$ indiquant si le plateau est favorable pour **Plein** quand p est grand et défavorable quand p est petit.

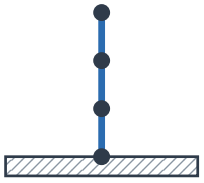
On commence par identifier facilement des parties nulles : on dit qu'un plateau est nul et vaut 0 quand le joueur qui commence à jouer perd nécessairement.

Voici un exemple de plateau nul :



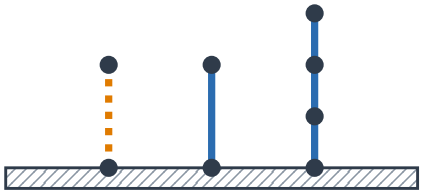
Un plateau qui ne contient que p traits du joueur **Plein** aura la valeur p, et un plateau qui ne contient que p traits du joueur **Pointillé** aura la valeur $-p$. Ainsi, on peut voir la valeur comme le nombre de coups d'avance du joueur **Plein**.

Un plateau de valeur 3 :



Étant donné deux plateaux, on appelle somme des plateaux le plateau obtenu en considérant les deux dessins côte à côte.

La somme des deux plateaux précédents est donc :



Question 1 Si on considère des plateaux simples de valeurs p et q, montrez que la somme des plateaux est de valeur $p + q$.

PARTIE IX

Devoirs MP2I 2021-2022

53 Devoir DS 1 - Programmation OCaml et C 787
54 Devoir DS 2 - Sous-ensembles et points fixes 797
55 Devoir DS 3 - Exponentiation rapide et tri du crêpier 811
56 Devoir DS 3 - Partie pratique (OCaml) 819
57 Devoir DS 4 - Listes chaînées, pavages et backtracking 827
58 Devoir DS 6 - Matrices de Monge et problème du monnayeur 837
59 Devoir DS 8 - Concours blanc - Plus longue sous-suite commune et coloration de graphes 843
60 Devoir Devoir des vacances de Pâques - Dessin de graphes, chasseurs de fantômes, expressions 853
61 Devoir Devoir maison - Lancer de rayons 863

DEVOIR 53

DS 1 - Programmation OCaml et C

53.1 DS 1 — Programmation OCaml et C

MP2I, année 2021-2022. Devoir surveillé 1 du 2 octobre 2021. Durée totale 4h.

Tout document interdit. Tout dispositif électronique interdit.

On vous demande d'écrire lisiblement et de bien présenter. Les programmes en particulier doivent être bien indentés et lisibles.

Bon courage et *Don't panic!* Le barème et les indications de temps sont purement indicatifs.

53.1.1 Reprise des TPs - 10pts

On demande ici de redonner les programmes correspondant plus ou moins directement à des questions posées dans les TP1 à 4.

OCaml - 5pts

Les programmes à donner en réponse ici doivent être écrits en `OCaml`.

Attention

On utilise ici les types par défaut de `OCaml`. C'est particulièrement important pour les listes.

Question 1 *(OCaml - 5 min - 1/2 pt)* Écrire une fonction `pow2 : int -> int` telle que `pow2 n` renvoie 2^n où $n \geq 0$ (c'est acquis que n ne sera jamais négatif donc ne vous en préoccupez pas dans les questions).

Corrigé

```
let rec pow2 n =
  if n = 0
  then 1
  else 2 * pow2 (n-1)
```

Question 2 *(OCaml - 5 min - 1/2 pt)* Écrire une fonction `suplog2 : int -> int` telle que `suplog2 n` renvoie le plus petit $d \in \mathbb{N}$ tel que $n < 2^d$. On supposera $n \geq 0$.

Corrigé

```
let rec suplog2 n =
  if n = 0
  then 0
  else 1 + suplog2 (n/2)
```

Question 3 *(OCaml - 5 min - 1/2 pt)* Écrire une fonction `comb : int -> int -> int` telle que `comb n k` calcule $\binom{n}{k}$ avec la relation du triangle de Pascal :

$$\forall n \in \mathbb{N}^*, \forall k \in \{1, \ldots, n\}, \quad \binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

On rappelle que $\binom{n}{0} = \binom{n}{n} = 1$ et $\binom{n}{k} = 0$ par convention quand $k \notin \{0, \ldots, n\}$.

Corrigé

```
let rec comb n k =
  if k > n || k < 0
  then 0
  else if k = n
  then 1
  else comb (n-1) (k-1) + comb (n-1) k
```

Question 4 *(OCaml - 5 min - 1/2 pt)* Écrire une fonction `trib : int -> int` telle que `trib n` calcule T_n où $T_0 = T_1 = 0, T_2 = 1$ et

$$\forall n \in \mathbb{N}, T_{n+3} = T_{n+2} - T_n T_{n+1}$$

Corrigé

```
let rec trib n =
  if n <= 1
  then 0
  else if n = 2
  then 1
  else trib (n-1) - trib (n-2) * trib (n-3)
```

Question 5 *(10 min - 3/4 pt)* On considère C_n le nombre de noeuds dans l'arbre d'appels récursifs de la fonction `trib`.

Déterminer une relation de récurrence satisfaite par la suite $(C_n)_{n\in\mathbb{N}}$.

Corrigé

On a $C_0 = C_1 = C_2 = 1$ et si $n > 2$,

$$C_n = 1 + C_{n-1} + C_{n-2} + C_{n-3}$$

Question 6 *(OCaml - 10 min - 3/4 pt)* Écrire une fonction `applique : 'a list -> ('a -> 'b) -> 'b list` telle que l'appel à `applique l f` renvoie la liste obtenue en appliquant à chaque élément de `l` la fonction `f`.

Exemple :

```
applique [1;2;3] (fun x -> 4 + x)
(* renvoie *)
[5;6;7]
```

Corrigé

```
let rec applique l f =
  match l with
  | [] -> []
  | t::q -> f t :: applique q f
```

Question 7 *(OCaml - 10 min - 3/4 pt)* Écrire une fonction `nieme : 'a list -> int -> 'a option` telle que l'appel à `nieme l n` renvoie `Some x` où `x` est le n-ième élément de `l` ou `None` s'il n'y en a pas.

Corrigé

```
let rec nieme l n =
  match l, n with
  | [], _ -> None
  | t::_, 0 -> Some t
  | _::q, _ -> nieme q (n-1)
```

Question 8 *(OCaml - 10 min - 3/4 pt)* Écrire une fonction `pourtout : 'a list -> ('a -> bool) -> bool` telle que l'appel à `pourtout l f` renvoie `true` *si et seulement si* tous les éléments `x` de `l` sont tels que `f x` soit égale à `true`.

Corrigé

```
let rec pourtout l f =
  match l with
  | [] -> true
  | t::q -> f t && pourtout q f
```

C - 5pts

Les programmes à donner en réponse ici doivent être écrits en C.

Question 9 *(C - 5 min - 1/2 pt)* Écrire une fonction de signature `int suplog2(int n)` qui renvoie le plus petit entier $d \geq 0$ tel que $0 \leq n < 2^d$.

Corrigé

```
int suplog2(int n)
{
    int p = 1;
    int d = 0;
    while (n >= p)
    {
        p = p*2;
        d = d+1;
    }
    return d;
}
```

Question 10 *(C - 5 min - 1/2 pt)* Écrire une fonction de signature `int factorielle(int n)` qui renvoie $n!$.

Attention

On ne fera pas de fonction récursive.

Corrigé

```
int factorielle(int n)
{
    int p = 1;
    for (int i = 1; i <= n; i++)
    {
        p = p*i;
    }
    return p;
}
```

Question 11 *(C - 5 min - 1/2 pt)* Écrire une fonction de signature `int somme_cubes(int t[], int nb_elts)` qui calcule la somme des cubes des éléments du tableau donné en arguments.

Corrigé

```c
int somme_cubes(int t[], int nb_elts)
{
    int s = 0;
    for (int i = 0; i < nb_elts; i++)
    {
        s = s + t[i]*t[i]*t[i];
    }
    return s;
}
```

Question 12 *(C - 2 min - 1/4 pt)* Écrire une fonction de signature `void swap(int t[], int i, int j)` qui échange les valeurs aux indices `i` et `j` dans le tableau `t`.

Corrigé

```c
void swap(int t[], int i, int j)
{
    int v = t[i];
    t[i] = t[j];
    t[j] = v;
}
```

Question 13 *(C - 10 min - 1 pt)* Écrire une fonction de signature `int indice_minimum(int t[], int i, int j)` qui renvoie l'indice du plus petit élément situé entre les indices i et j (inclus).

On supposera que la fonction sera toujours appelée avec $0 \leq i < j < n$ où n est le nombre d'éléments du tableau.

Corrigé

```c
int indice_minimum(int t[], int i, int j)
{
    int val_min = t[i];
    int indice_min = i;

    for (int k = i+1; k <= j; k++)
    {
        if (val_min > t[k])
        {
            val_min = t[k];
            indice_min = k;
        }
    }

    return indice_min;
}
```

Question 14 *(C - 10 min - 1 pt)* En déduire une fonction de signature `void tri(int t[], int nb_elts)` qui trie le tableau donné en argument à l'aide des deux fonctions précédentes.

Corrigé

```c
void tri(int t[], int nb_elts)
{
    for (int i = 0; i < nb_elts-1; i++)
    {
        int j = indice_minimum(t, i, nb_elts-1);
        swap(t, i, j);
    }
}
```

Question 15 *(2 min - 1/4 pt)* Comment s'appelle ce tri ?

Corrigé

C'est un tri par sélection car on sélectionne le minimum des éléments non triés pour le placer à la bonne position.

Question 16 *(C - 10 min - 1 pt)* Écrire une fonction de signature `int recherche_motif(char s[], char m[])` qui renvoie l'indice de la première occurrence de la chaine `m` dans la chaîne `s` si elle s'y trouve, ou -1 dans le cas contraire.

Corrigé

```c
int recherche_motif(char s[], char m[])
{
    int n = strlen(s);
    int p = strlen(m);
    for (int i = 0; i <= n-p; i++)
    {
        int j;
        for (j = 0; j < p; j++)
        {
            if (s[i+j] != m[j])
            {
                break;
            }
        }
        if (j == p)
        {
            return i;
        }
    }

    return -1;
}
```

53.1.2 Autour du crible d'Eratosthène - 2pts

On cherche ici à déterminer tous les nombres premiers $\leq N$ où N est une limite qu'on se fixe à l'avance.

L'idée du crible d'Eratosthène consiste à aligner tous les nombres de 2 à N, puis d'effectuer le processus suivant :

1. Déterminer le plus petit entier n non barré et non encore étudié.
2. Barrer tous les multiples de n qui sont $> n$.
3. S'il reste des nombres non barrés $> n$ retourner à l'étape 1.
4. Les nombres non barrés sont premiers.

Par exemple pour $N = 9$:

- a) initialisation : 2 3 4 5 6 7 8 9
- b) étape 1 – $n = 2$ on barre donc 4, 6, 8 : 2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ 9
- c) étape 2 – $n = 3$ on barre donc 6, 9 (mais 6 est déjà barré) : 2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ ~~9~~
- d) étape 3 – $n = 5$ mais il n'y a pas de multiples : 2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ ~~9~~
- e) étape 4 – $n = 7$ mais il n'y a pas de multiples : 2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ ~~9~~
- f) étape 5 – il n'y a plus de nombres non barrés, on a donc trouvé les premiers ≤ 9 : 2 3 5 7

Question 17 *(C - 20 min - 2 pt)* Écrire, en C, une fonction de signature `void crible(int p[], int N)` qui prend en entrée un tableau de $N + 1$ valeurs entières (non initialisées) et les remplit de sorte que `p[i]` contienne 1 si i est premier et 0 sinon.

On appliquera la méthode précédente.

On essaiera de faire en sorte que la fonction n'utilise que des additions ou des multiplications par deux (ni multiplication, ni division, ni modulo). Si vous n'y arrivez pas, écrivez tout de même la fonction avec ces opérations (mais vous aurez moins de points).

Corrigé

```c
void crible(int p[], int N)
{
    int i;
    p[0] = 0;
    p[1] = 0;
    for (i = 2; i <= N; i++)
    {
        p[i] = 1;
    }

    for (i = 2; i <= N; i++)
    {
        if (p[i] == 1)
        {
            int m = 2*i;
            while (m <= N)
            {
                p[m] = 0;
                m = m + i;
            }
        }
    }
}
```

53.1.3 Tours de Hanoi - 3pts

Le but de cet exercice est de résoudre le casse-tête suivant :

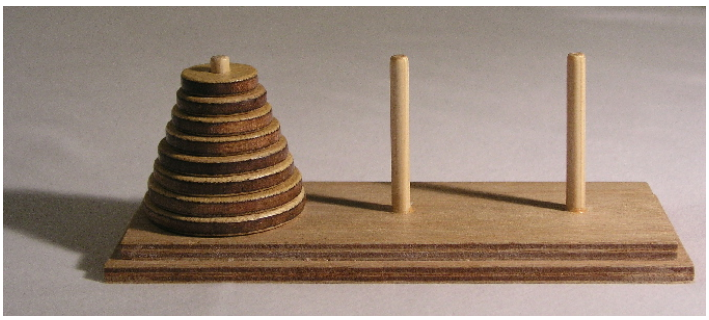


Figure 53.1 — Tours de Hanoi

On a trois poteaux, numérotés 1, 2 et 3, sur lesquels on peut enfiler des disques.

Le but du jeu est de partir de la situation décrite sur la photo : tous les disques empilés sur le poteau 1 (à gauche) et d'aboutir aux disques empilés dans le même ordre mais sur le poteau 3 (à droite). Pour cela on ne peut faire qu'un seul coup : déplacer le disque au sommet d'un poteau vers un autre poteau. Par contre, il est interdit de placer un disque sur un disque plus petit.

Par exemple les coups suivants sont licites :

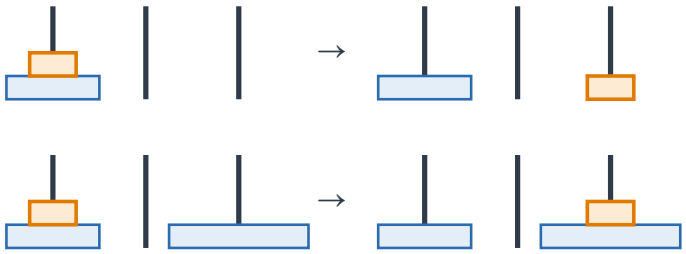


Figure 53.2 — Deux coups licites : le disque déplacé est en orange.

Mais celui-ci ne l'est pas :



Figure 53.3 — Un coup illicite : le disque déplacé (en rouge) est posé sur un disque plus petit.

On considère qu'au départ on dispose de n disques de tailles toutes différentes sur le poteau 1 et que les deux poteaux 2 et 3 sont vides. On cherche à déplacer tous les disques sur le poteau 3. Le poteau 2 joue un rôle crucial, il sert d'espace de travail pour déplacer les disques.

Question 18 *(5 min - 1/2 pt)* Pour quelle(s) valeur(s) de n peut-on se contenter de deux poteaux ?

Corrigé

Uniquement pour $n = 1$ (ou 0 mais c'est un cas extrême) car si on a au moins deux disques, seul le plus petit peut bouger entre les deux poteaux et on se retrouve bloqué.

Question 19 *(10 min - 1/2 pt)* On suppose qu'on sait déplacer $n - 1$ disques d'un poteau à un autre en utilisant un troisième poteau comme pivot, donner une stratégie pour résoudre le problème à n disques.

Corrigé

On déplace les $n - 1$ premiers disques sur le poteau du milieu en utilisant le poteau final comme espace temporaire. On se retrouve alors dans la configuration suivante : le premier poteau contient le plus grand disque, le deuxième les $n - 1$ premiers, et le dernier est vide. On place le plus grand disque sur le troisième poteau. On peut alors déplacer les $n - 1$ disques du milieu sur le poteau final.

Le point clé à remarquer c'est qu'un poteau ait le plus grand disque ou pas dessus ne change rien vis-à-vis des $n - 1$ disques restants : ils sont tous plus petits.

On suppose définie une fonction C `void deplace(int src, int dst)` qui déplace le disque du poteau numéroté `src` (un int) vers celui numéroté `dst`. Ainsi `deplace(1, 2)` déplace un disque du poteau 1 vers le poteau 2.

On indique également qu'il est possible d'écrire des fonctions récursives en C comme en Python en appelant juste la fonction comme dans :

```c
unsigned int fact(unsigned int n)
{
    if (n == 0)
    {
        return 1;
    }
    else
    {
        return n * fact(n-1);
    }
}
```

Question 20 *(C - 10 min - 1 pt)* Écrire une fonction `void hanoi(int n, int src, int dst, int inter)` qui résout Hanoi pour n disques en partant de `src` pour atteindre `dst` et utilisant `inter` comme pivot.

Corrigé

```c
void hanoi(int n, int src, int dst, int inter)
{
    if (n != 0)
    {
        hanoi(n-1, src, inter, dst);
        deplace(src, dst);
        hanoi(n-1, inter, dst, src);
    }
}
```

Question 21 *(10 min - 1 pt)* Soit D_n le nombre de déplacements effectués pour déplacer n disques avec la fonction précédente. Déterminer une relation de récurrence satisfaite par D_n et en déduire l'expression de D_n en fonction de n.

Corrigé

$D_0 = 0$ car on ne fait rien et si $n > 0$, $D_n = 2D_{n-1} + 1$ car on a fait deux appels récursifs et un déplacement.

On a donc $D_n = 1 + 2 + \cdots + 2^{n-1} = 2^n - 1$.

53.1.4 Couvertures - 5pts

On considère une ligne de n cases carrées :



que l'on souhaite recouvrir complètement avec des tuiles de différentes longueurs.

Par exemple, si on dispose de tuiles de longueur 1 ou 2, voici une couverture valide avec deux tuiles de longueur 1 et deux de longueur 2 :



Question 22 *(OCaml - 10 min - 1 pt)* Écrire une fonction OCaml `couvertures12 : int -> int` telle que `couvertures12 n` calcule le nombre de couvertures de n cases avec des tuiles de longueurs 1 ou 2.

Indication : on pourra raisonner sur la tuile la plus à gauche d'une telle couverture.

Corrigé

Dans le cas dégénéré $n = 0$ il n'y a qu'une couverture : c'est l'ensemble vide. Dans le cas $n = 1$ on ne peut placer qu'une tuile de taille 1.

Sinon on place une tuile de taille 1 et on couvre les $n - 1$ restants ou on place une tuile de taille 2 et on couvre les $n - 2$ restants.

```
let rec couvertures12 n =
  if n = 0 || n = 1
  then 1
  else couvertures12 (n-1) + couvertures12 (n-2)
```

Question 23 *(2 min - 1/4 pt)* En relisant la fonction précédente, reliez ce nombre de couvertures à une suite classique.

Corrigé

Il s'agit de la suite de Fibonacci !

Question 24 *(5 min - 1/2 pt)* Montrer qu'il est équivalent de couvrir n cases ainsi et de couvrir un échiquier de $2 \times n$ cases avec des dominos.

Rappel : un domino est un rectangle de deux cases de long, on peut donc le placer verticalement ou horizontalement pour couvrir deux cases de l'échiquier.

Corrigé

Pour couvrir un échiquier $2 \times n$ on a le choix sur la position du premier domino : vertical ou horizontal. Si on le place verticalement, il ne reste comme espace libre que de quoi placer un domino parallèle exactement à côté. Le choix que l'on fait est donc entre un domino horizontal ou deux dominos verticaux. C'est exactement le même choix qu'avec les tuiles précédentes. D'ailleurs on passe de l'un à l'autre en coupant l'échiquier en deux.

Question 25 *(10 min - 3/4 pt)* Déterminer une condition sur n et m pour qu'un échiquier de $n \times m$ cases puisse être recouvert par des dominos.

Corrigé

Si on colorie l'échiquier comme aux échecs, chaque domino est à cheval sur une case blanche et une case noire. Il faut donc que le nombre de cases blanches de l'échiquier soit égal au nombre de cases noires.

Si $n = 2p + 1$, les lignes sont tantôt à p cases noires et $p + 1$ cases blanches, tantôt l'inverse. Pour que cela s'équilibre, il faut que le nombre de lignes soit pair.

Si n est pair, l'équilibre est automatique.

La condition est donc d'avoir n ou m pair. Dans ce cas on peut découper l'échiquier en bandes de tailles $2 \times m$ ou $n \times 2$ et on sait que l'échiquier restant est couvrable (par exemple uniquement avec des dominos parallèles).

On a donc $n \times m$ admet une couverture $\Leftrightarrow$ nm pair (car n ou m pair $\Leftrightarrow$ nm pair).

Question 26 *(OCaml - 20 min - 2 pt)* **(difficile)** Écrire une fonction OCaml `couvertures : int list -> int -> int` telle que `couvertures l n` calcule le nombre de couvertures de n cases avec des tuiles de longueur données par la liste l.

Par exemple, la fonction précédente correspondra à `couvertures [1;2]`.

On ne suppose rien sur l. Si un nombre est présent plusieurs fois, c'est comme si on considérait des tuiles de même longueur mais de couleurs différentes, ce qui augmente le nombre de couvertures.

Corrigé

```
let rec couvertures tl n =
  let rec aux l = match l with
    | [] -> 0
    | t::q -> (if n >= t then couvertures tl (n-t) else 0) + aux q
  in if n = 0 then 1 else aux tl
```

Question 27 *(5 min - 1/2 pt)* Calculer le nombre de couvertures pour la liste `[1;1;1;1]` qui contient donc quatre fois une tuile de longueur 1 pour quatre couleurs différentes.

Corrigé

Ici on a quatre tuiles identiques de longueur 1, donc on a quatre choix par case. Ainsi, s'il y a n cases alors on a 4^n couvertures distinctes.

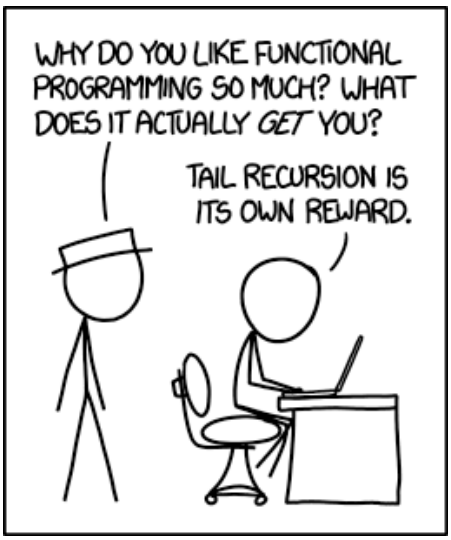


Figure 53.4 — xkcd : tail recursion



Figure 53.5 — Gru et la récursivité

DEVOIR 54

DS 2 - Sous-ensembles et points fixes

54.1 DS 2 — Sous-ensembles et points fixes

MP2I, année 2021-2022. Devoir surveillé 2 du 13 novembre 2021. Durée totale 4h.

Tout document interdit. Tout dispositif électronique interdit.

On vous demande d'écrire lisiblement et de bien présenter. Les programmes en particulier doivent être bien indentés et lisibles. Utilisez de préférence plusieurs couleurs. **Commentez** votre code dès que ce qu'il fait n'est pas évident.

54.1.1 Manipulation de sous-ensembles de $[\![0, n-1]\!]$

On va ici supposer définie une constante entière $n > 0$ accessible directement en `C` (on peut donc écrire `int t[n];` pour déclarer un tableau de cette taille).

L'objectif de ce problème est de réaliser une bibliothèque de manipulation de sous-ensembles de $E = [\![0, n-1]\!] = \{0, 1, 2, ..., n-1\}$.

Pour cela, on va les représenter sous la forme de tableaux de booléens (on suppose `stdbool.h` inclus). Si $S \subset E$ on le représente par le tableau `t` de n booléens tel que $\forall i \in E, i \in S \Leftrightarrow$ `t[i]` = `true`.

Ainsi, le sous-ensemble $\{1, 3, 5\}$ de $[\![0, 5]\!]$ est représenté par le tableau

`i`	0	1	2	3	4	5
`t[i]`	false	true	false	true	false	true

Dans la suite du sujet on utilise le mot **sous-ensemble** pour désigner un tel tableau représentant un sous-ensemble de E.

Convention d'appel : comme les sous-ensembles seront tous représentés par des tableaux de taille n, il n'est pas nécessaire de passer leur taille en argument. On écrira donc juste `f(bool *t)` pour une fonction prenant un tableau, `f(bool *t1, bool *t2)` pour une en prenant deux, et ainsi de suite.

Remarque : on vous laisse ici souvent libre des arguments, des valeurs de retour, de la nomenclature de votre fonction. En contrepartie, on attend que la déclaration des fonctions soit brièvement documentée.

Question 1 Écrire une fonction qui, étant donné un sous-ensemble S de E et un élément x, renvoie `true` si et seulement si $x \in S$.

Corrigé

```c
bool appartient(bool *S, int x)
{
    return S[x];
}
```

Question 2 Écrire une fonction qui renvoie le nombre d'éléments d'un sous-ensemble.

Corrigé

```
int cardinal(bool *S)
{
    int s = 0;

    for (int i = 0; i < n; i++)
    {
        if (S[i])
        {
            s++;
        }
    }

    return s;
}
```

Question 3 Écrire une fonction `void complementaire(bool *e, bool *ce);` qui prend en entrée un sous-ensemble `e` et place son complémentaire dans `ce`.

Corrigé

```
void complementaire(bool *e, bool *ce)
{
    for (int i = 0; i < n; i++)
    {
        ce[i] = !e[i];
    }
}
```

Question 4 Prouver la correction de la fonction précédente.

Corrigé

Il n'y a pas de problèmes de terminaison ici dans la mesure où on est sur une boucle `for` effectuant un nombre fini d'itérations.

Pour la correction, on va montrer l'invariant suivant :

$$I(i) = \forall j \in [\![0, i-1]\!], \texttt{ce}\,[j] = \texttt{true} \Leftrightarrow j \notin \texttt{e}$$

en identifiant `e` et le sous-ensemble qu'il représente.

- **Initialisation** : comme $[\![0, -1]\!] = \emptyset$, on a directement $I(0)$.
- **Hérédité** : si $I(i)$ est vérifiée en début d'itération, en fin d'itération on passe à $i+1$ et on a placé `true` dans `ce` $[i]$ si et seulement si il y avait `false` dans `e` $[i]$ et donc si et seulement si $i \notin$ `e`, ce qui permet de prolonger l'invariant et d'obtenir $I(i+1)$.

On a donc l'invariant $I(n)$ en sortie de boucle et ainsi `ce` contient bien le complémentaire.

Remarque : ici, la vérification est triviale, et c'était plus une question portant sur la méthode.

Question 5 Écrire une fonction `bool disjoint(bool *e1, bool *e2);` qui prend en entrée deux sous-ensembles `e1` et `e2`, puis renvoie `true` si et seulement si les sous-ensembles sont disjoints : ils n'ont aucun élément en commun.

Corrigé

```c
bool disjoint(bool *e1, bool *e2)
{
    bool dis = true;

    for (int i = 0; i < n; i++)
    {
        dis = dis && !(e1[i] && e2[i]);
    }

    return dis;
}
```

Question 6 Écrire une fonction `void union(bool *e1, bool *e2, bool *ue);` qui prend en entrée deux sous-ensembles `e1` et `e2`, puis place leur union dans `ue`.

Corrigé

Attention : à l'écrit cela ne pose pas de problèmes mais on ne peut pas utiliser `union` en C car c'est un mot clé du langage.

```c
void reunion(bool *e1, bool *e2, bool *ue)
{
    for (int i = 0; i < n; i++)
    {
        ue[i] = e1[i] || e2[i];
    }
}
```

Question 7 Écrire une fonction `void intersection(bool *e1, bool *e2, bool *ie);` qui prend en entrée deux sous-ensembles `e1` et `e2`, puis place leur intersection dans `ie`.

Corrigé

```c
void intersection(bool *e1, bool *e2, bool *ie)
{
    for (int i = 0; i < n; i++)
    {
        ie[i] = e1[i] && e2[i];
    }
}
```

Question 8 Écrire une fonction `void difference(bool *e1, bool *e2, bool *de);` qui prend en entrée deux sous-ensembles `e1` et `e2`, puis place la différence `e1` $\setminus$ `e2` dans `de`.

Corrigé

```c
void difference(bool *e1, bool *e2, bool *de)
{
    for (int i = 0; i < n; i++)
    {
        de[i] = e1[i] && !e2[i];
    }
}
```

On note $I = [\![0, 2^n - 1]\!]$. Si $S \subset [\![0, n-1]\!]$, on pose $u(S) = \sum_{i \in S} 2^i$.

Question 9 Écrire une fonction `int ensemble_vers_entier(bool *e);` qui prend en entrée un sous-ensemble e et renvoie l'entier $u(e)$.

Corrigé

```c
int ensemble_vers_entier(bool *e)
{
    int u = 0;
    int p = 1;

    for (int i = 0; i < n; i++)
    {
        if (e[i])
        {
            u = u + p;
        }

        p = p * 2;
    }

    return u;
}
```

Question 10 Écrire une fonction `void entier_vers_ensemble(int u, bool *e);` qui prend en entrée un entier u et place dans e le sous-ensemble tel que $u = u(e)$.

Corrigé

```c
void entier_vers_ensemble(int u, bool *e)
{
    for (int i = 0; i < n; i++)
    {
        e[i] = u % 2 == 1;
        u = u / 2;
    }
}
```

On dit qu'une suite finie $F_0, ..., F_{p-1}$ de sous-ensembles de E est une couverture de E si et seulement si

$$\forall i \in E, \exists j \in [\![0, p-1]\!], i \in F_j$$

Si, de plus, les F_j sont deux à deux disjoints, on dit qu'ils forment une partition de E.

Question 11 Écrire une fonction `bool couverture(int *f, int p);` qui prend en entrée un tableau de p entiers f où pour tout $i \in [\![0, p-1]\!]$, `f[i]` $= u(F_i)$ avec F_i un sous-ensemble de E, et renvoie un booléen indiquant si $(F_0, ..., F_{p-1})$ est une couverture de E.

Indications : on pourra soit garder les ensembles sous forme d'entiers, soit passer par des représentations temporaires en tableau en tirant parti du fait que n est constant pour les déclarer en tant que variables locales.

Corrigé

```c
bool couverture(int *f, int p)
{
    bool couv = true;

    for (int i = 0; i < n; i++)
    {
        bool present = false;
        for (int j = 0; j < p; j++)
        {
            bool fj[n];
            entier_vers_ensemble(f[j], fj);
            if (fj[i])
            {
                present = true;
            }
```

```
        }

        couv = couv && present;
    }

    return couv;
}
```

Question 12 Prouver la correction de la fonction précédente.

Corrigé

On va montrer la correction de la boucle intérieure ; comme elle termine directement, cela signifie de montrer qu'elle vérifie

$$C(i) = \texttt{present} = \texttt{true} \Leftrightarrow \exists k \in [\![0, p-1]\!], i \in S(\texttt{f}\ [k])$$

où $S(k)$ est l'ensemble tel que $u(S(k)) = k$.

Pour cela, on va prouver l'invariant

$$I(j) = \texttt{present} = \texttt{true} \Leftrightarrow \exists k \in [\![0, j-1]\!], i \in S(\texttt{f}\ [k])$$

- **Initialisation** : encore une fois, c'est vide donc direct d'avoir $I(0)$ car on a initialisé `present` à `false`.
- **Hérédité** : si on a $I(j)$ en début d'itération, deux cas : soit $i \in S(\texttt{f}\ [j])$ et on met `present` à `true`, soit $i \notin S(\texttt{f}\ [j])$ et on ne change pas sa valeur ; comme par hypothèse elle valait $\texttt{true} \Leftrightarrow \exists k \in [\![0, j-1]\!], i \in S(\texttt{f}\ [k])$ on a encore $\texttt{true} \Leftrightarrow \exists k \in [\![0, j]\!], i \in S(\texttt{f}\ [k])$.

  On a donc bien $I(j+1)$.

Ainsi $I(p) = C(i)$ est vérifié.

On montre alors facilement que $I'(i) = \texttt{couv} = \bigwedge_{l=0}^{i-1} \exists k \in [\![0, p-1]\!], l \in S(\texttt{f}\ [k])$ est un invariant car vu ce qu'on vient de montrer, le corps de boucle devient $\texttt{couv} = \texttt{couv} \wedge \exists k \in [\![0, p-1]\!], i \in S(\texttt{f}\ [k])$.

On a donc $I'(n) = \texttt{couv} = \bigwedge_{l=0}^{n-1} \exists k \in [\![0, p-1]\!], l \in S(\texttt{f}\ [k])$ ce qui est effectivement la correction.

Question 13 Écrire une fonction `bool partition(int *f, int p);` qui prend en entrée un tableau de p entiers f où pour tout $i \in [\![0, p-1]\!]$, `f[i]` $= u(F_i)$ avec F_i un sous-ensemble de E, et renvoie un booléen indiquant si $(F_0, ..., F_{p-1})$ est une partition de E.

Corrigé

```
bool partition(int *f, int p)
{
    bool couv = true;

    for (int i = 0; i < n; i++)
    {
        int compte = 0;
        for (int j = 0; j < p; j++)
        {
            bool fj[n];
            entier_vers_ensemble(f[j], fj);
            if (fj[i])
            {
                compte++;
            }
        }

        couv = couv && compte == 1;
    }
```

```
    return couv;
}
```

Question 14 Prouver la correction de la fonction précédente.

Corrigé

Il suffit de reprendre rapidement la preuve précédente en remplaçant l'invariant de la boucle intérieure par

$$I(j) = \texttt{compte} = |\{k \in [\![0, j-1]\!] \mid i \in S(\texttt{f}\ [k])\}|$$

ce qui permet d'obtenir la correction de la boucle intérieure

$$C(i) = \texttt{compte} = |\{k \in [\![0, p-1]\!] \mid i \in S(\texttt{f}\ [k])\}|$$

et donc `compte` $= 1 \Leftrightarrow i$ appartient à un unique sous-ensemble.

On conclut par le même invariant que précédemment, vu que c'est à nouveau une conjonction.

54.1.2 Points fixes de fonctions à domaine fini

Cette partie semble reprise, et transposée en C, d'un sujet de concours (style X-ENS, informatique commune) ; la source ne l'indique pas.

Dans ce problème, on s'intéresse aux points fixes des fonctions $f : E \to E$, où E est un ensemble fini. Le calcul effectif et efficace des points fixes de telles fonctions est un problème récurrent en informatique (transformation d'automates, vérification automatique de programmes, algorithmique des graphes, etc.), et admet différentes approches selon la structure de E et les propriétés de f.

On suppose par la suite un entier strictement positif $n > 0$ fixé et accessible comme une constante entière en `C` (on peut donc écrire `int t[n];` pour déclarer un tableau de cette taille) de même nom (dans la suite du sujet, on confondra l'entier n et la variable informatique `n`), et on pose $E_n = \{0, \ldots, n-1\}$. On représente une fonction $f : E_n \to E_n$ par un tableau `t` de taille n, autrement dit $f(x) = \texttt{t[x]}$ pour tout $x = 0, \ldots, n-1$. Ainsi la fonction f_0 qui à $x \in E_{10}$ associe $2x + 1$ modulo 10 est-elle représentée par le tableau `t` suivant :

i	0	1	2	3	4	5	6	7	8	9
t[i]	1	3	5	7	9	1	3	5	7	9

Le temps de calcul d'une fonction `f` de paramètres $p_1, \ldots, p_k$ est défini comme le nombre d'opérations (accès en lecture ou écritures à une case d'un tableau ou à une variable, appel à une des primitives données dans l'énoncé) exécutées par `f` pour ces paramètres ; on note $T(\texttt{f}, n)$ le temps de calcul maximal pris sur tous les paramètres possibles pour n fixé. On dit que `f` s'exécute en temps linéaire s'il existe des réels $\alpha, \beta > 0$ et un entier $N \geq 0$ tels que $\alpha n \leq T(\texttt{f}, n) \leq \beta n$ pour tout $n \geq N$. De même, on dit que `f` s'exécute en temps logarithmique s'il existe des réels $\alpha, \beta > 0$ et un entier $N \geq 0$ tels que $\alpha \log n \leq T(\texttt{f}, n) \leq \beta \log n$ pour tout $n \geq N$.

Recherche de point fixe : cas général

On rappelle que x est un point fixe de la fonction f si et seulement si $f(x) = x$.

Question 15 Écrire une fonction `bool admet_point_fixe(int *t)` qui prend en argument un tableau `t` de taille n et renvoie `true` si la fonction $f : E_n \to E_n$ représentée par `t` admet un point fixe, `false` sinon. Par exemple, `admet_point_fixe` devra renvoyer `true` pour le tableau donné en introduction, puisque 9 est un point fixe de la fonction f_0 qui à x associe $2x + 1$ modulo 10.

Corrigé

```c
bool admet_point_fixe(int *t)
{
    for (int i = 0; i < n; i++)
    {
        if (t[i] == i)
        {
            return true;
        }
    }
    return false;
}
```

Question 16 Écrire une fonction `int nb_points_fixes(int *t)` qui prend en argument un tableau `t` de taille n et renvoie le nombre de points fixes de la fonction $f : E_n \to E_n$ représentée par `t`. Par exemple, `nb_points_fixes` devra renvoyer 1 pour le tableau donné en introduction, puisque 9 est le seul point fixe de f_0.

Corrigé

```c
int nb_points_fixes(int *t)
{
    int nb = 0;

    for (int i = 0; i < n; i++)
    {
        if (t[i] == i)
        {
            nb = nb + 1;
        }
    }

    return nb;
}
```

On note f^k l'itérée k-ième de f, autrement dit

$$\begin{aligned} f^k : [t]E_n &\to E_n \\ x &\mapsto \underbrace{f(f(\ldots f(x))\ldots)}_{k \text{ fois}} \end{aligned}$$

Question 17 Écrire une fonction `int itere(int *t, int x, int k)` qui prend en premier argument un tableau `t` de taille n représentant une fonction $f : E_n \to E_n$, en deuxième et troisième arguments des entiers x, k de E_n et renvoie $f^k(x)$.

Corrigé

```c
int itere(int *t, int x, int k)
{
    for (int i = 0; i < k; i++)
    {
        x = t[x];
    }

    return x;
}
```

Question 18 Écrire une fonction `int nb_points_fixes_iteres(int *t, int k)` qui prend en premier argument un tableau `t` de taille n représentant une fonction $f : E_n \to E_n$, en deuxième argument un entier $k \geq 0$ et renvoie le nombre de points fixes de f^k.

Corrigé

```c
int nb_points_fixes_iteres(int *t, int k)
{
    int nb = 0;

    for (int i = 0; i < n; i++)
    {
        if (itere(t, i, k) == i)
        {
            nb = nb + 1;
        }
    }

    return nb;
}
```

Un élément $z \in E_n$ est dit *attracteur principal* de $f : E_n \to E_n$ si et seulement si z est un point fixe de f, et pour tout $x \in E_n$, il existe un entier $k \geq 0$ tel que $f^k(x) = z$.

Afin d'illustrer cette notion, on pourra vérifier que la fonction f_1 représentée par le tableau ci-dessous admet 2 comme attracteur principal.

i	0	1	2	3	4	5	6
t[i]	5	5	2	2	0	2	2

En revanche, on notera que la fonction f_0 donnée en introduction n'admet pas d'attracteur principal puisque $f_0^k(0) \neq 9$ quel que soit l'entier $k \geq 0$.

Question 19 Écrire une fonction `bool admet_attracteur_principal(int *t)` qui prend en argument un tableau `t` de taille n et renvoie `true` si et seulement si la fonction $f : E_n \to E_n$ représentée par `t` admet un attracteur principal, `false` sinon. On ne requiert pas ici une solution efficace.

Corrigé

```c
bool admet_attracteur_principal(int *t)
{
    for (int z = 0; z < n; z++)
    {
        bool attracteur = true;

        for (int x = 0; x < n; x++)
        {
            /* f^0(x), f^1(x), ..., f^n(x) prennent n+1 valeurs
             * dans 0, 1, ..., n-1 qui compte n valeurs distinctes
             * donc on a deux valeurs égales. Or, s'il existe
             * k<l tels que f^k(x) = f^l(x) et qu'on n'a pas vu
             * z, on va rester bloqué sur les mêmes valeurs à partir
             * de k, donc on ne verra jamais z. Si au contraire, z apparait
             * alors on reste sur z car c'est un point fixe.
             * Dans tous les cas, soit z n'est pas accessible, soit f^n(x) = z
             */
            attracteur = attracteur && itere(t, x, n) == z;
        }

        if (attracteur)
        {
            return true;
        }
    }

    return false;
}
```

On suppose aux questions 20 et 21 que f admet un attracteur principal. Le *temps de convergence* de f en $x \in E_n$ est le plus petit entier $k \geq 0$ tel que $f^k(x)$ soit un point fixe de f. Pour la fonction f_1 ci-dessus, le temps de convergence en 4 est 3. En effet, $f_1(4) = 0$, $f_1^2(4) = 5$, $f_1^3(4) = 2$, et 2 est un point fixe de f_1. On note $\texttt{tc}\ (f, x)$ le temps de convergence de f en x.

Question 20 Écrire une fonction `int temps_de_convergence(int *t, int x)` qui prend en premier argument un tableau `t` de taille n représentant une fonction $f : E_n \to E_n$ qui admet un attracteur principal, en deuxième argument un entier x de E_n, et renvoie le temps de convergence de f en x. On pourra admettre que $\texttt{tc}\ (f, x)$ vaut 0 si x est un point fixe de f, et $1 + \texttt{tc}\ (f, f(x))$ si x n'est pas un point fixe de f.

Corrigé

```c
int temps_de_convergence(int *t, int x)
{
    if (t[x] == x)
        return 0;
    return 1 + temps_de_convergence(t, t[x]);
}
```

Question 21 Écrire une fonction `temps_de_convergence_max(t)` qui prend en argument un tableau `t` de taille n représentant une fonction $f : E_n \to E_n$ qui admet un attracteur principal et renvoie $\max_{i=0\ldots n-1} \texttt{tc}\ (f, i)$. On impose un temps de calcul **linéaire** en la taille n du tableau (une fonction ne respectant pas cette condition ne rapportera pas de point !). À titre d'indication, on pourra au besoin créer des tableaux intermédiaires en tirant parti du fait que n est une constante.

On ne demande pas de démonstration du fait que le temps de calcul de la solution proposée est linéaire, mais il est impératif d'expliquer clairement le fonctionnement de votre fonction.

Corrigé

```c
/* Version naive et quadratique */
int temps_de_convergence_max_n2(int *t)
{
    int m = 0; // atteint pour l'attracteur

    for (int x = 0; x < n; x++)
    {
        int tx = temps_de_convergence(t, x);
        if (tx > m)
        {
            m = tx;
        }
    }

    return m;
}

/* Version linéaire en calculant rapidement les temps
 * de convergence le long des valeurs prises par les itérés de f
 */
int temps_de_convergence_max(int *t)
{
    int m = 0; // max atteint pour l'attracteur
    int temps[n]; // va contenir soit le temps précalculé, soit -1

    for (int x = 0; x < n; x++)
    {
        if (t[x] == x)
        {
            temps[x] = 0; // c'est un point fixe
        }
        else
```

```
        {
            temps[x] = -1;
        }
    }

    for (int x = 0; x < n; x++)
    {
        int iteres[n]; // les itérés en partant de x
        int k = 0;
        int u = x;
        iteres[k] = u;
        while (temps[u] < 0)
        {
            u = t[u];
            k = k + 1;
            iteres[k] = u;
        }
        // on est tombe sur un point fixe
        // donc sur l'attracteur principal
        for (int r = k-1; r >= 0; r--)
        {
            temps[iteres[r]] = 1 + temps[iteres[r+1]];
        }

        if (temps[iteres[0]] > m) // c'est le plus grand temps de tous
        {
            m = temps[iteres[0]];
        }
    }

    return m;
}
```

Recherche efficace de points fixes

Toute fonction `point_fixe(t)` retournant un point fixe d'une fonction arbitraire est de complexité au mieux linéaire en n. On s'intéresse maintenant à des améliorations possibles de cette complexité lorsque la fonction considérée possède certaines propriétés spécifiques. Nous examinons deux cas.

54.1.2.2.1 Premier cas

Le premier cas que nous considérons est celui d'une fonction croissante de E_n dans E_n. On rappelle qu'une fonction $f : E_n \to E_n$ est croissante si et seulement si pour tous $x, y \in E_n$ tels que $x \leq y$, $f(x) \leq f(y)$.

On admet qu'une fonction croissante de E_n dans E_n admet toujours un point fixe.

À titre d'exemple, la fonction dont le tableau et le graphe sont donnés ci-dessous est croissante. Elle a deux points fixes, à savoir les entiers 5 et 7.

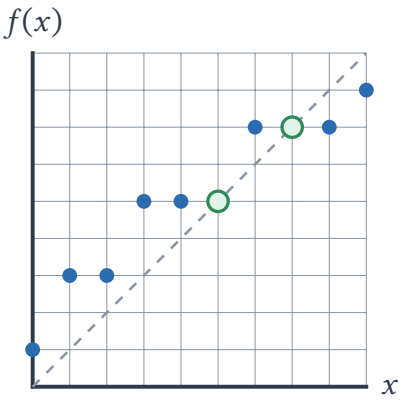


Figure 54.1 — Graphe de la fonction croissante du tableau ; ses points fixes 5 et 7 sont sur la diagonale.

i	0	1	2	3	4	5	6	7	8	9
t[i]	1	3	3	5	5	5	7	7	7	8

Question 22 Écrire une fonction `bool est_croissante(int *t)` qui prend en argument un tableau `t` de taille n et renvoie `true` si la fonction représentée par `t` est croissante, et `false` sinon. On impose un temps de calcul **linéaire** en la taille n du tableau. On ne demande pas de démonstration du fait que le temps de calcul de la solution proposée est linéaire.

Corrigé

```
bool est_croissante(int *t)
{
    for (int i = 0; i < n-1; i++)
    {
        if (t[i] > t[i+1])
        {
            return false;
        }
    }

    return true;
}
```

Question 23 Écrire une fonction `int point_fixe(int *t)` qui prend en argument un tableau `t` de taille n représentant une fonction croissante $f : E_n \to E_n$, et retourne un entier $x \in E_n$ tel que $f(x) = x$. On impose un temps de calcul **logarithmique** en la taille n du tableau. On ne demande pas ici de démonstration du fait que le temps de calcul de la solution proposée est logarithmique, ceci étant le sujet de la question suivante.

Corrigé

Recherche dichotomique en conservant l'invariant $\texttt{t[i]} \geq i$ et $\texttt{t[j]} \leq j$ (vrai au départ car les valeurs sont dans E_n). En sortie de boucle, $j = i + 1$ (ou $i = j$ si $n = 1$) : si $\texttt{t[i]} > i$ alors $\texttt{t[i]} \geq j$ donc, par croissance, $j \geq \texttt{t[j]} \geq \texttt{t[i]} \geq j$ et j est un point fixe.

```
// simple recherche dichotomique
int point_fixe(int *t)
{
    int i = 0;
    int j = n-1;
    while (j - i > 1)
    {
        int m = (i+j)/2;
        if (t[m] == m)
        {
            return m;
        }
        else if (t[m] < m)
        {
            j = m;
        }
        else
        {
            i = m;
        }
    }

    // ici j = i+1 (ou i = j si n = 1), t[i] >= i et t[j] <= j
    if (t[i] == i)
    {
        return i;
    }
```

```
    return j;
}
```

Question 24 Démontrer que la fonction de la question 23 termine.

Corrigé

Voir cours, $j - i$ est un variant de boucle.

54.1.2.2.2 Deuxième cas

On peut généraliser la notion de fonction croissante comme suit. On rappelle qu'une relation binaire $\preccurlyeq$ sur un ensemble E est une relation d'ordre si et seulement si elle est réflexive ($x \preccurlyeq x$ pour tout $x \in E$), anti-symétrique (pour tous $x, y \in E$, si $x \preccurlyeq y$ et $y \preccurlyeq x$, alors $x = y$), et transitive (pour tous $x, y, z \in E$, si $x \preccurlyeq y$ et $y \preccurlyeq z$ alors $x \preccurlyeq z$). Soit $\preccurlyeq$ une relation d'ordre sur E. Une fonction $f : E \to E$ est *croissante au sens de* $\preccurlyeq$ si et seulement si pour tous $x, y \in E$, $x \preccurlyeq y$ implique $f(x) \preccurlyeq f(y)$.

Ceci généralise la notion de fonction croissante de E_n dans E_n, que l'on retrouve en prenant $E = E_n$ et $\preccurlyeq$ la relation d'ordre $\leq$. On s'intéresse dorénavant à d'autres relations d'ordre sur E_n.

On dit qu'un élément m de E est un *plus petit élément* de E au sens de $\preccurlyeq$ si et seulement si, pour tout $x \in E$, $m \preccurlyeq x$. On admet que pour tout ensemble fini E, muni d'une relation d'ordre $\preccurlyeq$ et qui admet un plus petit élément m au sens de $\preccurlyeq$, pour toute fonction croissante $f : E \to E$ au sens de $\preccurlyeq$, il existe un entier $k \geq 0$ tel que $f^k(m)$ est un point fixe de f dans E.

Question 25 Soit E un ensemble fini quelconque muni d'une relation d'ordre $\preccurlyeq$ et admettant un plus petit élément m au sens de $\preccurlyeq$. Soit $f : E \to E$ une fonction croissante au sens de $\preccurlyeq$, et soit $k \geq 0$ un entier tel que $f^k(m)$ soit un point fixe de f dans E. Démontrer que $f^k(m)$ est en fait le plus petit point fixe de f au sens de $\preccurlyeq$, autrement dit que pour tout autre point fixe x de f dans E, on a $f^k(m) \preccurlyeq x$.

Corrigé

Comme f est croissante, alors f^k est croissante, donc $m \preccurlyeq x \Rightarrow f^k(m) \preccurlyeq f^k(x) = x$.

Nous nous intéressons maintenant à un choix particulier d'ordre $\preccurlyeq$, appelé *ordre de divisibilité* et noté $|$. Précisément, on note $a \mid b$ la relation d'ordre « a divise b » sur les entiers positifs, vraie si et seulement s'il existe un entier $c \geq 0$ tel que $ca = b$. Ainsi l'ensemble E_{10} ordonné par divisibilité peut se représenter graphiquement comme suit :

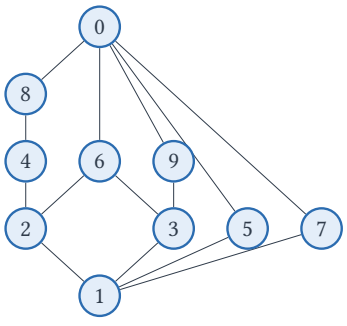


Figure 54.2 — L'ensemble E_{10} ordonné par divisibilité.

D'après la définition donnée précédemment, une fonction $f : E_n \to E_n$ *croissante au sens de l'ordre de divisibilité* est une fonction telle que pour tout x, y dans E_n, si $x \mid y$ alors $f(x) \mid f(y)$. Par exemple la fonction représentée par le tableau ci-dessous est croissante au sens de l'ordre de divisibilité.

i	0	1	2	3	4	5	6	7	8	9
t[i]	0	2	4	6	4	8	0	2	0	6

On remarque que, par la question 25, toute fonction de E_n dans E_n croissante au sens de l'ordre de divisibilité a un plus petit point fixe au sens de l'ordre de divisibilité.

On rappelle que le pgcd de deux entiers $x \geq 1$ et $y \geq 1$ est le plus grand entier non nul qui divise x et y. On étend cette définition à des entiers naturels quelconques, en convenant de définir le pgcd d'un entier $x \geq 0$ et de 0 comme valant x.

Question 26 Soit f une fonction de E_n dans E_n, croissante au sens de l'ordre de divisibilité, et notons $x_1, \ldots, x_m$ les points fixes de f dans E_n. Montrer que le plus petit point fixe de f au sens de l'ordre de divisibilité est exactement le pgcd de $x_1, \ldots, x_m$.

Corrigé

Si p est le pgcd des x_i et que y est le plus petit point fixe, on a $p \mid y$ car y est l'un des x_i. On va montrer que $y \mid p$.

Pour cela, on remarque que $p \in E_n$ et que pour $i \in [\![1, m]\!]$, $p \mid x_i$ et par croissance de f, $f(p) \mid f(x_i) = x_i$. Donc $f(p)$ divise leur pgcd : $f(p) \mid p$. Par croissance de f, on en déduit par récurrence que $f^{k+1}(p) \mid f^k(p)$ pour tout k : la suite $(f^k(p))_k$ est décroissante pour la divisibilité. Comme E_n est fini, elle est nécessairement stationnaire (tant qu'elle n'est pas stationnaire, elle décroît strictement et ne peut donc pas repasser par une même valeur), donc il existe k tel que $f^k(p)$ soit un point fixe de f. Comme y est le plus petit point fixe, on a $y \mid f^k(p)$ et $f^k(p) \mid p$, donc $y \mid p$ par transitivité.

Question 27 Écrire une fonction `int pgcd_points_fixes(int *t)` qui prend en argument un tableau `t` de taille n représentant une fonction de E_n dans E_n, croissante au sens de la divisibilité, et renvoie le pgcd de ses points fixes. On impose un temps de calcul **logarithmique** en la taille n du tableau. On ne demande pas ici de démonstration du fait que le temps de calcul de la solution proposée est logarithmique, ceci étant le sujet de la question qui suit.

Corrigé

1 est le plus petit élément de E_n pour la divisibilité (on suppose ici $n \geq 2$), donc d'après les questions 25 et 26 :

```c
int pgcd_points_fixes(int *t)
{
    // il suffit de trouver le point fixe accessible
    // depuis 1
    int x = 1;

    while (t[x] != x)
    {
        x = t[x];
    }

    return x;
}
```

Question 28 Justifier que la fonction de la question 27 a un temps de calcul logarithmique en la taille n du tableau.

Corrigé

Quand on boucle, comme $f^k(1) \mid f^{k+1}(1)$ et que ces deux nombres sont différents : soit $f^{k+1}(1) \geq 2f^k(1)$ (multiple propre), soit $f^{k+1}(1) = 0$ (cas trivial qu'on peut ignorer pour le moment).

Comme on double $f^k(1)$ à chaque étape et que les valeurs sont $\leq n$, on en effectue au plus $\log n$.

Dans le cas où on obtient 0, on s'arrête à l'itération suivante.

DEVOIR 55

DS 3 - Exponentiation rapide et tri du crêpier

55.1 DS 3 — Exponentiation rapide et tri du crêpier

MP2I, année 2021-2022. Devoir surveillé 3 du 16 décembre 2021, partie écrite. Durée totale 2h. La partie pratique sur machine est sur une page séparée.

Tout document interdit. Tout dispositif électronique interdit.

On vous demande d'écrire lisiblement et de bien présenter. Les programmes en particulier doivent être bien indentés et lisibles. Utilisez de préférence plusieurs couleurs. **Commentez** votre code dès que ce qu'il fait n'est pas évident.

55.1.1 Pirelli

On considère la fonction suivante :

```
let expo_naive x n =
  let p = ref 1 in
  for i = 1 to n do
    p := !p * x
  done;
  !p
```

Question 01 Montrer que la fonction précédente calcule x^n en considérant sa terminaison acquise.

Corrigé

Ici, c'est une boucle `for` donc on a l'indice géré pour nous ce qui peut compliquer la vérification de l'invariant. On pourrait le réécrire ainsi :

```
let expo_naive x n =
  let p = ref 1 in
  let i = ref 1 in
  while !i <= n do
    p := !p * x;
    i := !i + 1
  done;
  !p
```

En pratique, on dira donc qu'on considère dans une boucle `for i = a to b` que i vaut a avant la première itération, qu'on incrémente i à la fin de l'itération et ainsi que i vaut $b+1$ en sortie.

En faisant cela, on considère la propriété $I(i, p) := p = x^{i-1}$.

- **Init** : avant la boucle, on peut considérer que $i = 1$ et on a bien $p = 1 = x^0$ donc $I(i, p)$ est vérifié.
- **Her** : si $I(i, p)$ est vérifié en début d'itération on a $p = x^{i-1}$ et donc $p \times x = x^i$ donc on a bien $I(i+1, p \times x)$ vérifié.

Ainsi, la propriété est vérifiée tout au long de la boucle : c'est un invariant. En sortie de boucle, on a $p = x^{n+1-1} = x^n$.

Question 02 Combien de multiplications effectue cette fonction pour calculer x^n ?

Corrigé

On effectue une multiplication par itération donc n multiplications.

On considère maintenant la fonction :

```
let expo_binaire x n =
  let p = ref 1 in
  let a = ref x in
  let k = ref n in
  while !k > 0 do
    if !k mod 2 = 1
    then p := !p * !a;
    a := !a * !a;
    k := !k / 2
  done;
  !p
```

Question 03 Donner l'état des variables à chaque itération dans l'appel de `expo_binaire 3 15`.

Corrigé

p	a	k
1	3	15
3	9	7
27	81	3
2187	6561	1
14348907	43046721	0

Question 04 Prouver la correction de cette fonction :

1. sa terminaison ;
2. le fait qu'elle calcule x^n.

Corrigé

1. La quantité k est un variant de boucle car elle est entière, positive et dans une boucle, $k > 0$ donc $\lfloor k/2 \rfloor < k$ et elle diminue strictement à chaque itération. Ainsi, la boucle termine car elle admet un variant.

2. On va montrer que la propriété $I(a, p, k) := a^k p = x^n$ est un invariant de boucle.

   Init : avant la première itération, on a $a^k p = x^n \times 1 = x^n$ donc $I(x, 1, n)$ est vérifié.

   Her : si $I(a, p, k)$ est vérifié au début d'une itération :

   - soit $k = 2k'$ et $x^n = a^k p = (a^2)^{k'} p$ donc $I(a^2, p, k')$ est vérifié ;
   - soit $k = 2k' + 1$ et $x^n = a^k p = (a^2)^{k'} (ap)$ donc $I(a^2, ap, k')$ est vérifié.

   Dans tous les cas, $I(a, p, k)$ est vérifié en fin d'itération avec les nouvelles valeurs des variables.

   I est donc un invariant et en sortie de boucle, on a $I(a, p, 0)$ vérifié donc $x^n = a^0 p = p$. On renvoie bien x^n.

Soit $n \in \mathbb{N}$, on note

- $\lfloor \log_2 n \rfloor$ l'unique entier p tel que $2^p \leq n < 2^{p+1}$.
- $\lceil \log_2 n \rceil$ l'unique entier p tel que $2^{p-1} < n \leq 2^p$.
- $\nu(n)$ le nombre de 1 dans l'écriture binaire de n. Plus précisément, si $n = a_0 + a_1 2 + a_2 4 + \ldots + a_{d-1} 2^{d-1}$ avec $a_0, \ldots, a_{d-1} \in \{0, 1\}$ alors il y a exactement $\nu(n)$ des a_i qui valent 1.

Question 05 On suppose que $n > 0$. Montrer que la fonction `expo_binaire` effectue $\lceil \log_2 n \rceil + \nu(n)$ multiplications pour calculer x^n.

Corrigé

On écrit $n = \overline{1 a_{d-2} \ldots a_0}^2$ avec d chiffres en binaire, c'est-à-dire $n = a_0 + a_1 2 + \ldots + a_{d-2} 2^{d-2} + 2^{d-1}$ avec $a_0, \ldots, a_{d-2} \in \{0, 1\}$. On a $2^{d-1} \leq n < 2^d$, donc $d = \lfloor \log_2 n \rfloor + 1$.

À chaque étape, on décale dans k l'écriture binaire de n vers la droite :

$$\overline{1 a_{d-2} \ldots a_0}^2 \to \overline{1 a_{d-2} \ldots a_1}^2 \ldots \to \overline{1 a_{d-2}}^2 \to \overline{1}^2 \to 0$$

On effectue donc exactement d itérations, et `k mod 2` va prendre les valeurs $a_0, a_1, \ldots, a_{d-2}, 1$: on aura bien $\nu(n)$ valeurs 1 entraînant une multiplication.

Au final, on a donc $\nu(n)$ multiplications issues du test et d multiplications dues à la mise au carré de a, soit $\lfloor \log_2 n \rfloor + 1 + \nu(n)$ multiplications en tout.

- Si n n'est pas une puissance de 2, on a $2^{d-1} < n < 2^d$, donc $d = \lceil \log_2 n \rceil$ et on obtient bien $\lceil \log_2 n \rceil + \nu(n)$ multiplications.
- Si $n = 2^{d-1}$, on a $\lceil \log_2 n \rceil = d - 1$ et $\nu(n) = 1$: la fonction effectue $d + 1 = \lceil \log_2 n \rceil + \nu(n) + 1$ multiplications, une de plus que la formule de l'énoncé (par exemple 2 multiplications pour $n = 1$: $1 \times x$ puis $x \times x$).

Note : c'est un $O(\log_2 n)$.

L'algorithme précédent effectue des multiplications non nécessaires pour gérer les variables : le premier calcul $1 \times x$ et la dernière mise au carré. Si on effectue ce principe à la main, pour calculer x^{15} on place x dans p (pas de multiplications cf. remarque), puis on calcule x^2, on place $x^3 = x \times x^2$ dans p, puis on calcule x^4, on place $x^7 = x^3 \times x^4$ dans p, on calcule x^8 et on place $x^{15} = x^7 \times x^8$ dans p. Cette méthode de décomposition en binaire calcule ainsi x^{15} en six multiplications.

On peut faire mieux, en effet, on peut remarquer que $15 = 3 \times 5$. Pour cela, on peut calculer $x^3 = x \times x \times x = y$ en deux multiplications et $x^{15} = y^5 = (y^2)^2 \times y$ en trois multiplications en calculant $x^6 = x^3 \times x^3$, $x^{15} = x^6 \times x^6 \times x^3$. Au total, on aura ainsi effectué cinq multiplications.

La méthode qu'on vient d'utiliser est appelée la méthode du facteur :

- si n est non premier et > 1, on écrit $n = p \times q$ avec p le plus petit facteur premier de n. On a alors $q > 1$. Pour calculer x^n, on commence par calculer $x^p = y$ puis $y^q = (x^p)^q = x^{pq} = x^n$ par cette méthode.
- si n est premier, on écrit $x^n = x \times x^{n-1}$ et on applique la méthode à $n - 1$.
- si $n = 1$, on a directement $x = x^1$.

Question 06 Comparer les deux méthodes pour $n = 55$ et $n = 33$. Que peut-on en conclure ?

Corrigé

Pour $n = 55$:

- **binaire** : comme $55 = \overline{110111}^2$, on calcule x^2 qui place $x^3 = x^2 \times x$ dans p, x^4 qui place $x^7 = x^3 \times x^4$, x^8, x^{16} qui place $x^{23} = x^{16} \times x^7$ dans p, x^{32} qui place $x^{55} = x^{32} \times x^{23}$ dans p. Au total, on a effectué 5 mises au carré et 4 mises à jour de p, donc 9 multiplications. C'est bien $\lceil \log_2 55 \rceil + \nu(55) - 2$ car on a omis les deux multiplications inutiles $p = 1 \times x$ et la mise au carré finale.

Si on reprend les puissances calculées on a donc $x^2, x^3, x^4, x^7, x^8, x^{16}, x^{23}, x^{32}, x^{55}$, à chaque fois avec une multiplication.

- **facteur** : comme $55 = 5 \times 11$ on calcule $x^5 = x \times x^4 = x \times (x \times x)^2$, on a donc calculé x^2, x^4, x^5 avec 3 multiplications. Pour calculer $x^{55} = (x^5)^{11} = a^{11}$ on écrit $a^{11} = a \times a^{10} = a \times (a^2)^5$, donc une fois calculé a^2, pour calculer $(a^2)^5$ on doit encore effectuer 3 multiplications. On a donc, en tout, $3 + 2 + 3 = 8$ multiplications.

  Si on reprend les puissances calculées on a donc $x^2, x^4, x^5, x^{10}, x^{20}, x^{40}, x^{50}, x^{55}$, à chaque fois avec une multiplication.

Pour $n = 33$:

- **binaire** : comme $33 = \overline{100001}^2$, on calcule $x^2, x^4, x^8, x^{16}, x^{32}, x^{33}$ et on a donc effectué 6 multiplications.
- **facteur** : on a $33 = 3 \times 11$ donc, en suivant le calcul précédent, on calcule $x^2, x^3, x^6, x^{12}, x^{24}, x^{30}, x^{33}$ et on a donc effectué 7 multiplications.

Conclusion : aucune des méthodes n'est meilleure que l'autre.

Pour noter ces calculs de puissances, on peut considérer la suite finie des exposants utilisés. Ainsi, pour calculer x^{15} par la méthode binaire on obtient la chaine $(1, 2, 3, 4, 7, 8, 15)$ qui est de longueur 7. En effectuant la méthode du facteur, on obtient la chaine $(1, 2, 3, 6, 12, 15)$ de longueur 6.

Question 07 Donner, sans les justifier, les chaines correspondant aux deux méthodes pour 16, 27 et 125.

Corrigé

- pour 16 : $(1, 2, 4, 8, 16)$ dans les deux cas.
- pour 27 : $(1, 2, 3, 4, 8, 11, 16, 27)$ avec la méthode binaire et $(1, 2, 3, 6, 9, 18, 27)$ avec la méthode du facteur.
- pour 125 : $(1, 2, 4, 5, 8, 13, 16, 29, 32, 61, 64, 125)$ avec la méthode binaire et $(1, 2, 4, 5, 10, 20, 25, 50, 100, 125)$ avec la méthode du facteur.

Définition 55.1

Une suite finie $(n_0, ..., n_r)$ est appelée une chaine de calcul pour x^n si :

- $n_0 = 1$
- $n_r = n$
- $\forall k \in [\![1, r]\!], \exists i, j \in [\![0, k-1]\!], n_k = n_i + n_j$

La dernière condition exprime le fait qu'on a pu calculer $x^{n_k} = x^{n_i} \times x^{n_j}$.

Question 08 1. Montrer par récurrence sur r que toute chaine $(n_0, ..., n_r)$ de calcul pour x^n vérifie

```
$$
\forall k \in \range{0}{r}, n_k \le 2^k
$$
```

2. En déduire que tout calcul de x^n effectue au moins $\lceil \log_2 n \rceil$ multiplications.
3. Donner une infinité de valeurs de n pour lesquelles ce minorant est atteint.

Corrigé

1. On raisonne par récurrence sur r.
   - **Init** : si $r = 0$ la seule chaine est (1) et on a bien $1 \leq 2^0$.

- **Her** : si c'est vrai pour $r-1 \geq 0$, soit $(n_0, ..., n_r)$ une chaine de calcul ; on a $(n_0, ..., n_{r-1})$ chaine de calcul donc $\forall k \in [\![0, r-1]\!], n_k \leq 2^k$ et ainsi par hypothèse on a une décomposition $n_r = n_i + n_j \leq 2^i + 2^j \leq 2 \times 2^{r-1} = 2^r$. Donc, on a bien l'hypothèse au rang r.

On a validé la récurrence.

2. On remarque que toute chaine de longueur $r+1$ effectue r multiplications. Comme $n_r = n \leq 2^r$ on a $\log_2 n \leq r$ et comme r est un entier $\lceil \log_2 n \rceil \leq r$.
3. C'est atteint pour les 2^r où on a une chaine $(1, 2, 4, \cdots, 2^{r-1}, 2^r)$ qui effectue $r = \log_2(2^r)$ multiplications.

55.1.2 Breizhsort

Si `t` est un tableau et $0 \leq i \leq j < |t|$ des indices, on notera `t[i:j]` le sous-tableau de `t` composé des valeurs entre les indices i et j inclus.

On va définir dans la suite du sujet une fonction

```
retourner : 'a array -> int -> int -> unit
```

telle que `retourner t i j` retourne le sous-tableau `t[i:j]`.

Ainsi, si `t = [| 1; 2; 3; 4; 5; 6 |]` alors `retourner t 2 4` change `t` en `[| 1; 2; 5; 4; 3; 6 |]` car `t[2:4] = [| 3; 4; 5 |]`.

Question 09 Quel est le contenu de `t` à l'issue du programme suivant ?

```
let t = [| 5; 8; 2; 0; 3; 6; 5; 2; 1 |];;
retourner t 1 7;;
```

Note : vous pouvez ignorer les `;;` ils ne sont là que pour s'assurer qu'on puisse avoir cette interaction dans un toplevel `OCaml`.

Corrigé

```
[| 5; 2; 5; 6; 3; 0; 2; 8; 1 |]
```

On considère les fonctions suivantes :

```
let echanger t i j =
  let temp = t.(i) in
  t.(i) <- t.(j);
  t.(j) <- temp

let rec retourner t i j =
  if i < j
  then begin
    echanger t i j;
    retourner t (i+1) (j-1)
  end
```

Question 10 Montrer que `retourner` termine.

Corrigé

On remarque que $j - i$ est entière ; si elle est négative ou nulle on termine, sinon on effectue un appel tel que $(j-1) - (i+1) = j - i - 2 < j - i$ donc elle diminue strictement.

Ainsi, il ne peut y avoir qu'un nombre fini d'appels récursifs, la fonction termine.

Question 11 Montrer que `retourner` correspond bien à la spécification de l'énoncé. On pourra montrer par récurrence sur $j - i + 1$ (le $+1$ n'est là que pour déplacer l'initialisation au cas de base) que

$$\forall k \in [\![i, j]\!], \texttt{t.(k)} = \texttt{t'.(i+j-k)}$$

où `t'` est la valeur initiale de `t`.

Corrigé

On suit l'énoncé, par récurrence forte sur $j - i + 1$.

- **Init** : pour $j - i + 1 \leq 1$ on a $j \leq i$, donc soit $[\![i, j]\!] = \emptyset$, soit $i = j$ et la propriété $\texttt{t.(i)} = \texttt{t'.(i)}$ est vérifiée car on ne fait rien.
- **Her** : si la propriété est vérifiée pour toutes les valeurs de $j - i + 1$ inférieures ou égales à n, avec $n \geq 1$, on considère i, j tels que $j - i + 1 = n + 1$, c'est-à-dire que $j - i = n \geq 1$ et donc $i < j$. On échange les valeurs aux indices i et j, ce qui donne la propriété pour $k = i$ et $k = j$, puis on appelle `retourner t (i+1) (j-1)` avec $(j - 1) - (i + 1) + 1 = n - 1$: par hypothèse, on a alors

  $$\forall k \in [\![i + 1, j - 1]\!], \texttt{t.(k)} = \texttt{t'.(i+1+j-1-k)} = \texttt{t'.(i+j-k)}$$

  (les cases d'indices $i + 1$ à $j - 1$ n'ayant pas été modifiées par l'échange). On a donc la propriété voulue.

On considère maintenant les fonctions suivantes :

```
let indice_max t k =
  let m = ref 0 in
  for i = 1 to k do
    if t.(!m) < t.(i)
    then m := i
  done;
  !m

let placer t k =
  let i = indice_max t k in
  if i <> k
  then begin
    retourner t 0 i;
    retourner t 0 k
  end
```

Question 12 Montrer qu'en sortie de l'appel à `placer t k` le tableau `t` vérifie :

$$\forall i \in [\![0, k]\!], \texttt{t.(i)} \leq \texttt{t.(k)}$$

Corrigé

- On commence par montrer que la fonction `indice_max` fait ce qu'on attend d'elle. Pour cela, on considère la propriété

  $$I(i, m) \coloneqq \texttt{t.(m)} = \max_{0 \leq j < i} \texttt{t.(j)}$$

  dont on montre facilement que c'est un invariant de boucle. En sortie, on a donc $I(k + 1, m) \coloneqq \texttt{t.(m)} = \max_{0 \leq j \leq k} \texttt{t.(j)}$ donc m est bien l'indice d'un maximum de t entre les indices 0 et k.
- Pas de boucle ici, donc on va juste utiliser les propriétés précédentes. Si $i = k$, c'est direct, `t.(k)` est le maximum.

  Sinon, c'est que le maximum est à l'indice i. Suite au premier appel à `retourner` on place ce maximum à l'indice 0 et suite au second on le place à l'indice k, ce qui permet ainsi d'assurer la propriété voulue.

On en déduit alors un algorithme de tri, dit *tri du crêpier* car il consiste à faire des retournements successifs comme une pile de crêpes.

```
let rumman t =
  for k = Array.length t - 1 downto 0 do
    placer t k
  done
```

Question 13 Donner l'état du tableau à chaque itération dans l'appel à `rumman t` où `t = [| 0; 5; 7; 1; 2 |]`.

Corrigé

- `[| 0; 5; 7; 1; 2 |]`
- $k = 4$: → `[| 7; 5; 0; 1; 2 |]` → `[| 2; 1; 0; 5; 7 |]`
- $k = 3$: pas de changement
- $k = 2$: → `[| 2; 1; 0; 5; 7 |]` → `[| 0; 1; 2; 5; 7 |]`
- $k = 1$: pas de changement
- $k = 0$: pas de changement

Question 14 En utilisant un invariant de boucle, montrer que la fonction `rumman` trie le tableau passé en argument dans l'ordre croissant.

Corrigé

On commence par remarquer que les seules modifications du tableau sont effectuées par `retourner` et que cette modification ne fait que permuter des éléments. Donc, à l'issue de `rumman` les valeurs ont juste été permutées.

On montre alors que la propriété suivante est un invariant de boucle :

$$I(k) \coloneqq t[k+1 : n-1] \text{ est trié et } \forall i \in [\![0, k]\!], \forall j \in [\![k+1, n-1]\!], \texttt{t.(i)} \leq \texttt{t.(j)}$$

où $|\, t \,| = n$, en raisonnant sur l'indice comme auparavant : on le décrémente en fin d'itération.

Note : on aurait pu simplifier l'invariant en indiquant que `t.(k+1)` est plus grand que les valeurs précédentes, mais l'indice $k+1$ est invalide au départ.

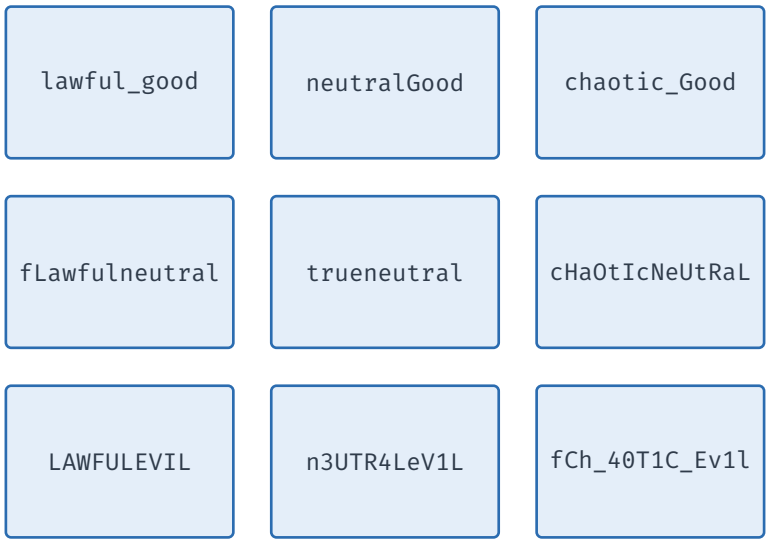
- **Init** : au départ $k = n-1$ et donc $t[k+1 : n-1] = \emptyset$ et la seconde partie de la propriété est triviale.
- **Her** : si la propriété est vérifiée au début d'une itération, on sait que $t[k+1 : n-1]$ est trié et que suite à l'appel à `placer t k` on va avoir à l'indice k le maximum des valeurs aux indices de 0 à k qui sont elles-mêmes plus petites que `t.(k+1)` par hypothèse. Donc on a bien la propriété en fin d'itération.

En sortie de boucle, on a donc t trié.

Question 15 Donner, en les justifiant, les complexités temporelles dans le pire des cas des fonctions `retourner`, `placer`, `indice_max` et `rumman`.

Corrigé

- `retourner t i j` effectue des appels en enlevant 2 à $j - i$ à chaque appel, on a donc $O(j-i)$ appels, chacun en temps constant.
- `indice_max t k` est directement en $O(k)$ par la boucle.
- `placer` est en (indice, retourne, retourne) $O(k) + O(i) + O(k) = O(k)$ car $i \leq k$.
- `rumman` est en $\sum_{k=0}^{n-1} O(k) \leq C \sum_{k=0}^{n-1} k = C\frac{n(n-1)}{2}$ donc en $O(n^2)$.


lawful_good
neutralGood
chaotic_Good
fLawfulneutral
trueneutral
cHaOtIcNeUtRaL
LAWFULEVIL
n3UTR4LeV1L
fCh_40T1C_Ev1l

DEVOIR 56

DS 3 - Partie pratique (OCaml)

56.1 DS 3 — Partie pratique (OCaml)

MP2I, année 2021-2022. Devoir surveillé 3 du 16 décembre 2021, partie pratique sur machine (exercices Learn OCaml « DS3 - Récursif » et « DS3 - Impératif »). La partie écrite est sur une page séparée.

56.1.1 Récursif

Attention dans toute cette partie, vous n'avez pas le droit d'utiliser de références ni d'utiliser les modules `List` ou `Array`.

Les fonctions avec  en indication vous garantissent, si vous les faites toutes, une note de 15.

Question 1  Écrire une fonction `fact` calculant la factorielle de l'entier naturel passé en argument.

Corrigé

Pas de difficulté pour `fact` mis-à-part le fait de se rappeler de la définition...

```
let rec fact n =
  if n = 0 then 1 else n * fact (n-1)
```

Question 2  Écrire une fonction `somme_liste` qui renvoie la somme des éléments de la liste d'entiers passée en paramètre.

Corrigé

Toutes les questions qui portent sur un parcours d'une liste vont être récursives et de la forme :

```
let rec parcours l = match l with
  | [ ] -> (* Cas de la liste vide *)
  | t::q -> (* On traite le cas où la liste commence par t
    et on va appeler parcours q *)
```

```
let rec somme_liste l =
  match l with
  | [] -> 0
  | t::q -> t + somme_liste q
```

Question 3  Écrire une fonction `renverse` qui renvoie la liste passée en argument dans l'ordre inverse. **Attention** vous ne pourrez pas la programmer directement en récursif, il faut passer par une fonction auxiliaire qui prend un accumulateur.

Attention vous aurez la moitié des points si votre fonction est en $O(n^2)$.

Corrigé

Ici, on tire parti du fait que le parcours d'une liste va naturellement **voir** les éléments de gauche à droite : le premier élément vu est la tête de la liste, ce sera le dernier élément de la liste renversée, donc si on la construit progressivement, c'est le premier élément qui sera mis devant la liste vide. Pour cela, on utilise donc une fonction auxiliaire qui va avoir un accumulateur et qui va rajouter chaque tête à l'accumulateur.

Si on le faisait avec des références ça correspondrait à ça :

```
let renverse l =
  let acc = ref [] in
  let lr = ref l in
  while !lr <> [] do
    let t::q = !lr in (* warning car possible [] meme si exclu *)
    acc := t :: !acc;
    lr := q
  done;
  !acc
```

Le code récursif est équivalent mais bien plus clair et élégant.

```
let renverse l =
  let rec aux l acc = match l with
    | [] -> acc
    | t::q -> aux q (t::acc)
  in aux l []
```

Question 4 Ici et dans les deux questions qui suivent, on va considérer des listes du type `('a * 'b) list` qui permettent de stocker des associations entre des éléments du type `'a` et des éléments du type `'b`. C'est une manière de réaliser un dictionnaire. Ainsi, la liste `[ ("a", 42); ("toto", 12) ]` va associer la valeur 42 à la clé `"a"` et la valeur 12 à la clé `"toto"`. On appelle cette structure de donnée une liste d'associations.

Écrire une fonction `ajoute_assoc : ('a * 'b) list -> 'a -> 'b -> ('a * 'b) list` qui prend une liste d'associations en entrée et a et b et rajoute une association entre a et b.

Attention cette fonction est **très simple**.

Corrigé

Question très simple mais qui demande de bien comprendre l'énoncé !

```
let ajoute_assoc l a b =
  (a,b) :: l
```

Question 5 Écrire une fonction `cherche_assoc : ('a * 'b) list -> 'a -> 'b` qui prend une liste d'associations et une valeur a en entrée et renvoie la valeur b qui lui est associée ou produit une erreur avec l'instruction `failwith "invalide"`.

Corrigé

On effectue une simple recherche dans la liste mais en tirant parti du fait que les éléments sont des couples. À noter ici le fait de faire un seul filtrage en filtrant aussi la forme de la tête.

```
let rec cherche_assoc l a =
  match l with
  | [] -> failwith "invalide"
  | (a',b)::q -> if a = a' then b else cherche_assoc q a
```

Question 6 Écrire une fonction `cherche_assoc_opt : ('a * 'b) list -> 'a -> 'b option` qui se comporte comme la fonction précédente hormis qu'elle utilise le type option pour renvoyer `None` en cas d'échec plutôt que de produire une erreur.

Corrigé

Comme précédemment mais avec des options.

```
let rec cherche_assoc_opt l a =
  match l with
  | [] -> None
  | (a',b)::q -> if a = a' then Some b else cherche_assoc_opt q a
```

Question 7 Écrire une fonction `zip : 'a list -> 'b list -> ('a * 'b) list` qui prend deux listes **de même longueur** et renvoie la liste des couples d'un élément de la première liste associé avec l'élément correspondant de la seconde.

Ainsi `zip [1;2;3] ['a';'b';'c']` doit renvoyer `[ (1,'a'); (2,'b'); (3,'c') ]`.

Corrigé

Encore une question assez simple mais qui déroute par l'énoncé. Prenez l'habitude de vous appuyer sur les exemples.

```
let rec zip l1 l2 =
  match l1, l2 with
  | t1::q1, t2::q2 -> (t1, t2)::zip q1 q2
  | _ -> []
```

Question 8 Écrire une fonction `range : int -> int -> int list` telle que l'appel à `range a b` renvoie la liste d'entiers `[ a;a+1;...;b ]`.

Corrigé

Ici, on calcule $[\![a, b]\!]$ en remarquant que $[\![a, b]\!] = \emptyset =$ `[ ]` quand $a > b$ et sinon, $[\![a, b]\!] = a :: [\![a + 1, b]\!]$.

```
let rec range a b =
  if a > b
  then []
  else a :: range (a+1) b
```

Question 9 *(Difficile)* Écrire une fonction `gen_tuples : 'a list -> int -> 'a list list` qui prend en entrée une liste `[a1;...;an]` et un entier p et renvoie tous les p-uplets de la liste sous forme de liste et dans l'ordre lexicographique correspondant à l'ordre de la liste. Ainsi pour `['a';'c';'b']` et 2 on va renvoyer

```
[ ['a';'a']; ['a';'c']; ['a';'b']; ['c';'a']; ['c';'c'];
  ['c';'b']; ['b';'a']; ['b';'c']; ['b';'b'] ]
```

Corrigé

```
let rec gen_tuples l n =
  if n = 0
  then [ [] ]
  else
    let tt = gen_tuples l (n-1) in
    let rec ajoute t l = match l with
      | [] -> []
```

```
      | ll::q -> (t::ll) :: ajoute t q
    in
    let rec aux l' =
      match l' with
      | [] -> []
      | t::q -> (ajoute t tt) @ aux q
    in aux l
```

Question 10 *(Difficile)* Écrire une fonction `gen_plists : 'a list -> int -> 'a list list` qui prend en entrée une liste `[a1;...;an]` et un entier p et renvoie toutes les p-listes de la liste sous forme de liste et dans l'ordre lexicographique correspondant à l'ordre de la liste.

La différence entre une p-liste et un p-uplet est qu'on ne peut avoir qu'une seule fois chacun des ai dans une p-liste. Attention, on distingue ici les positions dans la liste d'une même valeur. Pour `[1;2;1]` on a une 2-liste `[1;1]` mais `[2;2]` n'en est pas une.

Ainsi pour `['a';'c';'b']` et 2 on va renvoyer

```
[ ['a';'c']; ['a';'b']; ['c';'a']; ['c';'b']; ['b';'a']; ['b';'c'] ]
```

Corrigé

```
let rec gen_plists l n =
  if n = 0
  then [ [] ]
  else
    let rec retire t l =
      match l with
      | [] -> []
      | x::q -> if x = t then q else x :: retire t q
    in
    let rec ajoute t l = match l with
      | [] -> []
      | ll::q -> (t::ll) :: ajoute t q
    in
    let rec aux l' =
      match l' with
      | [] -> []
      | t::q -> (ajoute t (gen_plists (retire t l) (n-1))) @ aux q
    in aux l
```

56.1.2 Impératif

Attention dans toute cette partie, vous n'avez pas le droit d'utiliser de fonctions du module `Array` autres que `length`, `make`, `make_matrix`, `init`.

Les fonctions avec  en indication vous garantissent, si vous les faites toutes, une note de 15.

Question 1  Écrire une fonction `somme` qui prend en entrée un tableau d'entiers et renvoie la somme de ses éléments.

Corrigé

```
let somme a =
  let s = ref 0 in
  for i = 0 to Array.length a - 1 do
    s := a.(i) + !s
  done;
  !s
```

Question 2 Écrire une fonction `compte` qui prend en entrée un tableau d'éléments de type `'a` et une valeur `x` de type `'a` et renvoie le nombre de fois que cette valeur est présente dans le tableau.

Corrigé

```
let compte a x =
  let c = ref 0 in
  for i = 0 to Array.length a - 1 do
    if a.(i) = x
    then incr c
  done;
  !c
```

Question 3 Écrire une fonction `indice_minimum` qui prend en entrée un tableau d'éléments de type `'a` et renvoie le plus petit indice du minimum du tableau.

Cette fonction ne sera jamais appelée sur un tableau vide.

Corrigé

```
let indice_minimum a =
  let i = ref 0 in
  for j = 0 to Array.length a - 1 do
    if a.(j) < a.(!i)
    then i := j
  done;
  !i
```

Question 4 Écrire une fonction `array_concat` qui prend en entrée deux tableaux `t1`, `t2` et renvoie le tableau correspondant à mettre les éléments de `t1` suivis des éléments de `t2`.

Corrigé

Attention aux cas de base et à la valeur par défaut de `Array.make`.

```
let array_concat a1 a2 =
  if a1 = [||] then a2 else
  let n1 = Array.length a1 in
  let n2 = Array.length a2 in
  let a = Array.make (n1+n2) a1.(0) in
  for i = 0 to n1-1 do
    a.(i) <- a1.(i)
  done;
  for i = 0 to n2-1 do
    a.(n1+i) <- a2.(i)
  done;
  a
```

Question 5 Écrire une fonction `compte_inversions` qui prend en entrée un tableau `t` et compte le nombre de paires d'indices `i`, `j` tels que `i < j` et `t.(i) > t.(j)`.

Corrigé

```
let compte_inversions a =
  let c = ref 0 in
  let n = Array.length a in
  for i = 0 to n-1 do
    for j = i+1 to n-1 do
      if a.(i) > a.(j)
```

```
      then incr c
    done
  done;
  !c
```

Question 6  Écrire une fonction `cherche_motif` qui étant donnés deux tableaux `t` et `m` renvoie un booléen indiquant si `m` est un sous-tableau de `t`.

C'est la même notion que l'inclusion de motif dans une chaîne vue en TP.

Corrigé

```
let cherche_motif a am =
  let n = Array.length a in
  let m = Array.length am in
  let v = ref false in
  for i = 0 to n-m do
    let trouve = ref true in
    for k = 0 to m-1 do
      trouve := !trouve && a.(i+k) = am.(k)
    done;
    v := !trouve || !v
  done;
  !v
```

Question 7 *(Chronophage)* Écrire une fonction `dessine_sapin` qui étant donné un entier `p > 0` dessine un sapin ainsi :

```
    p        p      le motif fait 2*p+1
<------> <------>
        *
       * *
      *   *
     *     *<- dernier caractère
    *       *   on ne met pas les espaces
   *         *
  *           *
 *             *
**********
       |||         <- comme p>0 on a 2*p+1>=3 et
       |||               il y a toujours un tronc
```

Pour 5 on aura donc le sapin

```
     *
    * *
   *   *
  *     *
 *       *
***********
    |||
    |||
```

On utilisera les fonctions `print_char : char -> unit` et `print_newline ()`. On fera un saut de ligne après la dernière ligne.

Corrigé

```
let dessine_sapin p =
  for i = 0 to p-1 do
    for j = 1 to p-i do
```

```
    print_char ' '
  done;
  print_char '*';
  for j = 1 to 2*i-1 do
    print_char ' '
  done;
  if i > 0 then print_char '*';
  print_newline ()
done;
for i = 0 to 2*p do
  print_char '*'
done;
print_newline ();
for i = 1 to 2 do
  for j = 0 to p-2 do
    print_char ' '
  done;
  for _ = 1 to 3 do print_char '|' done;
  print_newline ()
done
```

Question 8 Écrire une fonction `tri` qui trie un tableau en place par la méthode de votre choix. Cette fonction ne renvoie rien vu qu'elle modifie le tableau passé en argument.

Corrigé

Tri par sélection :

```
let tri a =
  let echange a i j =
    let temp = a.(i) in
    a.(i) <- a.(j);
    a.(j) <- temp
  in
  for i = 0 to Array.length a - 2 do
    let j = ref i in
    for k = i+1 to Array.length a - 1 do
      if a.(k) < a.(!j)
      then j := k
    done;
    echange a i !j
  done
```

Question 9 *(Difficile)* Écrire une fonction `carre : bool array array -> int -> int -> int` telle que `carre m x y` renvoie la dimension du plus grand carré de `true` dans la matrice carrée de booléens `m` où le coin supérieur gauche du carré a pour coordonnées `(x, y)`.

Si on voit cette matrice comme une image en noir et blanc, il s'agit de trouver la taille du carré noir commençant au pixel `(x, y)`.

Cette fonction est toujours appelée avec des coordonnées pour lesquelles la valeur est `true`.

Corrigé

On agrandit le carré d'une couronne (colonne $y + r$ et ligne $x + r$) tant que celle-ci ne contient que des `true` et ne sort pas de la matrice.

```
let carre m x y =
  if m.(x).(y)
  then begin
    let r = ref 1 in
    let valid = ref true in
```

```
    let n = Array.length m in
    while x + !r < n && y + !r < n && !valid do
      for xi = x to x + !r do
        valid := !valid && m.(xi).(y + !r)
      done;
      for yi = y to y + !r do
        valid := !valid && m.(x + !r).(yi)
      done;
      if !valid then incr r
    done;
    !r
  end else 0
```

Question 10 *(Nécessite la précédente)* Écrire une fonction `plus_grand_carre` qui renvoie la dimension du plus grand carré dans la matrice donnée en entrée selon la définition précédente.

Corrigé

```
let plus_grand_carre m =
  let n = Array.length m in
  let c = ref 0 in
  for x = 0 to n-1 do
    for y = 0 to n-1 do
      let cxy = carre m x y in
      if cxy > !c
      then c := cxy
    done
  done;
  !c
```

DEVOIR 57

DS 4 - Listes chaînées, pavages et backtracking

57.1 DS 4 — Listes chaînées, pavages et backtracking

MP2I, année scolaire 2021-2022 — devoir surveillé du 1er février 2022 — durée totale 3h.

Consignes

Tout document interdit. Tout dispositif électronique interdit.

On vous demande d'écrire lisiblement et de bien présenter. Les programmes en particulier doivent être bien indentés et lisibles. Utilisez de préférence plusieurs couleurs. **Commentez** votre code dès que ce qu'il fait n'est pas évident.

Avant chaque fonction compliquée à écrire, ou quand vous n'y arrivez pas, indiquez avec des phrases ce que vous voulez faire. Cela permet de vous donner éventuellement des points si l'idée était bonne ou si vous y êtes presque.

57.1.1 A link to a past TP

On va reprendre le type des listes chaînées en `C` :

```c
struct maillon {
    int valeur;
    struct maillon *suivant;
};

typedef struct maillon maillon;

struct liste {
    maillon *debut;
    maillon *fin;
};

typedef struct liste liste;
```

- Dans un maillon, soit `suivant` pointe vers le maillon suivant de la chaîne ou vaut `NULL`.
- La liste vide aura la valeur `NULL` pour `debut` et `fin`.
- Dans une liste non vide, `debut` pointe sur le premier maillon et `fin` sur le dernier.

On suppose définies des fonctions

```c
// cree un maillon
maillon *creer_maillon(int valeur, maillon *suivant);
// supprime un maillon
void suppr_maillon(maillon *);
// cree une liste vide
liste *creer_liste(void);
// supprime une liste
void suppr_liste(liste *);
```

Question 1 Écrire une fonction de prototype

```
int longueur(liste *);
```

qui renvoie la longueur d'une liste.

Corrigé

```
int longueur(liste *l)
{
    int n = 0;
    maillon *m = l->debut;
    while(m != NULL)
    {
        n++;
        m = m->suivant;
    }
    return n;
}
```

Question 2 Écrire une fonction de prototype

```
int somme(liste *);
```

qui renvoie la somme des valeurs des éléments d'une liste.

Corrigé

```
int somme(liste *l)
{
    int s = 0;
    maillon *m = l->debut;
    while(m != NULL)
    {
        s = s + m->valeur;
        m = m->suivant;
    }
    return s;
}
```

Question 3 Écrire une fonction de prototype

```
void liste_ajout_en_tete(liste *l, int valeur);
```

qui ajoute un maillon en tête de la liste passée en paramètre en $O(1)$.

On fera attention au cas où la liste est vide.

Corrigé

```
void liste_ajout_en_tete(liste *l, int valeur)
{
    l->debut = creer_maillon(valeur, l->debut);
    if (l->fin == NULL)
        l->fin = l->debut;
}
```

Question 4 Écrire une fonction de prototype

```
maillon *liste_cherche_maillon(liste *l, int valeur);
```

qui renvoie un pointeur sur le premier maillon de valeur `valeur` ou renvoie `NULL` s'il n'y en a pas.

Corrigé

```
maillon *liste_cherche_maillon(liste *l, int valeur)
{
    maillon *m = l->debut;
    while(m != NULL && m->valeur != valeur)
    {
        m = m->suivant;
    }
    return m;
}
```

Question 5 Écrire une fonction de prototype

```
void liste_ajout_maillon_apres(liste *l, maillon *m, int valeur);
```

qui insère en $O(1)$ un maillon après le maillon `m`. On fera en sorte que l'insertion soit valide même si `m` est le dernier maillon de la liste.

Corrigé

```
void liste_ajout_maillon_apres(liste *l, maillon *m, int valeur)
{
    m->suivant = creer_maillon(valeur, m->suivant);
    if (l->fin == m)
        l->fin = m->suivant;
}
```

Question 6 Expliquer comment cette structure permet d'implémenter une pile et donner les différentes complexités des opérations de la structure liste.

Corrigé

L'ajout et la suppression en tête sont en $O(1)$.

57.1.2 Domino's Tiling

On cherche à générer des pavages d'un rectangle de taille $w \times h$ par des dominos, c'est-à-dire des tuiles de tailles 2×1 et sans recouvrement. Un exemple d'un tel pavage est donné ici :

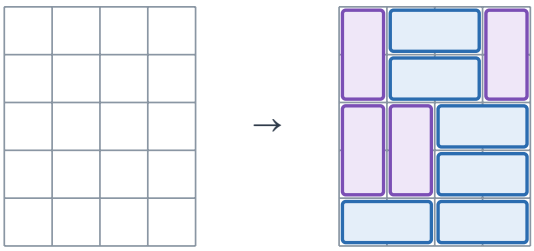


Figure 57.1 — Un pavage d'un rectangle 4×5 par des dominos horizontaux (bleu) et verticaux (violet).

Question 7 Donner une condition nécessaire et suffisante sur le couple (w, h) pour qu'un tel pavage existe.

Attention : CNS veut dire qu'il faut donner une condition, montrer qu'elle est vérifiée quand il y a un pavage et montrer que si la condition est vérifiée, alors un pavage existe.

Corrigé

Un tel pavage existe ssi wh est pair.

En effet, si ce n'est pas le cas, le nombre de cases couvertes par des dominos est forcément pair, donc on ne peut pas paver. Si c'est le cas, soit w, soit h est pair, et il suffit de placer des dominos régulièrement horizontalement ou verticalement pour paver.

On suppose à partir de maintenant que (w, h) vérifie cette CNS et qu'ainsi il existe un pavage.

On va représenter une telle grille par des cases de trois types :

- `E` si la case est à remplir (E comme *empty*)
- `H` si la case appartient à un domino horizontal
- `V` si la case appartient à un domino vertical

```
type case = E | H | V
```

Ainsi, on a perdu la représentation en dominos en les *coupant en deux*. Si on considère le pavage de l'exemple précédent, il correspond à la matrice OCaml :

```
[|
  [| V; H; H; V |];
  [| V; H; H; V |];
  [| V; V; H; H |];
  [| V; V; H; H |];
  [| H; H; H; H |];
|]
```

Cependant, si on considère la première valeur `V` elle ne peut venir que d'un seul domino vertical commençant ici, on peut ainsi procéder à l'envers en l'enlevant :

```
[|
  [| E; H; H; V |];
  [| E; H; H; V |];
  [| V; V; H; H |];
  [| V; V; H; H |];
  [| H; H; H; H |];
|]
```

Et on continue ensuite avec la première case différente de `E` en allant de haut en bas et de gauche à droite.

On identifie un domino à un couple de coordonnées $((x_1, y_1), (x_2, y_2))$ indiquant les cases occupées (l'ordre n'est pas important ici).

```
type domino = (int * int) * (int * int)
```

Question 8 Écrire une fonction de signature

```
val extrait_pavage : case array array -> domino list
```

qui extrait la liste des dominos d'une matrice de pavage éventuellement partielle. On pourra modifier la matrice passée en entrée si c'est plus simple pour vous (indice : ce sera le cas).

Corrigé

```
let rec extrait_pavage a =
  let w, h = Array.length a.(0), Array.length a in
  let rec non_vide i j =
    if a.(j).(i) <> E
    then Some (i, j)
    else if i <> w-1
    then non_vide (i+1) j
    else if j <> h-1
```

```
    then non_vide 0 (j+1)
    else None
  in
  match non_vide 0 0 with
  | Some (i,j) ->
    if a.(j).(i) = H
    then begin
      a.(j).(i) <- E;
      a.(j).(i+1) <- E;
      ( (i,j), (i+1,j) ) :: extrait_pavage a
    end else begin
      a.(j).(i) <- E;
      a.(j+1).(i) <- E;
      ( (i,j), (i,j+1) ) :: extrait_pavage a
    end
  | None -> []
```

Pour déterminer un pavage, on va utiliser un algorithme de backtracking. Les mouvements sont des dominos.

Question 9 Écrire une fonction de signature

```
val mouvements : case array array -> domino list
```

qui étant donnée une grille renvoie tous les dominos qu'on peut placer.

Attention : même si on a le même type, le problème est différent de la fonction précédente.

Corrigé

```
let mouvements a =
  let m = ref [] in
  let w, h = Array.length a.(0), Array.length a in
  for i = 0 to w-1 do
    for j = 0 to h-1 do
      if a.(j).(i) = E
      then begin
        if i < w-1 && a.(j).(i+1) = E then m := ((i,j),(i+1,j))::!m;
        if j < h-1 && a.(j+1).(i) = E then m := ((i,j),(i,j+1))::!m
      end
    done
  done;
  !m
```

Question 10 Écrire des fonctions de signature

```
val place : case array array -> domino -> unit
val enleve : case array array -> domino -> unit
```

qui place et enlève respectivement un domino.

Corrigé

```
let place a ((i,j),(k,l)) =
  let c = if i = k then V else H in
  a.(j).(i) <- c;
  a.(l).(k) <- c

let enleve a ((i,j),(k,l)) =
  a.(j).(i) <- E;
  a.(l).(k) <- E
```

Question 11 Écrire une fonction de signature

```
val complet : case array array -> bool
```

qui indique si une grille est complètement pavée.

Corrigé

```
let complet a =
  let w, h = Array.length a.(0), Array.length a in
  let b = ref true in
  for j = 0 to h-1 do
    for i = 0 to w-1 do
      b := a.(j).(i) <> E && !b
    done
  done;
  !b
```

Question 12 Écrire une fonction récursive de signature

```
val enumere : case array array -> unit
```

qui énumère les pavages de dominos de la grille passée en paramètre et lève l'exception `Solution l` quand la grille correspond à un pavage et `l` est la suite de dominos correspondante telle qu'obtenue avec `extrait_pavage`.

L'exception est définie par :

```
exception Solution of domino list
```

Corrigé

```
exception Solution of domino list

let enumere a =
  let rec aux () =
    if complet a
    then raise (Solution (extrait_pavage (Array.map Array.copy a)))
    else List.iter
        (fun m ->
           place a m;
           aux ();
           enleve a m) (mouvements a)
  in aux ()
```

Question 13 Écrire une fonction de signature :

```
val pavage : int -> int -> domino list
```

telle que `pavage w h` renvoie une liste de dominos correspondant à un pavage d'une grille de dimension $w \times h$.

Corrigé

```
let pavage w h =
  try
    enumere (Array.make_matrix h w E);
    failwith "pas de pavage"
  with Solution l -> l
```

Question 14 Reprendre la fonction `enumere` pour en déduire une fonction de signature

```
val enumere_tous : case array array -> domino list list
```

qui renvoie la liste de tous les pavages depuis une grille donnée.

En déduire ainsi une fonction

```
val pavages : int -> int -> domino list list
```

qui renvoie tous les pavages pour une dimension donnée de grille.

Corrigé

Pour ne pas obtenir plusieurs fois le même pavage (une fois pour chaque ordre de pose des dominos), on ne considère que les dominos qui couvrent la première case vide dans l'ordre de lecture : cette case doit être couverte et ne peut l'être que par un domino dont elle est la première case.

```
let premiere_vide a =
  let w = Array.length a.(0) in
  let rec aux i j =
    if a.(j).(i) = E then (i, j)
    else if i < w-1 then aux (i+1) j
    else aux 0 (j+1)
  in aux 0 0

let enumere_tous a =
  let rec aux l =
    if complet a
    then [ l ]
    else begin
      let c = premiere_vide a in
      let acc = ref [] in
      List.iter
        (fun m ->
           if fst m = c then begin
             place a m;
             acc := aux (m::l) @ !acc;
             enleve a m
           end) (mouvements a);
      !acc
    end
  in aux []

let pavages w h = enumere_tous (Array.make_matrix h w E)
```

On trouve ainsi 36 pavages pour une grille 4×4 et 95 pour une grille 4×5.

57.1.3 Matoucheespaceestcassée

> *Et ma blême araignée, ogre illogique et las Aimable, aime à régner, au gris logis qu'elle a.*
>
> **Victor Hugo**

On s'intéresse dans cette partie au problème du placement des espaces dans une phrase qui n'en a pas. Par exemple la phrase suivante en anglais

bothearthandsaturnspin pourrait se décomposer en **both earth and saturn spin** ou **bot heart hands at urns pin**.

On va considérer des `char array` ici pour les chaînes, mais tout marche de la même manière avec des `string`.

On considère un dictionnaire, ou *wordlist*, défini globalement :

```
val dico : char array list
```

Notation : `t[i..j]` est le sous-tableau de `t` constitué des éléments d'indice `i` à `j`.

Question 15 Écrire une fonction de signature :

```
val est_mot : char array -> int -> int -> bool
```

telle que l'appel à `est_mot t i j` renvoie `true` lorsque `t[i..j]` est dans le dictionnaire.

Corrigé

```
let est_mot c i j =
  let m = Array.init (j-i+1) (fun k -> c.(i+k)) in
  List.mem m dico
```

On va résoudre le problème de l'ajout d'espaces par backtracking avec une version plus simple : pour chaque chaîne, on se contente de décider s'il est possible ou non de rajouter des espaces pour qu'elle ne soit composée que de mots dans le dictionnaire. On dit qu'un tableau est **séparable** quand on peut ajouter ces espaces.

On a donc `t` **séparable** s'il existe i tel que `t[0..i-1]` est un mot et `t[i..n-1]` est **séparable** où `|t| = n` est la longueur de t.

Question 16 Écrire une fonction récursive de signature :

```
val separable : char array -> int -> bool
```

telle que `separable t i` indique si `t[i..n-1]` est séparable où $n = |\,t\,|$.

Corrigé

```
let rec separable t i =
  if i >= Array.length t
  then true
  else begin
    let separe = ref false in
    let j = ref i in
    while not !separe && !j < Array.length t do
      separe := est_mot t i !j && separable t (!j+1);
      incr j
    done;
    !separe
  end
```

Question 17 On note T_p le nombre d'appels à `est_mot` dans l'exécution de `separable t i` quand $p = |\,t\,| - i$ est le nombre d'éléments du sous-tableau considéré.

1. Montrer que $T_p \leq \sum_{i=0}^{p-1} T_i + O(p)$.
2. Quitte à se placer dans un pire cas, on peut supposer qu'il existe une constante A telle que $T_p = \sum_{i=0}^{p-1} T_i + Ap$.

   Montrer que pour $p > 0$, $T_p = 2T_{p-1} + A$.
3. En déduire que $T_p = O(2^p)$.

Corrigé

1. On considère que T_p correspond à l'appel `separable t k`. Le pire cas correspond au cas où tous les préfixes de `t[k..n-1]` sont des mots. Ainsi, pour chaque $j \in [\![k, n-1]\!]$ on a `t[k..j]` qui est un mot et on effectue un appel récursif à `separable t (j+1)` qui va effectuer T_{n-j-1} appels à `est_mot`. On a donc

$$T_p \leq (n-k) + \sum_{j=k}^{n-1} T_{n-j-1} \leq O(p) + \sum_{i=0}^{p-1} T_i$$

   en remarquant que $p = n - k$ et en posant $i = n - j - 1$.

2. On écrit la récurrence pour $p-1$:

$$T_{p-1} = \sum_{i=0}^{p-2} T_i + A(p-1)$$

et on soustrait avec la récurrence pour p :

$$T_p - T_{p-1} = T_{p-1} + A$$

3. On a $T_0 = 0, T_1 = A, T_2 = 3A, T_3 = 7A, \ldots$. On conjecture donc rapidement que $T_p = (2^p - 1)A$ et on le valide par récurrence.
   - Initialisation : $T_0 = 0 = (2^0 - 1)A$.
   - Hérédité : si $T_p = (2^p - 1)A$ on a $T_{p+1} = 2T_p + A = 2^{p+1}A - 2A + A = (2^{p+1} - 1)A$.

   Ainsi, $T_p = O(2^p)$.

DEVOIR 58

DS 6 - Matrices de Monge et problème du monnayeur

58.1 DS 6 — Matrices de Monge et problème du monnayeur

MP2I, année scolaire 2021-2022 — durée 4h, tout document ou dispositif électronique interdit.

Partie « Le problème du monnayeur » d'après l'épreuve d'informatique du concours Centrale-Supélec 2002.

58.1.1 Matrices de Monge

Dans cette partie, on note $A[i, j]$ le coefficient situé sur la i-ème ligne et j-ème colonne d'une matrice.

Une matrice $A \in \mathscr{M}_{mn}(\mathbb{R})$ est une **matrice de Monge** lorsque $\forall i, j, k$ et l tels que $1 \leq i < k \leq m$ et $1 \leq j < l \leq n$, on a

$$A[i, j] + A[k, l] \leq A[i, l] + A[k, j]$$

Autrement dit, étant donné deux lignes et deux colonnes dans la matrice, leurs intersections sont les quatre sommets d'un rectangle dont on demande que la somme des éléments en haut à gauche et en bas à droite soit inférieure ou égale à la somme des éléments en haut à droite et en bas à gauche.

Par exemple, la matrice suivante est une matrice de Monge :

10	17	13	28	23
17	22	16	29	23
24	28	22	34	24
11	13	6	17	7
45	44	32	37	23
36	33	19	21	6
75	66	51	53	34

Figure 58.1 — Une matrice de Monge et un rectangle : $17 + 37 \leq 44 + 28$.

Un exemple de rectangle est présenté et on voit bien que $17 + 37 \leq 44 + 28$. Cette propriété est vraie pour tous les sous-rectangles.

Question 1 Prouver qu'une matrice est de Monge si et seulement si

$$\forall i \in [\![1, m-1]\!], \forall j \in [\![1, n-1]\!], A[i, j] + A[i+1, j+1] \leq A[i, j+1] + A[i+1, j]$$

Corrigé

Il y a une implication directe $\Rightarrow$ par restriction.

Dans l'autre sens, on montre d'abord que :

$$\forall i, k \in [\![1, m]\!], \forall j \in [\![1, n-1]\!], i < k \Rightarrow A[i, j] + A[k, j+1] \leq A[i, j+1] + A[k, j]$$

Pour cela, on procède par récurrence sur $k-i$. L'hypothèse est exactement la propriété pour $k-i=1$. Si la propriété est vraie au rang r, soit $k=i+r+1$. On a, par hypothèse,

$$A[i,j]+A[k-1,j+1]\leq A[i,j+1]+A[k-1,j]$$

et

$$A[k-1,j]+A[k,j+1]\leq A[k-1,j+1]+A[k,j]$$

Donc,

$$A[i,j]+A[k-1,j+1]+A[k-1,j]+A[k,j+1]\leq A[i,j+1]+A[k-1,j]+A[k-1,j+1]+A[k,j]$$

et il suffit de simplifier pour avoir la propriété voulue.

On montre de même, par récurrence sur $l-j$, que la matrice est de Monge.

Question 2 La matrice suivante n'est pas de Monge. Changez un seul de ses éléments pour faire en sorte qu'elle le soit. On utilisera la question précédente.

$$\begin{pmatrix}37&23&22&32\\21&6&7&10\\53&34&30&31\\32&13&9&6\\43&21&15&8\end{pmatrix}$$

Corrigé

On remarque que seul le carré

$$\begin{pmatrix}23&22\\6&7\end{pmatrix}$$

est problématique car $23+7=30>22+6=28$. Pour résoudre ce problème, le plus simple semble être de changer la valeur du coefficient valant 22 par une valeur $v\geq 24$.

Cependant, on doit avoir le carré suivant

$$\begin{pmatrix}v&32\\7&10\end{pmatrix}$$

qui satisfait la condition, donc $10+v\leq 39\Leftrightarrow v\leq 29$.

On peut ainsi prendre $v=24$.

Question 3 Soit $f(i)$ l'indice de la colonne contenant l'élément minimum le plus à gauche de la ligne i. Prouver que f est croissante pour une matrice de Monge. C'est-à-dire que $f(1)\leq f(2)\leq\ldots\leq f(m)$.

Corrigé

Si, par l'absurde, on a $i<j$ et $f(i)>f(j)$, alors pour le carré correspondant, comme la matrice est de Monge on devrait avoir :

$$A[i,f(j)]+A[j,f(i)]\leq A[i,f(i)]+A[j,f(j)]$$

Or, par minimalité, $A[i,f(i)]\leq A[i,f(j)]$ et $A[j,f(j)]\leq A[j,f(i)]$. On a donc $A[i,f(j)]+A[j,f(i)]=A[i,f(i)]+A[j,f(j)]$. Si $A[i,f(j)]>A[i,f(i)]$ alors $A[j,f(i)]<A[j,f(j)]$ pour compenser ce qui est contradictoire. Ainsi, $A[j,f(j)]=A[j,f(i)]$ et donc $A[i,f(j)]=A[i,f(i)]$ or l'élément sur la colonne $f(j)$ est plus à gauche que celui sur la colonne $f(i)$ qui était censé être le plus à gauche des minimaux. Contradiction.

Question 4 On présente un algorithme Diviser-pour-régner afin de déterminer le minimum le plus à gauche de chaque ligne dans une matrice de Monge A :

On construit une sous-matrice A' de A contenant les lignes de numéro pair. On détermine récursivement le minimum le plus à gauche de chaque ligne de A'. Ensuite on détermine le minimum le plus à gauche de chaque ligne de numéro impair de A.

Expliquer comment déterminer le minimum le plus à gauche des lignes de numéro impair de A, sachant qu'on connaît ces minimums pour les lignes de numéro pair, en temps $O(m+n)$.

Corrigé

Il suffit de chercher le minimum pour la ligne $2p+1$ entre les colonnes $f(2p)$ et $f(2p+2)$ par croissance de f. Cette recherche effectue $f(2p+2) - f(2p) + 1$ opérations élémentaires. On a donc en tout $\sum f(2p+2) - f(2p) + 1 = O(m+n)$ (au maximum on fait une comparaison pour chaque colonne autre que la première, donc $O(n)$, et on fait une affectation pour chaque écart, donc $m/2 = O(m)$).

Question 5 Écrire la récurrence régissant le temps de calcul de l'algorithme précédent. Montrer qu'on peut la résoudre et obtenir une complexité globale en $O(m + n\log m)$.

Corrigé

À n fixé, notons T_m le temps pour une matrice (m,n), on a T croissante et la récurrence

$$T_{2m} = T_m + O(m+n)$$

Donc, il existe une constante M telle que $T_{2m} \leq T_m + M(m+n)$.

On a $T_{2^k} \leq M(2^k + nk)$ par récurrence rapide :

$$T_{2^{k+1}} \leq T_{2^k} + M(2^k + n) \leq M2^{k+1} + nM(k+1)$$

Ainsi, par croissance et encadrement, on conclut que $T_m = O(m + n\log m)$.

58.1.2 Le problème du monnayeur

Notations

- $|\,E\,|$ désigne le cardinal de l'ensemble fini E.
- $\lfloor x \rfloor$ désigne la partie entière inférieure du réel x, et $\lceil x \rceil$ sa partie entière supérieure : ce sont les relatifs définis respectivement par $\lfloor x \rfloor \leq x < \lfloor x \rfloor + 1$ et $\lceil x \rceil - 1 < x \leq \lceil x \rceil$.
- $[\![p,q]\!]$ désigne l'intervalle discret délimité par les naturels p et q : $[\![p,q]\!] = \{k \in \mathbb{N} \mid p \leq k \leq q\}$.

Présentation du problème

Nous nous intéressons au problème suivant, rencontré lors de la conception d'un monnayeur : comment rendre la monnaie, en utilisant le plus petit nombre possible de pièces ?

Formalisons ce problème : soit $(c_i)_{1\leq i\leq m}$ une liste d'entiers non nuls, vérifiant

$$c_1 > c_2 > \ldots > c_{m-1} > c_m = 1$$

Nous utiliserons le terme système pour désigner une telle liste. Les c_i sont les valeurs faciales des pièces en service. Par exemple, si nous omettons les pièces de valeur inférieure à 1 euro, le système utilisé en France est $(200, 100, 50, 20, 10, 5, 2, 1)$.

Pour chaque $i \in [\![1,m]\!]$, nous disposons d'une quantité illimitée de pièces de valeur c_i. Soit x un entier (le montant à rendre). Une représentation de x dans le système c est un m-uplet $(k_i)_{1\leq i\leq m}$ de naturels vérifiant

$$x = \sum_{1\leq i\leq m} k_i c_i$$

k_i est donc le nombre de pièces de valeur c_i que l'on doit rendre.

Pour épargner les poches des clients, nous souhaitons minimiser le poids de cette représentation, c'est-à-dire le nombre $\| k \| = \sum_{1 \leq i \leq m} k_i$ de pièces rendues.

Nous utiliserons en Caml des listes d'entiers pour représenter aussi bien un système (liste de valeurs de pièces), qu'une représentation d'un montant dans ce système. Par exemple, la liste $[4; 1; 3]$ est une représentation de 30 dans le système $(6, 3, 1)$.

Question 6 Rédigez une fonction de signature

```
est_un_systeme : int list -> bool
```

spécifiée comme suit : `est_un_systeme c` indique si la liste `c` est bien un système.

Par exemple, `est_un_systeme [5 ; 2 ; 1]` rendra la valeur `true`, tandis que `est_un_systeme [5 ; 7 ; 1]` rendra la valeur `false`, tout comme le fera `est_un_systeme [7 ; 5 ; 2]`.

✍ **Corrigé**

```
let rec est_un_systeme c =
  match c with
  | [] -> false
  | [x] -> x = 1
  | a :: b :: q -> a > b && est_un_systeme (b :: q)
```

Représentations de poids minimal

Soient $c = (c_i)_{1 \leq i \leq m}$ un système et $x \geq 0$. Nous notons $M_c(x)$ (ou même $M(x)$ lorsqu'aucune ambiguïté n'est à craindre) le plus petit nombre de pièces nécessaires pour représenter x dans le système c :

$$M(x) = \min \left\{ \| k \| \mid k \in \mathbb{N}^m \text{ et } \sum_{i=1}^{m} k_i c_i = x \right\}$$

Nous nous intéresserons aux représentations de poids minimal de x : ce sont les représentations k telles que $\| k \| = M(x)$. Pour alléger la rédaction, nous parlerons simplement de représentations minimales.

Question 7 Prouvez l'encadrement $\lceil x/c_1 \rceil \leq M(x) \leq x$.

Question 8 Exhibez un système c et un nombre $x > 0$ possédant plusieurs représentations minimales dans le système c.

Question 9 Soient c un système et $x > 0$. Montrez que $M(x) \leq 1 + M(x - c_j)$ pour tout indice j tel que $c_j \leq x$.

Question 10 Soient c un système, $x > 0$ et j tel que $c_j \leq x$. Montrez que $M(x) = 1 + M(x - c_j)$ si et seulement s'il existe une représentation minimale de x, faisant intervenir c_j (c'est-à-dire telle que $k_j > 0$).

Question 11 Soit $x > 1$; notons s le plus petit indice i tel que $c_i \leq x$. Nous avons donc $c_s \leq x$; et, soit $s = 1$, soit $s > 1$ et $c_{s-1} > x$. Justifiez l'égalité

$$M(x) = 1 + \min_{s \leq i \leq m} M(x - c_i)$$

Question 12 Soit $x > 0$. Montrez que l'on peut construire la liste des valeurs de $M(y)$ pour $y \leq x$, pour un coût $\mathcal{O}(mx)$.

Question 13 Rédigez en Caml une fonction de signature

```
poids_minimaux : int -> int list -> int list
```

spécifiée comme suit : `poids_minimaux x c` construit la liste des valeurs de $M_c(y)$, pour $1 \leq y \leq x$. Par exemple, `poids_minimaux 5 [5 ; 2 ; 1]` rendra la liste `[1 ; 1 ; 2 ; 2 ; 1]`. Vous utiliserez la formule de la question 11.

Si besoin, vous avez le droit **uniquement pour cette question** de passer des listes aux tableaux avec

```
Array.of_list : 'a list -> 'a array
Array.to_list : 'a array -> 'a list
```

58.1.2.3.1 L'algorithme glouton

L'algorithme glouton consiste à choisir le plus grand $c \leq x$, puis à rendre le montant $x - c$. Par exemple, avec le système $(10, 5, 2, 1)$ l'algorithme glouton décomposera le montant 27 en $27 = 10 + 10 + 5 + 2$.

Avec le formalisme proposé, $m = 4, c_1 = 10, c_2 = 5, c_3 = 2, c_4 = 1$; la solution fournie par l'algorithme glouton est le quadruplet $(k_1 = 2, k_2 = 1, k_3 = 1, k_4 = 0)$.

Le fonctionnement de l'algorithme glouton peut être accéléré avec la remarque suivante : notant $q = \lfloor x/c_1 \rfloor$, cet algorithme rend q pièces de valeur c_1, puis rend le montant $x - qc_1$ en utilisant le système $(c_i)_{2 \leq i \leq m}$.

Question 14 Rédigez une fonction de signature

```
glouton : int -> int list -> int list
```

spécifiée comme suit : `glouton x c` construit la représentation de x dans le système c, en appliquant l'algorithme glouton.

Par exemple, `glouton 13 [5 ; 2 ; 1]` rendra la liste `[2 ; 1 ; 1]`.

✍ **Corrigé**

On applique directement la remarque précédente :

```
let rec glouton x c =
  match c with
  | [] -> []
  | ci :: q -> (x / ci) :: glouton (x mod ci) q
```

Soient $c = (c_i)_{1 \leq i \leq m}$ un système et $x \geq 0$. Nous noterons $\Gamma_c(x)$ la représentation gloutonne de x dans le système c, et $G_c(x)$ (ou même $G(x)$ lorsqu'aucune ambiguïté n'est à craindre) le nombre de pièces utilisées par l'algorithme glouton : $G_c(x) = \| \Gamma_c(x) \|$.

Nous dirons qu'un système $c = (c_i)_{1 \leq i \leq m}$ est canonique si l'algorithme glouton donne toujours une représentation minimale ; on a donc $M(x) = G(x)$ quel que soit x.

Question 15 Montrez que tout système $c = (c_1, c_2)$ de deux valeurs est canonique.

Question 16 Exhibez un système $c = (c_1, c_2, c_3)$ non canonique.

Question 17 Soient q et n deux entiers ≥ 2. Montrer que le système $(q^n, q^{n-1}, \ldots, q, 1)$ est canonique.

Question 18 Montrer que le système *Euro* $(200, 100, 50, 20, 10, 5, 2, 1)$ est canonique.

On pourra montrer que dans une représentation minimale de $x : k = (k_1, \ldots, k_8)$, on a $k_8 \leq 1, 2k_7 + k_8 \leq 4, 5k_6 + 2k_7 + k_8 \leq 9, \ldots$.

Question 19 Avant la réforme de 1971 (introduisant un système décimal), le Royaume-Uni utilisait le système $(30, 24, 12, 6, 3, 1)$. Montrez que ce système n'est pas canonique.

58.1.2.3.2 L'algorithme de Kozen et Zaks

Nous nous proposons de décrire un algorithme efficace permettant de déterminer si un système c est canonique. Nous dirons qu'un entier x est un contre-exemple pour c si $M(x) < G(x)$. Un système canonique n'admet donc aucun contre-exemple.

Dans la suite, $m \geq 3$.

Question 20 Soit c un système non canonique. Montrez qu'il existe un contre-exemple x vérifiant l'encadrement

$$c_{m-2} + 1 < x < c_1 + c_2$$

Nous allons montrer que l'intervalle de recherche des contre-exemples ne peut pas être diminué.

Question 21 Soit $q \geq 3$. Montrez que le système $(q+1, q, 1)$ n'est pas canonique. Quel est le plus petit contre-exemple pour ce système ?

Question 22 Soit $q \geq 3$. Déterminez $\alpha(q) > q$ tel que le système $(\alpha(q), q, 1)$ ne soit pas canonique, et admette $\alpha(q) + 2$ comme plus petit contre-exemple.

Le résultat de la question 20 nous donne un algorithme déterminant si un système est canonique : il suffit de les rechercher dans l'intervalle discret $[\![c_{m-2} + 2, c_1 + c_2 - 1]\!]$. Ceci nécessite la construction des représentations minimales de chacun des éléments de cet intervalle.

Nous allons étudier un algorithme plus efficace, dû à Kozen et Zaks ; leur méthode repose sur la notion de témoin. Nous dirons qu'un entier x est un témoin pour le système $c = (c_i)_{1 \leq i \leq m}$ s'il existe un indice i tel que $G(x - c_i) < G(x) - 1$.

Question 23 Montrez que tout témoin est un contre-exemple.

Question 24 Montrez que le résultat de la question précédente n'admet pas de réciproque.

On pourra travailler dans le système $(5, 4, 1)$.

Question 25 Montrez que si le système c admet des contre-exemples, alors le plus petit d'entre eux est un témoin.

Il résulte de cette étude que, pour savoir si un système est canonique, il suffit de vérifier l'inégalité $G(x) \leq G(x - c_i) + 1$ pour tout $x \in [\![c_{m-2} + 2, c_1 + c_2 - 1]\!]$, et tout indice $i \in [\![1, m]\!]$ tel que $c_i < x$. C'est le principe de l'algorithme de Kozen et Zaks.

Question 26 En utilisant l'algorithme de Kozen et Zaks, montrer que les systèmes $(q, 2, 1)$, $(q \geq 3)$ et $(7, 4, 1)$ sont canoniques, alors que $(6, 5, 1)$ ne l'est pas.

Question 27 Rédigez en Caml une fonction de signature

```
kozen_zaks : int list -> bool
```

spécifiée comme suit : `kozen_zaks c` indique si la liste c est canonique.

Par exemple, `kozen_zaks [5 ; 2 ; 1]` rendra la valeur `true`, tandis que `kozen_zaks [6 ; 5 ; 1]` rendra la valeur `false`. Vous utiliserez l'algorithme de Kozen et Zaks.

Question 28 Montrez que le coût de l'algorithme de Kozen et Zaks peut être exponentiel, par rapport au nombre m de pièces du système.

On exhibera deux systèmes (l'un canonique, l'autre pas) pour lesquels le coût de l'algorithme est exponentiel en m.

DEVOIR 59

DS 8 - Concours blanc - Plus longue sous-suite commune et coloration de

59.1 DS 8 — Concours blanc : plus longue sous-suite commune et graphes d'intervalles

MP2I, année scolaire 2021-2022 — concours blanc — durée 4h, tout document ou dispositif électronique interdit.

Corrigé

Pas de corrigé disponible pour ce sujet.

59.1.1 Partie 1 : programmation dynamique

Plus longue sous-suite commune

En biologie, il est souvent nécessaire de comparer l'ADN provenant d'organismes différents. Un brin d'ADN est une chaîne de molécules appelées bases, où les bases sont l'adénine, la guanine, la cytosine et la thymine. En représentant chaque base par son initiale, un brin d'ADN est une chaîne de caractères sur les lettres A, G, C et T.

Par exemple, un organisme peut avoir comme ADN la chaîne

$$S_1 = \texttt{"ACCGGTCGAGTGCGCGGAAGCCGGCCGAA"}$$

et un autre

$$S_2 = \texttt{"GTCGTTCGGAATGCCGTTGCTCTGTAAA"}$$

Pour déterminer si deux organismes sont proches, il n'est pas suffisant d'observer les motifs contigus présents dans les deux chaînes car des mutations peuvent avoir modifié le brin en insérant des bases. Cependant, même dans ce cas, il doit être possible de retrouver une partie commune en supprimant des bases. On en déduit la définition suivante :

Définition 59.1

Soit $s = "c_0 \dots c_{n-1}"$ une chaîne de caractères, une chaîne s' est une sous-chaîne de s s'il existe un p-uplet $(i_1, ..., i_p)$ tel que $s' = "c_{i_1} \dots c_{i_p}"$ et $0 \leq i_1 < i_2 < ... < i_p \leq n-1$.

Le problème qu'on se pose est de déterminer la plus longue sous-chaîne commune à deux chaînes. Ainsi, « GTCGTCGGAAGCCGGCCGAA » est la plus longue sous-chaîne commune à S_1 et S_2.

59.1.1.1.1 Caractérisation des plus longues sous-suites communes

Afin de simplifier la programmation d'une solution, on ne va pas s'intéresser à des chaînes de caractères mais à des suites finies d'éléments. On notera ici $(a_0, a_1, ..., a_{n-1})$ une suite finie de n éléments. On adapte directement la notion précédente :

Définition 59.2

Soit $A = (a_0, ..., a_{n-1})$ une suite finie et B une autre suite finie. On dit que B est une sous-suite de A s'il existe un p-uplet $(i_1, ..., i_p)$ tel que $B = (a_{i_1}, ..., a_{i_p})$ et $0 \leq i_1 < i_2 < ... < i_p \leq n-1$.

Question 1 Si A est une suite finie de n éléments, combien comporte-t-elle de sous-suites ? Que penser d'un algorithme qui, pour déterminer une plus longue sous-suite commune entre A et B, itère sur les sous-suites de A et teste si elles sont sous-suites de B ?

Soit $A = (a_0, ..., a_{n-1})$ et $i \in [\![0, n]\!]$, on note $A_i = (a_0, ..., a_{i-1})$ le i-ème préfixe de A. Par définition, A_0 est la suite finie vide () et $A_n = A$.

Question 2 Soient $A = (a_0, ..., a_{m-1})$ et $B = (b_0, ..., b_{n-1})$ deux suites finies et $C = (c_0, ..., c_{p-1})$ une plus longue sous-suite commune à A et B.

Montrer que :

1. Si $a_{m-1} = b_{n-1}$ alors $c_{p-1} = a_{m-1} = b_{n-1}$ et C_{p-1} est une plus longue sous-suite commune à A_{m-1} et B_{n-1}.
2. Si $a_{m-1} \neq b_{n-1}$ alors $c_{p-1} \neq a_{m-1}$ entraîne que C est une plus longue sous-suite commune à A_{m-1} et B.
3. Si $a_{m-1} \neq b_{n-1}$ alors $c_{p-1} \neq b_{n-1}$ entraîne que C est une plus longue sous-suite commune à A et B_{n-1}.

59.1.1.1.2 Équation de récurrence

On considère ici deux suites $A = (a_0, ..., a_{m-1})$ et $B = (b_0, ..., b_{n-1})$ et, pour $i \in [\![0, m]\!]$ et $j \in [\![0, n]\!]$, on note $f(i, j)$ la longueur d'une plus grande sous-suite commune à A_i et B_j.

Question 3 1. Calculer $f(i, j)$ quand $i = 0$ **ou** $j = 0$. 2. On suppose que $i, j > 0$ et $a_{i-1} = b_{j-1}$. Justifier que $f(i, j) = 1 + f(i-1, j-1)$. 3. On suppose que $i, j > 0$ et $a_{i-1} \neq b_{j-1}$. Justifier que $f(i, j) = \max(f(i, j-1), f(i-1, j))$.

59.1.1.1.3 Résolution de la récurrence

On va résoudre la récurrence en OCaml ici.

Les sous-suites sont ainsi représentées par des `'a array`.

Question 4 Écrire une fonction naïve `longueur_sous_suite` de signature

```
'a array -> 'a array -> int -> int -> int
```

telle que l'appel à `longueur_sous_suite a b i j` renvoie la valeur de $f(i, j)$ pour les suites données par a et b.

On rappelle l'existence des fonctions suivantes :

- `Hashtbl.create n` qui crée une table de hachage de taille initiale n.
- `Hashtbl.find t c` qui renvoie la valeur associée à la clé `c` dans la table et produit l'exception `Not_found` s'il n'y en a pas.
- `Hashtbl.add t c v` qui rajoute une correspondance entre la clé `c` et la valeur `v`.

Question 5 Écrire une version mémoïsée de la fonction précédente.

On pourra, au choix, créer un cache en dehors de la fonction ou dans la fonction elle-même.

59.1.1.1.4 Tabulation

On va considérer ici une version tabulée en `C`.

On va représenter f dans une matrice de dimension $(m+1) \times (n+1)$. Pour simplifier, on peut considérer un tableau `M` de $(m+1)(n+1)$ entiers où on place $f(i, j)$ dans `M[j * (m+1) + i]`.

Question 6 Déterminer un ordre de remplissage de M qui soit compatible avec la relation de récurrence établie au paragraphe précédent.

Question 7 Écrire une fonction de prototype

```
int longueur_sous_suite(int *a, int m, int *b, int n);
```

qui renvoie la longueur d'une plus longue sous-suite commune aux deux tableaux d'entiers `a` de longueur `m` et `b` de longueur `n`.

On utilisera une tabulation selon le principe qui vient d'être exposé.

59.1.2 Partie 2 : problème sur les graphes

I. Graphes d'intervalles

On considère le problème concret suivant : certains cours doivent avoir lieu dans un intervalle de temps précis (par exemple de 8h à 9h50) et on cherche à attribuer une salle à chaque cours. On souhaite évidemment qu'à tout moment une salle ne puisse pas être attribuée à deux cours différents et on aimerait utiliser le plus petit nombre de salles possible.

Ce problème d'allocation de ressources (ici les salles) en fonction de besoins fixes (ici les horaires des cours) intervient dans de nombreuses situations très diverses (allocation de pistes d'atterrissage aux avions, répartition de la charge de travail sur plusieurs machines, etc.).

59.1.2.1.1 Représentation du problème

On modélise le problème ainsi :

- chaque besoin est représenté par un segment $[a, b]$ où $a, b \in \mathbb{N}$ et $a \leq b$;
- deux besoins I et J sont en conflit quand $I \cap J \neq \emptyset$.

La donnée du problème est une suite finie $(I_0, \ldots, I_{n-1})$ de n segments où $n \in \mathbb{N}$.

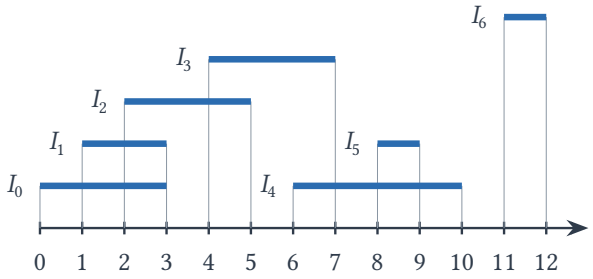


Figure 59.1 — Figure 1 — Problème (a)

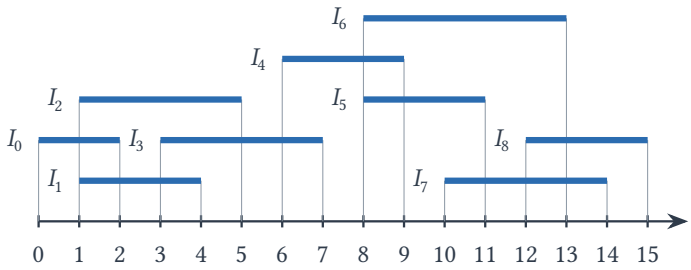


Figure 59.2 — Figure 2 — Problème (b)

Question 8 Dans le problème (b) quels sont les intervalles en conflit avec I_5 ?

On représente un segment en OCaml par un couple d'entiers, la donnée du problème est représentée par un tableau d'intervalles, de type `(int * int) array`. Dans toute la suite, on supposera toujours que les intervalles sont des segments bien formés et donc que si `(a, b) : int * int` représente un intervalle alors on a `a <= b` et cela représente toujours le segment $[a, b]$.

Le problème (a) de la figure 1 est ainsi représenté par le tableau

```
[|(0, 3); (1, 3); (2, 5); (4, 7);
  (6, 10); (8, 9); (11, 12)|]
```

On cherche à écrire une fonction `conflit` de type `int * int -> int * int -> bool` qui renvoie `true` si et seulement si deux segments passés en paramètres sont en conflit. Voici plusieurs versions proposées par plusieurs élèves.

```
let conflit1 (a, b) (c, d) = (a <= c) && (c <= b)
let conflit2 (a, b) (c, d) = (a < d) && (c < b)
let conflit3 (a, b) (c, d) = not ((b < c) || (d < a))
let conflit4 (a, b) (c, d) =
  ((a <= c) && (d <= b)) || ((c <= a) && (b <= d))
```

Question 9 Une seule de ces versions est correcte. Laquelle ?

On suppose par la suite disposer de la version correcte de cette fonction `conflit : int * int -> int * int -> bool`.

59.1.2.1.2 Graphe simple non orienté

On rappelle qu'un *graphe simple non orienté*, que l'on appellera par la suite simplement *graphe*, est un couple $G = (S, A)$ où

- S est un ensemble fini dont les éléments sont appelés les sommets du graphe (ici S peut être vide) ;
- A est un ensemble de paires (donc d'éléments distincts) de S. Lorsque $\{x, y\} \in A$ on dit que x et y sont reliés dans G et $\{x, y\}$ est appelée une arête de G. Les sommets reliés à un sommet x sont appelés les voisins de x.

Étant donnée une énumération de S sous la forme d'une suite $(x_0, \ldots, x_{n-1})$ on représente l'ensemble A en OCaml par un élément de type `aretes : int list array`. Ainsi pour $i \in \{0, \ldots, n-1\}$, la liste `aretes.(i)` contient les indices j tels que x_i soit relié à x_j dans G.

Question 10 Comment s'appelle cette représentation d'un graphe ?

On représente graphiquement le graphe G par un diagramme où les arêtes sont représentées par des traits entre les sommets représentés par des cercles.

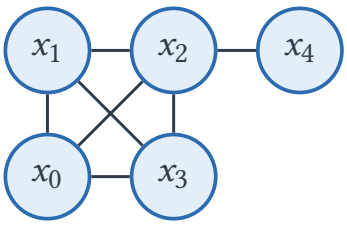


Figure 59.3 — Figure 3 — Un exemple de graphe

Les arêtes du graphe dont une représentation graphique est donnée à la figure 3 sont représentées en Caml par le tableau :

```
[|[1; 2; 3]; [0; 2; 3]; [0; 1; 3; 4]; [0; 1; 2]; [2]|]
```

Un tel tableau de listes d'arêtes suffit pour déterminer un graphe lorsque l'énumération des sommets est connue car on peut alors identifier un sommet à son indice. Dans la suite de ce problème on identifiera ainsi un graphe à son tableau de listes d'arêtes.

59.1.2.1.3 Graphe d'intervalles

Soit $\overline{I} = (I_0, \ldots, I_{n-1})$ une suite finie de segments, on appelle graphe d'intervalles associé à $\overline{I}$ le graphe $G(\overline{I})$

- dont les sommets sont les segments $I_0, \ldots, I_{n-1}$

- et où, pour $i, j \in \{0, \ldots, n-1\}$, avec $i \neq j$ les sommets I_i et I_j sont reliés si et seulement s'ils sont en conflit.

Par exemple, le graphe d'intervalles qui correspond au problème (a) de la figure 1 admet la représentation graphique de la figure 4.

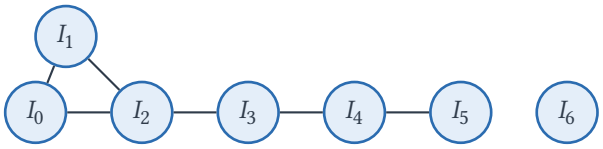


Figure 59.4 — Figure 4 — Graphe d'intervalles associé au problème (a)

Question 11 Donner une représentation graphique du graphe d'intervalles pour le problème (b) de la figure 2.

Question 12 Écrire en OCaml une fonction `construit_graphe` de type `(int * int) array -> int list array` qui étant donné un tableau `segments` des segments $\overline{I} = (I_0, \ldots, I_{n-1})$, énumérés dans cet ordre, renvoie la représentation des arêtes de G. Quelle est sa complexité ?

II. Coloration d'un graphe

Soit $G = (S, A)$ un graphe simple non orienté dont les sommets sont donnés par une énumération $x_0, \ldots, x_{n-1}$. On appelle coloration de G (associée à cette énumération) une suite finie d'entiers naturels $(c_0, \ldots, c_{n-1})$ telle que

$$\forall (i, j) \in \{0, \ldots, n-1\}^2, \{x_i, x_j\} \in A \Rightarrow c_i \neq c_j$$

L'entier c_i est appelé la couleur du sommet x_i et la condition se comprend ainsi : deux sommets reliés ont des couleurs distinctes. Dorénavant, le terme couleur sera synonyme d'entier naturel. Par exemple, la suite finie $(0, 1, 2, 3, 0)$ est une coloration du graphe de la figure 3, utilisant 4 couleurs.

Lorsqu'une coloration utilise le plus petit nombre de couleurs distinctes possibles, on dit qu'elle est optimale. On note alors $\chi(G)$ ce nombre minimum de couleurs, appelé le *nombre chromatique* de G.

> **Remarque 59.3** En associant une salle à chaque couleur, on peut répondre au problème initial à l'aide d'une coloration de son graphe d'intervalles associé.

Question 13 Déterminer $\chi(G)$ pour les deux graphes d'intervalles associés aux problèmes (a) de la figure 1 et (b) de la figure 2. Justifier rigoureusement.

59.1.2.2.1 Couleurs disponibles

Question 14 Quelle est la complexité temporelle de la fonction `List.mem` ?

Question 15 Écrire en OCaml une fonction `plus_petit_absent : int list -> int` telle que l'appel `plus_petit_absent liste` sur une liste d'entiers naturels `liste` renvoie le plus petit entier naturel non présent dans `liste`. Justifier que cette fonction termine. Une fonction de complexité quadratique en la taille de la liste dans le pire cas convient ici.

Question 16 Expliquer précisément comment on pourrait réaliser la fonction précédente avec une complexité linéaire en la taille de la liste dans le pire cas. On ne demande pas d'écrire la fonction en OCaml mais d'en décrire précisément l'idée.

Par la suite on suppose que la fonction `plus_petit_absent` réalise cette complexité linéaire dans le pire cas.

On considère une coloration progressive des sommets d'un graphe. Pour cela, une coloration partielle est un tableau `couleurs : int array` tel que `couleurs.(i)` contient la couleur de i s'il est coloré et -1 sinon, ce qui ne pose pas de problème car les couleurs sont toujours positives.

Question 17 Écrire en OCaml une fonction `couleurs_voisins` de signature `int list array -> int array -> int -> int list` telle que l'appel `couleurs_voisins aretes couleurs i` renvoie, avec les doublons éventuels et

dans un ordre quelconque, la liste des couleurs des voisins colorés du sommet d'indice i dans le graphe décrit par `aretes` où le tableau `couleurs` décrit une coloration partielle. Quelle est sa complexité ?

Question 18 En déduire une fonction `couleur_disponible : int list array -> int array -> int -> int` de telle manière à ce que l'appel de fonction `couleur_disponible aretes couleurs i` renvoie la plus petite couleur pouvant être attribuée au sommet i afin qu'il n'ait la couleur d'aucun de ses voisins dans le graphe décrit par `aretes`. Quelle est sa complexité ?

59.1.2.2.2 Colorations et cliques

Soit $G = (S, A)$ un graphe. Un sous-ensemble $C \subseteq S$ est appelé une clique de G lorsqu'il vérifie

$$\forall x, y \in C, x \neq y \Rightarrow \{x, y\} \in A$$

La taille d'une clique est son nombre d'éléments. La taille de la plus grande clique de G (celle qui possède le plus d'éléments) est notée $\omega(G)$.

Question 19 Déterminer, sans justifier, $\chi(G)$ et $\omega(G)$ pour un graphe $G = (S, A)$ avec $|S| = n > 0$ lorsque

- (a) G ne possède pas d'arête (tous les sommets sont isolés) ;
- (b) G est un graphe complet à n sommets ;
- (c) G est un cycle à n sommets.

Question 20 Comparer $\chi(G)$ et $\omega(G)$ pour un graphe G quelconque. Justifier.

III. Algorithme glouton pour la coloration d'intervalles

Étant donnée une liste de segments $\overline{I} = (I_0, ..., I_{n-1})$ de longueur $n \geq 1$, on se propose de déterminer une coloration optimale de son graphe d'intervalles associé. On appelle coloration de $\overline{I}$ une suite finie d'entiers naturels $(c_0, ..., c_{n-1})$ telle que

$$\forall (i, j) \in \{0, ..., n-1\}^2, I_i \cap I_j \neq \emptyset \Rightarrow c_i \neq c_j$$

On suppose dans cette partie que les segments $I_k = [a_k, b_k]$, pour $k \in \{0, ..., n-1\}$, sont énumérés dans l'ordre croissant de leurs extrémités gauches, c'est-à-dire que

$$a_0 \leq a_1 \leq ... \leq a_{n-1}$$

On propose l'algorithme suivant :

> Pour k variant de 0 à $n-1$, colorer l'intervalle I_k avec la plus petite couleur non encore utilisée dans la coloration des intervalles I_j, avec $0 \leq j < k$, qui ont une intersection non vide avec I_k.

Ainsi, l'intervalle I_0 est toujours coloré avec la couleur 0, l'intervalle I_1 reçoit la couleur 0 si $I_0 \cap I_1 = \emptyset$, et la couleur 1 sinon, etc.

59.1.2.3.1 L'algorithme sur un exemple

Question 21 Déterminer la coloration renvoyée par cet algorithme pour le problème (b) décrit sur la figure 2, pour l'énumération des intervalles indiquée (qui respecte bien la contrainte demandée).

59.1.2.3.2 Coloration

Question 22 Écrire en OCaml une fonction `coloration : int list array -> int array` telle que l'appel `coloration aretes`, où `aretes` représente les arêtes du graphe d'intervalles associé à une liste de segments triés par ordre croissant de leurs extrémités gauches, renvoie la coloration obtenue avec l'algorithme ci-dessus.

Question 23 Déterminer *très soigneusement* la complexité de la fonction précédente en fonction du nombre n d'intervalles et du nombre m d'arêtes du graphe d'intervalles associé à la liste $\overline{I}$.

59.1.2.3.3 Preuve de l'algorithme

On se propose maintenant de démontrer que l'algorithme ci-dessus fournit une coloration optimale de l'ensemble des segments. Soit k un entier entre 0 et $n-1$. On suppose qu'à la k-ième étape de l'algorithme, le segment $I_k = [a_k, b_k]$ reçoit la couleur c.

Question 24 Montrer que I_k a au moins c voisins d'indices inférieurs à k, c'est-à-dire parmi $I_0, \ldots, I_{k-1}$.

Question 25 Montrer que l'ensemble constitué de I_k et de ses voisins d'indice inférieur à k constitue une clique de taille au moins $c+1$ dans le graphe d'intervalles associé. *On pourra raisonner sur l'extrémité gauche a_k du segment I_k.*

Question 26 En déduire que le nombre de couleurs nécessaires à une coloration de l'ensemble des segments est au moins égal à $c+1$.

Question 27 Conclure.

IV. Coloration gloutonne

L'algorithme de coloration étudié dans la partie précédente s'applique plus généralement à tout graphe étant donné une énumération $(x_0, \ldots, x_{n-1})$ des sommets :

> pour i allant de 0 à $n-1$, on colore x_i avec la plus petite couleur qui ne soit pas utilisée par un de ses voisins déjà colorés.

On appelle *coloration gloutonne* d'un graphe $G = (S, A)$ une coloration obtenue par application de l'algorithme ci-dessus, ce qui dépend uniquement de l'énumération des sommets choisie. Ainsi, si $|S| = n$, chacune des $n!$ façons d'ordonner les sommets du graphe induit une coloration gloutonne des sommets du graphe.

Question 28 Montrer qu'une coloration gloutonne n'est pas toujours optimale. *On donnera un contre-exemple sous la forme d'un graphe, d'une énumération de ses sommets, de la coloration gloutonne associée et du nombre chromatique de ce graphe (qui ne sera donc pas le même que le nombre de couleurs utilisées par la coloration gloutonne).*

Question 29 Montrer qu'il existe des colorations optimales qui ne sont pas des colorations gloutonnes. *On donnera à nouveau un contre-exemple sous la forme d'un graphe et d'une coloration associée et on justifiera rapidement que cette coloration ne saurait être une coloration gloutonne.*

L'algorithme de coloration glouton pour une énumération des sommets ne donne pas, en général, une solution optimale. Cependant, il existe toujours une énumération des sommets qui fournit une coloration gloutonne optimale.

Propriété 59.4

Soit un graphe $G = (S, A)$. Il existe une énumération $(x_0, \ldots, x_{n-1})$ des sommets de G telle que la coloration gloutonne calculée à partir de cet ordre soit optimale.

On se propose de démontrer cette proposition. Considérons une coloration valide du graphe G utilisant k couleurs numérotées de 0 à $k-1$ et notons, pour $i \in \{0, \ldots, k-1\}$, C_i l'ensemble des sommets de G auxquels a été affectée la couleur i dans cette coloration. Considérons une énumération des sommets de G qui respecte l'ordre sur les classes C_i, c'est-à-dire telle que si $0 \leq i < j \leq k-1$, tout sommet de couleur i est classé avant tout sommet de couleur j. Considérons maintenant la coloration gloutonne obtenue avec cet ordre.

Question 30 Montrer que, pour $i \in \{0, \ldots, k-1\}$, lorsque l'algorithme glouton affecte une couleur c à un sommet x de la classe C_i, cette couleur c ne peut être qu'inférieure ou égale à i.

Question 31 Montrer que cette coloration gloutonne utilise au plus k couleurs.

Question 32 Conclure.

Il suffit donc de calculer les colorations gloutonnes associées à toutes les énumérations possibles des sommets pour trouver le nombre chromatique d'un graphe.

Question 33 Que pensez-vous de cette approche ?

Trouver un ordre sur les sommets d'un graphe tel que la coloration gloutonne associée à cet ordre soit optimale est un problème difficile que l'on ne sait pas résoudre de manière efficace, du moins dans le cas général.

V. Graphes munis d'un ordre d'élimination parfait

On introduit ici la notion d'ordre d'élimination parfait, dont on montre qu'il existe toujours pour un graphe d'intervalles, et pour lequel l'algorithme glouton donne toujours une solution optimale.

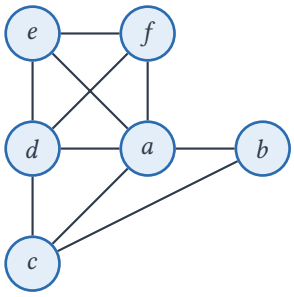


Figure 59.5 — Figure 5 — Un exemple de graphe admettant un ordre d'élimination parfait

Soient $G = (S, A)$ un graphe et $(x_0, ..., x_{n-1})$ une énumération des sommets de G. Pour tout $i \in \{0, ..., n-1\}$, on note $G_i = (S_i, A_i)$ où $S_i = \{x_0, ..., x_i\}$ et

$$\forall k, l \in \{0, ..., n-1\}, \{x_k, x_l\} \in A_i \Leftrightarrow (k, l \leq i \text{ et } \{x_k, x_l\} \in A)$$

G_i est ainsi le graphe déduit de G en se restreignant aux sommets de x_0 à x_i.

Une énumération $(x_0, ..., x_{n-1})$ des sommets de G est appelée un *ordre d'élimination parfait* si pour tout $i \in \{0, ..., n-1\}$, les voisins de x_i d'indices inférieurs à i (c'est-à-dire apparaissant avant i dans l'énumération) forment une clique (c'est-à-dire sont tous reliés).

Par exemple, dans la figure 5, l'énumération (a, b, c, d, e, f) est bien un ordre d'élimination parfait alors que (b, c, e, d, f, a) ne l'est pas puisque d y est précédé de e et c qui sont tous deux voisins de d sans être reliés.

59.1.2.5.1 Un exemple

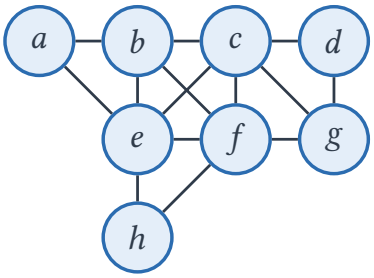


Figure 59.6 — Figure 6 — Graphe dont on demande dans la question suivante de déterminer des ordres d'élimination parfaits

Question 34 Déterminer, sans justifier, parmi les exemples proposés lesquels sont bien des ordres d'élimination parfaits pour le graphe G donné par la figure 6 :

- (a) (a, b, c, d, e, f, g, h).
- (b) (a, c, b, d, f, g, h, e).
- (c) (a, b, e, c, f, h, g, d).
- (d) (b, c, e, f, a, g, d, h).

Question 35 Montrer qu'il existe des graphes qui n'admettent aucun ordre d'élimination parfait. *On donnera un contre-exemple sous la forme d'un graphe et l'on justifiera rapidement.*

59.1.2.5.2 Validation d'un ordre d'élimination parfait

Dans cette partie, les fonctions devront être écrites en C. Pour cela, un graphe à n sommets et de degré maximum d sera représenté par un tableau statique `g` à n lignes et $d+1$ colonnes où chaque ligne `i` du tableau `g` est un tableau `g[i]` contenant dans sa première case `g[i][0]` le degré d_i du sommet i, puis, dans les cases indices 1 à d_i, et dans un ordre arbitraire, les indices des voisins de i. On n'accèdera donc pas aux cases d'indices strictement supérieurs à d_i pour un sommet i, qui peuvent contenir une valeur non initialisée.

Par exemple, le graphe de la figure 3 peut être représenté par :

```
int g[5][4] = {
    {3, 1, 2, 3},
    {3, 0, 3, 2},
    {3, 0, 1, 3},
    {3, 0, 1, 2},
    {1, 2}
};
```

On remarquera qu'ici encore, une fois que la représentation d'un graphe est choisie, elle implique une énumération des sommets dont on cherchera ici à déterminer si elle correspond ou non à un ordre d'élimination parfait.

Étant donné un graphe $G = (S, A)$ et un ensemble $S' \subseteq S$ de sommets de G, le *sous-graphe de G induit par S'* est le graphe $H = (S', A')$ où $A' \subseteq A$ est l'ensemble des arêtes de G dont les extrémités appartiennent à S'. On pourra identifier le sous-graphe $H = (S', A')$ avec l'ensemble de sommets S'.

Un graphe $G = (S, A)$ à $|S| = n$ sommets est représenté en C par un tableau bidimensionnel `g`. On représente en C un sous-graphe induit $H = (S', A')$ d'un graphe $G = (S, A)$ possédant n sommets par la donnée du graphe d'origine `g` et d'un tableau de booléens `sommets` à n cases dont la case i vaut `true` si et seulement si le sommet d'indice i est un sommet du sous-graphe induit et `false` sinon. On représente de même un sous-ensemble $S' \subseteq S$ de sommets du graphe par un tableau de booléens de taille n indiquant pour chaque sommet de S sa présence dans S' ou non.

Attention

Dans toute cette partie n et d seront des constantes globales.

Question 36 Écrire en C une fonction de type `void voisins_inferieurs(int g[n][d + 1], int x, bool voisins_inf[n])` telle que l'appel `voisins_inferieurs(g, x, voisins_inf)` met à jour le tableau de booléens `voisins_inf` pour qu'il représente l'ensemble des voisins du sommet `x` d'indices strictement inférieurs à celui de `x`. On ne présupposera rien sur le remplissage initial du tableau `voisins_inf`. On garantira une complexité linéaire en $|S| = n$.

Question 37 Écrire en C une fonction de type `bool est_clique(int g[n][d + 1], bool sommets[n])` telle que l'appel `est_clique(g, sommets)` renvoie `true` si et seulement si l'ensemble de sommets représenté par le tableau de booléens `sommets` forme une clique de `g`, c'est-à-dire que le sous-graphe de `g` engendré par ces sommets est complet. On garantira une complexité linéaire en $|S| + |A|$. *Indication : calculer en temps linéaire la taille et le nombre d'arêtes du sous-graphe.*

Question 38 Écrire en C une fonction de signature `bool est_ordre_parfait(int g[n][d + 1])` telle que `est_ordre_parfait(g)` renvoie `true` si et seulement si l'énumération associée au graphe représenté par `g` est un ordre d'élimination parfait. Quelle est sa complexité ?

59.1.2.5.3 Ordre d'élimination parfait pour un graphe d'intervalles

Question 39 Justifier rapidement qu'un graphe d'intervalles admet un ordre d'élimination parfait. *Indication : on en a explicité un dans la partie III.*

59.1.2.5.4 Ordre d'élimination parfait

Un sommet v d'un graphe G est dit simplicial lorsque l'ensemble des voisins de v dans G est une clique.

Question 40 Quels sont les sommets simpliciaux du graphe de la figure 5 ?

Question 41 Écrire une fonction de type `bool simplicial(int g[n][d + 1], bool sg[n], int k)` telle que l'appel `simplicial(g, sg, k)`, où le sommet d'indice k est supposé appartenir au sous-graphe induit H décrit par `(g, sg)`, renvoie `true` si le sommet d'indice k est simplicial dans H et `false` sinon. Déterminer la complexité de la fonction `simplicial`.

Question 42 Écrire une fonction `int trouver_simplicial(int g[n][d + 1], bool sg[n])` telle que `trouver_simplicial(g, sg)` renvoie, s'il en existe, un sommet simplicial du sous-graphe induit décrit par `(g, sg)` et sinon renvoie l'entier relatif -1. Déterminer la complexité de cette fonction.

Question 43 Montrer que si un graphe $G = (S, A)$ possède un ordre d'élimination parfait, alors pour tout sommet simplicial x de G, le sous-graphe $H = (S', A')$ engendré par $S' = S \setminus \{x\}$ en possède également un.

Question 44 Écrire une fonction `int *ordre_parfait(int g[n][d + 1])` telle que l'appel `ordre_parfait(g)` renvoie un ordre d'élimination parfait du graphe décrit par `g`, s'il en existe un et `NULL` sinon. Déterminer la complexité de la fonction `ordre_parfait`.

DEVOIR 60

Devoir des vacances de Pâques - Dessin de graphes, chasseurs de fan-

60.1 DM de Pâques — Dessin de graphes, chasseurs de fantômes, expressions

MP2I, année scolaire 2021-2022 — devoir pour les vacances de Pâques.

Sources

La partie « Chasseurs de fantômes » est d'après le sujet d'informatique de Centrale 2009 (« La chasse aux fantômes »). La partie « Dessin de graphes » cite un extrait de l'article de P. Eades, *A heuristic for graph drawing*.

60.1.1 Dessin de graphes

Il s'agit ici de la partie bonus du TP19 avec plus d'explications.

On reproduit l'explication succincte mais complète de l'article de 1983 de Eades :

To embed a graph we replace the vertices by steel rings and replace each edge with a spring to form a mechanical system. The vertices are placed in some initial layout and let go so that the spring forces on the rings move the system to a minimal energy state. Two practical adjustments are made to this idea: firstly, logarithmic strength springs are used; that is, the force exerted by a spring is:

$$c_1 * \log(d/c_2),$$

where d is the length of the spring, and c_1 and c_2 are constants. Experience shows that Hookes Law (linear) springs are too strong when the vertices are far apart; the logarithmic force solves this problem. Note that the springs exert no force when $d = c_2$. Secondly, we make nonadjacent vertices repel each other. An inverse square law force,

$$c_3/d^2,$$

where c_3 is constant and d is the distance between the vertices, is suitable. The mechanical system is simulated by the following algorithm.

---

algorithm SPRING(G:graph);
place vertices of G in random locations;
repeat M times
    calculate the force on each vertex;
    move the vertex $c_4 * (force\ on\ vertex)$
draw graph on CRT or plotter.

The values $c_1 = 2$, $c_2 = 1$, $c_3 = 1$, $c_4 = 0.1$, are appropriate for most graphs. Almost all graphs achieve a minimal energy state after the simulation step is run 100 times, that is, $M = 100$.

Figure 60.1 — Extrait de l'article d'Eades

Implémenter cette stratégie pour afficher des graphes avec `graphics`. Testez-la sur des graphes variés en nombre de sommets et d'arêtes.

60.1.2 Chasseurs de fantômes

Un groupe de n chasseurs de fantômes combat un groupe de n fantômes. Chaque chasseur est armé d'un canon à ions, capable d'éradiquer un fantôme d'un coup de rayon. Un rayon se propage en ligne droite et termine sa course en touchant le fantôme. Les chasseurs sont confrontés à deux problèmes : il est nécessaire d'éliminer tous les fantômes en même temps, sinon ils se multiplient pour revenir au nombre initial de n fantômes ; il est très dangereux que deux rayons se croisent, cela ne doit donc pas se produire. Le combat se déroule dans une grotte cylindrique creuse, dont la base est un disque. Les déplacements (chasseurs et fantômes) ne sont possibles que sur un rebord confondu avec la circonférence, mais les tirs se font entre deux points de la circonférence, selon des cordes du cercle.

A

Les $2n$ protagonistes sont représentés par des points distincts $P_1, P_2, ..., P_{2n}$ répartis dans l'ordre des indices croissants sur la circonférence d'un cercle parcouru dans le sens trigonométrique ; n sont des chasseurs, n des fantômes, mais, dans cette section, on cherche les liens possibles entre les P_i sans s'occuper de qui est

chasseur et qui est fantôme. Les points doivent être reliés deux à deux par une corde du cercle selon une stratégie gagnante pour les chasseurs. Pour cela, il faut donc respecter les deux conditions :

1. chaque point est une extrémité d'une et une seule corde,
2. les différentes cordes ne doivent pas se couper.

Dans toute la suite on fixe un entier $n > 0$ et on appelle *stratégie de taille n* un $2n$-uplet $(c_1, c_2, ..., c_{2n})$ tel que pour tout entier i vérifiant $1 \leq i \leq 2n$ on ait :

$$1 \leq c_i \leq 2n, \qquad c_{c_i} = i, \qquad c_i \neq i.$$

Cette notation exprime bien le lien entre les points P_i et P_{c_i}.

En C, on représente le $2n$-uplet avec un tableau de `int`. Les éléments seront donc numérotés de 0 à $N - 1$. Pour simplifier, on prendra des tableaux de taille $2n + 1$ et on ignorera la valeur à l'indice 0. Ainsi `c[i]` contiendra c_i.

On considère des stratégies où les points d'une paire sont reliés par des segments de droites situés dans un même plan. Une stratégie est dite « admissible » si les segments ainsi formés ne se coupent pas. On admet que cette propriété ne dépend pas de la position des points sur la circonférence mais seulement de la stratégie.

Par exemple, pour les stratégies de taille 4, c_1 et c_2 données dans le tableau, on présente les figures correspondantes, c_1 admissible à gauche, et c_2 non admissible à droite :

	1	2	3	4	5	6	7	8
c_1	2	1	6	5	4	3	8	7
c_2	2	1	7	5	4	8	3	6

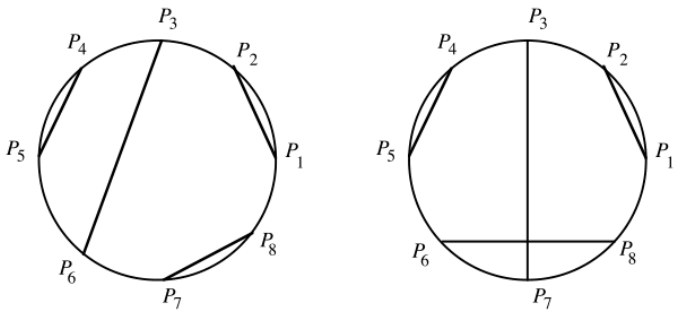


Figure 60.2 — Stratégies c1 et c2

Question 1 Dans le cas $n = 3$, donner le nombre de stratégies possibles, admissibles ou non.

Question 2 Préciser le nombre de stratégies de taille 3 qui sont admissibles. Les représenter sur une figure.

Question 3 Déterminer le nombre de stratégies de taille n entier positif (qu'elles soient ou non admissibles).

B

Soit i, j, k, l quatre entiers distincts tels que : $1 \leq i < j \leq 2n$ et $1 \leq k < l \leq 2n$.

Question 4 Donner une condition nécessaire et suffisante, portant uniquement sur ces entiers, pour que les segments $[P_iP_j]$ et $[P_kP_l]$ se croisent.

Question 5 Écrire une fonction `croise` prenant les quatre entiers pour paramètres et renvoyant un booléen dont le résultat est `true` si et seulement si les segments $[P_iP_j]$ et $[P_kP_l]$ se croisent. On suppose que les paramètres donnés à la fonction satisfont les conditions données en introduction de cette partie.

Corrigé

```
int croise(int i, int j, int k, int l)
{
```

```
    return (i < k && k < j && j < l)
        || (k < i && i < l && l < j);
}
```

Question 6 En utilisant la fonction `croise`, écrire une fonction `estAdmissible` qui prend en paramètre le tableau et le nombre n de chasseurs, et renvoie un booléen dont le résultat est `true` si la stratégie est admissible.

Corrigé

```
int estAdmissible(int *c, int n)
{
    int p = 2 * n + 1;
    for (int i = 1; i < p; i++)
    {
        for (int k = i+1; k < c[i]; k++)
        {
            if (k < c[k] && croise(i, c[i], k, c[k]))
                return false;
            if (k >= c[k] && croise(i, c[i], c[k], k))
                return false;
        }
    }

    return true;
}
```

Question 7 Donner, en la justifiant, la complexité de cette méthode, en terme de temps de calcul (on se contentera d'une évaluation du type $O(n^\alpha)$, en précisant la valeur de α).

C

On souhaite tester la stratégie de façon plus efficace. Pour cela on utilise une fonction `evalue` qui prend deux entiers i et j comme paramètres (i et j étant compris entre 1 et $2n$) et dont le résultat est `true` si et seulement si les deux propositions suivantes sont vraies :

- $(i \leq k \leq j) \Rightarrow (i \leq c_k \leq j)$
- les segments ayant leurs extrémités dans l'arc (fermé) $[P_iP_j]$ ne se croisent pas.

Question 8 Donner la valeur de `evalue(i,j)` dans les trois cas particuliers a, b et c suivants :

$$a)\ j < i; \quad b)\ j \geq i \text{ et } c_i < i; \quad c)\ j \geq i \text{ et } c_i > j$$

Question 9 Pour $i < c_i < j$, montrer que `evalue(i,j)` se déduit de `evalue(i+1,c[i]-1)` et de `evalue(c[i]+1,j)`.

Question 10 Écrire une fonction `testStrategie` qui prend en paramètre un tableau et le nombre n de chasseurs, et renvoie un booléen dont la valeur est `true` si la stratégie est admissible. Cette fonction devra être de complexité $O(n)$.

Corrigé

```
bool evalue(int *c, int i, int j)
{
    if (j < i)
        return true;
    if (c[i] < i || c[i] > j)
        return false;
    return evalue(c, i+1, c[i]-1)
        && evalue(c, c[i]+1, j);
}

bool testStrategie(int *c, int n)
```

```
{
    return evalue(c, 1, 2*n);
}
```

Question 11 Démontrer avec soin que la fonction `testStrategie` est bien de complexité $O(n)$.

D

La méthode précédente teste si une stratégie est admissible. On souhaite maintenant, le statut (chasseur ou fantôme) des points étant donné, déterminer directement une stratégie admissible, de façon également efficace.

Question 12 Montrer que si P_i est un chasseur ($1 \leq i \leq 2n$), alors il existe $j \neq i$, avec $1 \leq j \leq 2n$ tel que P_j est un fantôme et tel que le nombre de chasseurs soit égal au nombre de fantômes de chaque côté de l'axe P_iP_j.

Question 13 En déduire un algorithme simple pour construire une stratégie admissible.

Question 14 On se propose d'évaluer la complexité de cette méthode.

1. Déterminer la complexité dans le pire des cas.
2. Évaluer, sans démonstration, la complexité moyenne.

Question 15 Un tableau p contient la nature des points P_i : `p[i]` vaut 1 pour un chasseur et -1 pour un fantôme ($1 \leq i \leq 2n$). On a donc $\sum_{i=1}^{2n} p[i] = 0$ puisqu'il y a autant de fantômes que de chasseurs. Écrire une fonction `cibles` qui prend en paramètre le tableau p et le nombre n de chasseurs, et renvoie un tableau T de $2n+1$ valeurs représentant les couples d'une stratégie admissible où le chasseur $T[2i+1]$ cible le fantôme $T[2i+2]$ (on ignore ici aussi la première case du tableau).

Corrigé

```
// renvoie k tel qu'il y ait autant de fantômes que de chasseurs
// dans PiPk
int coupe(int *p, int n, int i)
{
    int k = i;
    int s = p[i];
    while (s != 0)
    {
        k++;
        s += p[k];
    }
    return k;
}

// Remplit récursivement le tableau t
void tir(int *p, int *t, int i, int j, int nbTir)
{
    if (p[i] == 1)
    {
        t[2*nbTir+1] = i;
        t[2*nbTir+2] = j;
    }
    else
    {
        t[2*nbTir+1] = j;
        t[2*nbTir+2] = i;
    }
}

int remplit(int *p, int n, int *t, int i, int j, int nbTir)
{
    if (j <= i)
        return nbTir;
```

```
    if (j == i+1)
    {
        tir(p, t, i, j, nbTir);
        return nbTir+1;
    }

    int k = coupe(p, n, i);
    nbTir = remplit(p, n, t, i+1, k-1, nbTir);
    nbTir = remplit(p, n, t, k+1, j, nbTir);
    tir(p, t, i, k, nbTir);

    return nbTir+1;
}

int *cibles(int *p, int n)
{
    int *t = malloc(sizeof(int) * (2*n+1));
    remplit(p, n, t, 1, 2*n, 0);
    return t;
}
```

Exemple d'utilisation :

```
int main(void)
{
    int p[] = { 0, 1, -1, 1, 1, -1, -1, 1, -1 };
    int n = 4;

    int *t = cibles(p, n);

    for (int i = 1; i <= n; i++)
    {
        printf("%d -> %d\n", t[2*i-1], t[2*i]);
    }
    free(t);
}
```

60.1.3 Manipulation d'expressions

L'objectif de ce devoir, à réaliser sur machine, est de manipuler informatiquement des expressions mathématiques. On a déjà vu cela avec l'évaluation d'expressions arithmétiques, on va ici prolonger ce qui a été fait en rajoutant :

- des variables
- des fonctions
- des simplifications
- des dérivées ou primitives formelles
- ...

Pour cela, on va utiliser un type d'expression qui va changer à mesure qu'on progresse dans ce devoir.

Expressions avec variables libres

On va utiliser le type `expr` suivant :

```
type bin_op = Add | Mult | Minus | Div
type un_op = Neg | Inv
type expr =
  | Const of int
  | Var of string
  | BinOp of expr * bin_op * expr
  | UnOp of un_op * expr
```

Ainsi, l'expression $-\frac{x+1}{2+x^2}$ sera représentée par :

```
UnOp(Neg, BinOp(BinOp(Var "x",Add,Const 1),
                Div, BinOp(Const 2, Add, BinOp(Var "x", Mult, Var "x"))))
```

Les variables ici sont nécessairement libres : elles ne peuvent être liées comme dans $\forall x, P(x)$ ou $x \mapsto x + 1$.

Question 16 Écrire une fonction `variables_libres : expr -> string list` qui renvoie la liste des variables apparaissant dans l'expression.

Corrigé

```
let rec variables_libres e = match e with
  | Const _ -> []
  | Var x -> [x]
  | BinOp (e, _, e') -> variables_libres e @ variables_libres e'
  | UnOp (_, e) -> variables_libres e
```

Question 17 Écrire une fonction `string_of_expr : expr -> string` qui renvoie une expression sous la forme d'une chaîne de caractère lisible.

Dans l'exemple introductif, on renverra la chaîne : `"-((x+1)/(2+(x*x)))"`.

Corrigé

```
let string_of_bin_op op = match op with
  | Add -> "+" | Mult -> "*" | Minus -> "-" | Div -> "/"

let string_of_un_op op = match op with
  | Neg -> "-" | Inv -> "1/"

let rec string_of_expr e =
  let protect e =
    let s = string_of_expr e in
    match e with
    | Const _ | Var _ -> s
    | _ -> "(" ^ s ^ ")"
  in
  match e with
  | Const n -> string_of_int n
  | Var x -> x
  | BinOp(e, op, e') ->
    protect e
    ^ string_of_bin_op op
    ^ protect e'
  | UnOp(op, e) -> string_of_un_op op ^ protect e
```

Pour évaluer une telle expression, il est nécessaire de connaitre la valeur de chaque variable. Une association de valeurs aux variables est appelé un environnement d'évaluation. Pour simplifier, on va supposer qu'un environnement est une liste de couples (*variable*, *valeur*).

```
type env = (string * float) list
```

Attention : dans l'expression, les constantes sont uniquement entières mais en raison de la présence de la division, il est nécessaire que l'évaluation se fasse avec des flottants.

Question 18 Écrire une fonction `eval : env -> expr -> float` qui évalue l'expression étant donné un environnement associant une valeur à chacune des variables.

Corrigé

```
let rec eval env e =
  match e with
```

```
  | Const n -> float_of_int n
  | Var x -> List.assoc x env
  | BinOp(e, op, e') ->
    let v = eval env e in
    let v' = eval env e' in
    (match op with
     | Add -> v +. v'
     | Mult -> v *. v'
     | Minus -> v -. v'
     | Div -> v /. v')
  | UnOp(Neg, e) -> -. (eval env e)
  | UnOp(Inv, e) -> 1. /. (eval env e)
```

Si e, e' sont deux expressions et x est une variable, on note $e[x \setminus e']$ l'expression déduite de e en substituant chaque occurrence libre (pour le moment elles le sont toutes) de x par l'expression e'.

Exemple : si $e = (2 + x) - (x \times x)$ et $e' = 4 + y$ alors $e[x \setminus e'] = (2 + (4 + y)) - ((4 + y) \times (4 + y))$.

Si x n'est pas une variable libre de e, alors $e[x \setminus e'] = e$.

Question 19 Écrire une fonction `subst : expr -> string -> expr -> expr` telle que `subst e x e'` renvoie $e[x \setminus e']$.

Corrigé

```
let rec subst e x e' =
  match e with
  | Var y when x = y -> e'
  | BinOp(e1, op, e2) ->
    BinOp(subst e1 x e', op, subst e2 x e')
  | UnOp(op, e) -> UnOp(op, subst e x e')
  | _ -> e
```

On va essayer de simplifier des expressions en appliquant les règles suivantes :

- $e + 0 \to e$
- $0 + e \to e$
- $e - 0 \to e$
- $0 - e \to -e$
- $0 \times e \to 0$
- $e \times 0 \to 0$
- $1 \times e \to e$
- $e \times 1 \to e$
- $0/e \to 0$
- $e/1 \to e$
- $e \times (e' + e'') \to (e \times e') + (e \times e'')$
- $(e + e') \times e'' \to (e \times e'') + (e' \times e'')$

Question 20 Écrire une fonction `simpl : expr -> expr` appliquant ces règles.

Corrigé

```
let rec simpl e = match e with
  | BinOp(e, Add, Const 0)
  | BinOp(e, Mult, Const 1)
  | BinOp(e, Div, Const 1)
  | BinOp(Const 1, Mult, e)
  | BinOp(Const 0, Add, e) -> simpl e
  | BinOp(e, Minus, Const 0) -> simpl e
```

```
  | BinOp(Const 0, Minus, e) -> UnOp(Neg, simpl e)
  | BinOp(e, Mult, Const 0)
  | BinOp(Const 0, Mult, e)
  | BinOp(Const 0, Div, e) -> Const 0
  | BinOp(BinOp(e', Add, e''), Mult, e)
  | BinOp(e, Mult, BinOp(e', Add, e'')) ->
    let s_e = simpl e in
    simpl (BinOp(BinOp(s_e, Mult, simpl e'),
                 Add, BinOp(s_e, Mult, simpl e'')))
  | BinOp(e, op, e') ->
    let s_e = simpl e in
    let s_e' = simpl e' in
    let bin = BinOp(s_e, op, s_e') in
    if s_e <> e || s_e' <> e'
    then simpl bin (* pour propager *)
    else bin
  | UnOp(op, e) -> UnOp(op, simpl e)
  | _ -> e
```

Essayez d'améliorer au maximum votre fonction de simplification.

Dérivation formelle

Question 21 Écrire une fonction `derive : string -> expr -> expr` telle que `derive x e` renvoie l'expression correspondant à la dérivée de l'expression e par rapport à la variable x.

Exemples :

- `derive "x" (BinOp(Var "x",Add,Const 1))` va renvoyer `Const 1`
- `derive "x" (BinOp(Var "x",Mult,Var "x"))` va renvoyer

```
BinOp(BinOp(Const 1,Mult,Var "x"),
      Add,BinOp(Var "x",Mult,Const 1))
```

- `derive "y" (BinOp(Var "x",Mult,Var "x"))` va renvoyer `Const 0`

Corrigé

```
let rec derive x e = match e with
  | Const _ -> Const 0
  | Var y -> Const (if x = y then 1 else 0)
  | BinOp(e, Mult, e') ->
    BinOp(BinOp(derive x e, Mult, e'), Add, BinOp(e, Mult, derive x e'))
  | BinOp(e, Div, e') ->
    BinOp(
      BinOp(BinOp(derive x e, Mult, e'), Minus, BinOp(e, Mult, derive x e')),
      Div, BinOp(e', Mult, e'))
  | BinOp(e, op, e') ->
    BinOp(derive x e, op, derive x e')
  | UnOp(Neg, e) -> UnOp(Neg, derive x e)
  | UnOp(Inv, e) -> UnOp(Neg, BinOp(derive x e, Div, BinOp(e, Mult, e)))
```

Sans simplification, le premier exemple donne `BinOp(Const 1, Add, Const 0)` ; on obtient `Const 1` (et `Const 0` pour le troisième) en appliquant `simpl` au résultat.

On peut rajouter des symboles de fonctions élémentaires comme cos, sin … et adapter la dérivation formelle.

DEVOIR 61

Devoir maison - Lancer de rayons

61.1 DM — Lancer de rayons

Devoir maison en deux parties, mathématique puis informatique (classement provisoire : MP2I, année scolaire 2021-2022).

61.1.1 Préambule

Dans ce problème on se propose d'étudier les bases mathématiques nécessaires à réaliser des images par lancer de rayons.

Les données du problème sont :

- une description géométrique d'un certain nombre d'objets, *une scène*
- une caméra, caractérisée par sa position dans l'espace, la direction dans laquelle elle regarde et ses caractéristiques optiques (champ de vision, focale, largeur de pellicule).

Un tel problème est présenté dans la figure suivante où les objets sont des sphères.

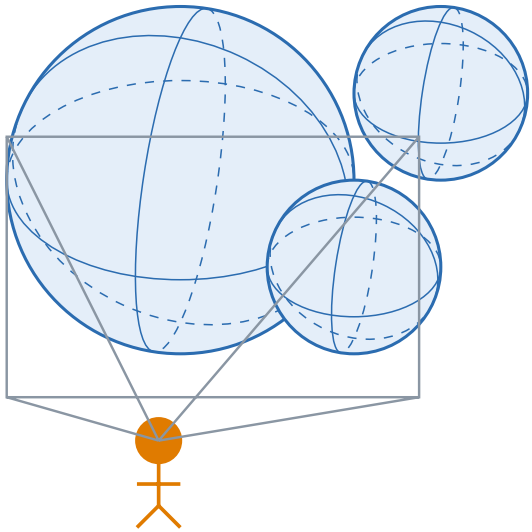


Figure 61.1 — Une scène de trois sphères et la caméra avec son champ de vision.

La réalisation d'une image consiste à retracer à l'envers le trajet d'un photon atterrissant dans l'objectif de la caméra. Pour une direction donnée, on considère donc une demi-droite issue de la caméra, appelée un *rayon* et on cherche à décider :

- si ce rayon intersecte un objet de la scène
- si c'est le cas quel est l'objet effectivement vu, c'est-à-dire celui pour lequel l'intersection est la plus proche de la caméra.

Dans la première partie mathématique du problème on s'intéresse à une simplification dans le plan. Dans une seconde partie on étend les résultats à l'espace puis on s'intéresse à l'étude de l'éclairage et des réflexions.

La partie informatique reprend ces résultats et demande de les réaliser sous forme de programme.

61.1.2 Mathématiques

Dans le plan

On considère une simplification du problème où les objets sont des disques, et la caméra est donnée par sa position $O\begin{vmatrix}x_0\\y_0\end{vmatrix}$, la direction $\vec{u}\begin{vmatrix}u_x\\u_y\end{vmatrix}$, de norme 1, de son regard, l'angle θ de son champ de vision, où $\theta \in]0, \pi$, sa distance f à sa pellicule appelée la *focale* et la *largeur l de sa pellicule*. Cette définition est illustrée dans la figure suivante.

![Caméra dans le plan : position O, direction $\vec{u}$, angle θ, focale f, pellicule de largeur l.{width=36%}

61.1.2.1.1 Expression des rayons

Question 1 Exprimer une relation liant l, θ et f.

Traditionnellement, on fixe l à une valeur constante (liée à la taille des pellicules de cinéma) et on donne uniquement le champ de vision de la caméra.

Question 2 Exprimer les coordonnées des deux extrémités P_1 et P_2 de la pellicule.

Soit $n \in \mathbb{N}^*$. On subdivise régulièrement la pellicule en n cases et on considère les rayons $r_1, ..., r_n$ passant par les centres $C_1, ..., C_n$ des cases. La figure suivante montre un exemple pour $n = 4$ cases. On appellera ces cases des *pixels*.

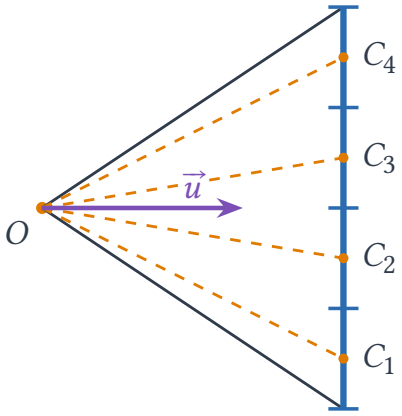


Figure 61.2 — Pellicule subdivisée en $n = 4$ pixels et rayons passant par leurs centres $C_1, ..., C_4$.

Question 3 Soit $k \in \{1, ..., n\}$, exprimer C_k comme barycentre de P_1 et P_2 et en déduire ses coordonnées ainsi que les coordonnées de la direction $\vec{u_k}$ du rayon correspondant.

61.1.2.1.2 Réalisation d'une image de synthèse

On réalise une image unidimensionnelle en attribuant à chaque pixel la couleur du disque dont l'intersection avec le rayon correspondant est la plus proche de la caméra, si une telle intersection n'existe pas on lui attribue la couleur blanche.

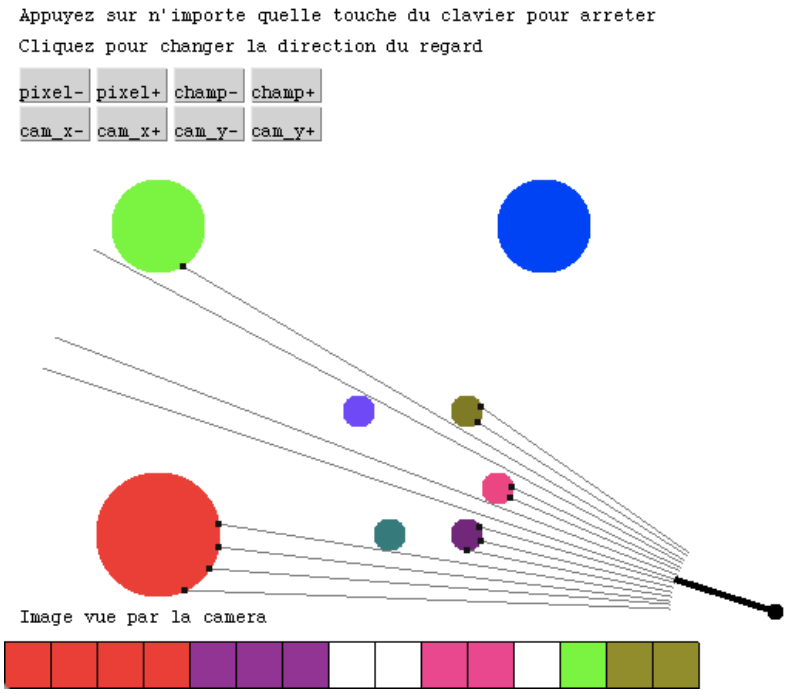


Figure 61.3 — Image unidimensionnelle

On est donc ramené à l'étude des intersections avec la scène d'un rayon.

61.1.2.1.3 Intersection avec la scène

On considère maintenant un rayon fixé r de départ $A\,|x_a\;y_a$ et de direction $\vec{v}\,|\alpha\;\beta$ et on cherche la plus proche intersection entre la demi-droite correspondante et la scène courante.

On considère donc l'équation paramétrique donnée par $\overrightarrow{AM} = t\vec{v}$ où $t \in \mathbb{R}^+$.

$M\,|x\;y$

Question 4 Si , en déduire x et y en fonction de x_a, y_a, α, β et t.

On identifie maintenant une intersection entre un rayon et un objet au $t_0 \in \mathbb{R}^+$ correspondant à ce point.

Question 5 Si on trouve deux intersections pour t_1 et t_2, comment peut-on caractériser celle qui est la plus proche de A ?

Question 6 Déterminer, en séparant les différents cas de figure, l'intersection entre r et le disque de centre $C\,|x_c\;y_c$ et de rayon R la plus proche de A.

Dans l'espace

On revient au problème initial où la scène est constituée de sphères dans l'espace. Ici la caméra est donnée

$\vec{v}\,|v_x\;v_y\;v_z$

avec deux paramètres supplémentaires : son vecteur unitaire déterminant la direction verticale de la pellicule et son *facteur d'aspect* a déterminant la hauteur h de la pellicule en fonction de sa largeur l. Par exemple un facteur d'aspect de $\frac{16}{9}$ signifie que $l = \frac{16}{9}h$. Un autre facteur d'aspect courant est le $\frac{4}{3}$.

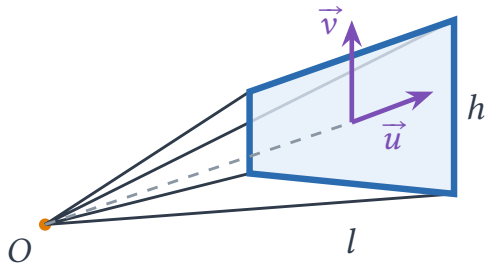


Figure 61.4 — Caméra dans l'espace : pellicule de dimensions $l \times h$, vecteurs $\vec{u}$ et $\vec{v}$.

61.1.2.2.1 Rappels

On rappelle que le produit scalaire dans $\mathbb{R}^3$ s'exprime ainsi : $z' = xx' + yy' + zz'$. De même la norme est $||\vec{v}|| = \sqrt{x^2 + y^2 + z^2}$ où z .

Si z et z' sont des vecteurs de l'espace leur produit vectoriel $\vec{w} = \vec{u} \wedge \vec{v}$ a pour coordonnées $xy' - x'y$.
Si $\vec{u}$ et $\vec{v}$ ne sont pas colinéaires, alors $\vec{u}, \vec{v}, \vec{w}$ est une base de $\mathbb{R}^3$. De plus, si $\vec{u}$ et $\vec{v}$ sont unitaires et orthogonaux, alors $\vec{w}$ est unitaire et le repère est orthonormé (direct si $(\vec{u}, \vec{v}) \equiv \frac{\pi}{2}[2\pi]$).

La sphère de centre C et de rayon R est constituée des points situés exactement à distance R de C. I.e. les points qui vérifient

$$(x - x_c)^2 + (y - y_c)^2 + (z - z_c)^2 = R^2$$

où z_c .

61.1.2.2.2 Expression des rayons

On exprimera ici les expressions en fonction de f, l, a, z_0 , u_z et v_z .

Question 7 Déterminer les coordonnées du point P en bas à gauche de la pellicule, ainsi que celles du vecteur $\vec{w}$ unitaire et dans la direction horizontale, vers la droite, de la pellicule.

Soit $w, h \in \mathbb{N}^*$. On subdivise régulièrement la pellicule en une grille de w cases horizontales et h cases verticales et on considère les rayons $r_{i,j}$ passant par les centres $C_{i,j}$ des cases. Une telle case est encore appelée un pixel, et le couple (i, j) représente les coordonnées du pixel.

Question 8 Soit (i, j) les coordonnées d'un pixel, exprimer les coordonnées de $C_{i,j}$ et du vecteur direction $\overrightarrow{u_{i,j}}$ correspondant.

61.1.2.2.3 Réalisation d'une image de synthèse

On réalise une image bidimensionnelle en attribuant à chaque pixel la couleur de la sphère comme dans le cas plan. Pour rendre les images plus réalistes, on ombre la couleur à l'aide d'un modèle rudimentaire d'éclairage,

de plus les sphères sont considérées comme étant semi-réfléchissantes. L'image suivante a été réalisée avec cette méthode :

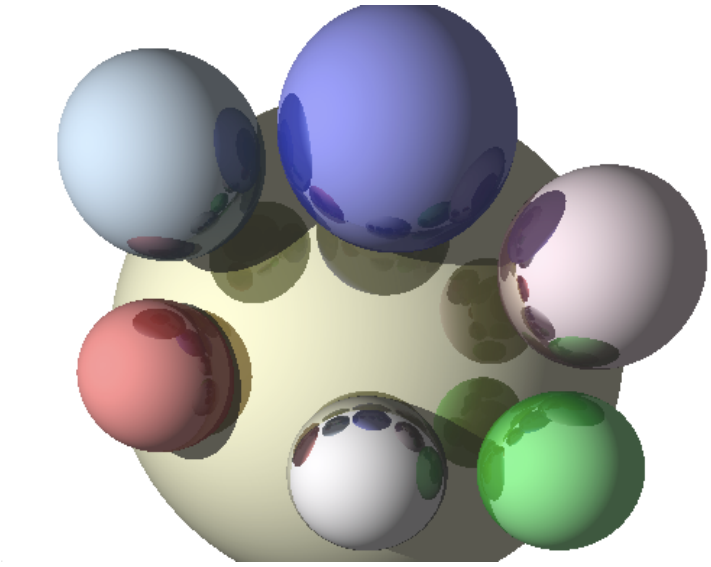


Figure 61.5 — Image de synthèse

61.1.2.2.4 Intersection avec la scène

$A\begin{vmatrix}x_a\\y_a\\z_a\end{vmatrix}$ $\vec{v}\begin{vmatrix}\alpha\\\beta\\\gamma\end{vmatrix}$

Comme précédemment, on considère un rayon fixé r de départ et de direction et on cherche la plus proche intersection entre la demi-droite correspondante et la scène courante.

Question 9 Déterminer, en séparant les différents cas de figure, l'intersection entre r et la sphère de centre $C\begin{vmatrix}x_c\\y_c\\z_c\end{vmatrix}$ et de rayon R la plus proche de A.

On se ramènera comme pour les disques à une équation du second degré en t.

61.1.2.2.5 Modèle d'éclairage de Phong

$L\begin{vmatrix}x_l\\y_l\\z_l\end{vmatrix}$ $M\begin{vmatrix}x\\y\\z\end{vmatrix}$

On considère une source lumineuse ponctuelle située en et émettant de manière isotrope (dans toutes les directions). Pour déterminer si un point est éclairé par L on lance un rayon issu de M en direction de L, appelé un *rayon d'ombrage*, et on teste s'il intersecte la scène.

$\vec{u_s}\begin{vmatrix}x_s\\y_s\\z_s\end{vmatrix}$

Question 10 Déterminer les coordonnées du vecteur direction unitaire du rayon d'ombrage.

Lorsque le rayon n'intersecte rien, le modèle de Phong permet d'exprimer l'intensité lumineuse reçue en M si la normale à la surface sur laquelle il se situe admet pour vecteur unitaire $\vec{n}\begin{vmatrix}x_n\\y_n\\z_n\end{vmatrix}$.

La formule, n'ayant aucune validité physique, est $I = \frac{\overrightarrow{AL}}{||\overrightarrow{AL}||} \cdot \vec{n}$.

$A\,|x_a$ $C\,|x_c$ $\vec{n}\,|n_x$
y_a y_c n_y

Question 11 Déterminer pour un point z_a d'une sphère de centre z_c, les coordonnées de n_z et en déduire une expression de I.

Ce qui suit n'est utile que pour la partie informatique.

De plus on rajoute souvent une composante ambiante permettant d'exprimer qu'un objet non éclairé directement n'est pas entièrement sombre. Ici nous supposerons que l'on a une couleur ambiante grise.

Pour le calcul explicite de la couleur, on suppose que notre objet a une couleur déterminée par trois composantes (c_r, c_v, c_b) pour rouge, vert et bleu comprises entre 0 et 1.

La couleur ambiante est alors (a_r, a_v, a_b).

On suppose que si l'objet voit la lumière, il reçoit un minimum de 0.2 en lumière même si I vaut moins. On $I_+ = \{0.2 \text{ si } I < 0.2$
pose donc I sinon.

Si l'objet est éclairé on aura alors comme couleur au point M :

$$(a_r + (1 - a_r)I_+c_r, a_v + (1 - a_v)I_+c_v, a_b + (1 - a_b)I_+c_b)$$

Sinon il n'aura que la couleur (a_r, a_v, a_b).

61.1.2.2.6 Réflexion

On considère un rayon r de départ A et de direction $\vec{u}$ unitaire, intersectant une sphère de centre C et de rayon R en un point B de normale $\vec{n}$. On cherche à déterminer la direction $\vec{u_r}$ du rayon réfléchi par la sphère. Pour obtenir la couleur d'une sphère réfléchissante il suffira alors de déterminer la couleur du rayon issu de B et de direction $\vec{u_r}$ (en fait il faut considérer tous les rebonds possibles car une seule direction de réflexion c'est une très grosse approximation).

La figure suivante présente un plan de coupe de la situation :

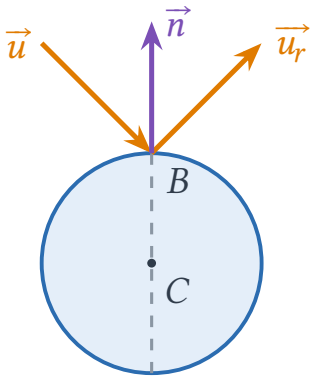


Figure 61.6 — Plan de coupe : réflexion du rayon $\vec{u}$ en B selon la normale $\vec{n}$.

Question 12 Déterminer $\vec{u_r}$ en fonction de $\vec{u}$ et $\vec{n}$.

61.1.3 Informatique

On trouvera différents fichiers à l'adresse http://marc.de-falco.fr/dmray.

- `dm2d.ml` reprend les résultats de la partie 2.1 sous la forme d'une interface. Vous pouvez étudier son code pour comprendre le fonctionnement et la mise en place de ces algorithmes.
- `dm3d.ml` présente un squelette de programme de calcul d'image par cette technique. Le but est de remplir les fonctions factices. Chaque fonction correspond à une des questions traitées dans la partie mathématique, celle-ci est indiquée en commentaires.

Voici quelques consignes pratiques :

- La note de votre DM sera fonction des résultats de votre programme.
- Vous pouvez vous mettre en groupe de 2 maximum.
- Vous présenterez votre programme lors d'une séance de TP. Vous ne pourrez pas faire illusion en prenant le code de quelqu'un d'autre, ce n'est même pas la peine d'essayer.

On procédera de manière incrémentale pour le calcul final de la couleur d'un rayon (c'est de loin la fonction la plus complexe).

Voici les étapes que je préconise : …

Voici des propositions d'extensions qui augmenteront votre note :

- Rajouter des plans infinis, par exemple pour faire un sol.
- Rajouter des triangles. En déduire une manière de faire des formes standards : cubes, pyramides, …
- Rajouter des motifs types quadrillage en plus des couleurs.
- Implémenter la loi de Descartes de la réfraction sur le modèle de la réflexion. Et, oui, on pourra avoir des matériaux semi-transparents et semi-réfléchissants. Au hasard, la vitre de votre chambre…
- Rajouter la possibilité de tracer des surfaces définies par une fonction : l'ensemble des points $M \begin{vmatrix} x \\ y \\ z \end{vmatrix}$ tels que $f(x, y, z) = 0$. Pour une sphère on aurait $x^2 + y^2 + z^2 - R^2 = 0$ par exemple.
- Essayez de faire en sorte que le ciel soit réaliste.
- Rajouter des Mandelbulbs (20/20 garanti pour la personne qui le fait).

La première échéance est la semaine du 19 mai. La version finale qui sera notée définitivement sera évaluée juste avant le conseil de classe.

PARTIE X

Devoirs 2022-2023 MP2I

62 Devoir DS 1 - Récursivité en OCaml 873
63 Devoir DS 2 - À la recherche du M-PIece 879
64 Devoir DS 4 - C et mémoire 897
65 Devoir DS 5 - Listes chaînées 905
66 Devoir DS 6 - Arbres n-aires en C 909
67 Devoir DS 7 - Sac à dos fractionnaire 919

DEVOIR 62

DS 1 - Récursivité en OCaml

62.1 DS 1 — Récursivité en OCaml

MP2I, année scolaire 2022-2023. Durée totale 2h.

Tout document interdit. Tout dispositif électronique interdit (calculatrice, cafetière, R2D2...)

On vous demande d'écrire lisiblement et de bien présenter. Les programmes en particulier doivent être bien indentés et lisibles.

Bon courage et *Don't panic!* 😃

62.1.1 Fonctions proches du TP

Attention

On utilise ici les types par défaut de `OCaml`. C'est particulièrement important pour les listes.

Quand on vous dit que $n \geq 0$, c'est acquis que n ne sera jamais négatif donc ne vous en préoccupez pas dans les questions et ne faites jamais de `if n < 0 then failwith...`.

Question 1 Écrire une fonction récursive `fact : int -> int` telle que `fact n` renvoie $n!$ où $n \geq 0$.

Corrigé

```
let rec fact n =
  if n = 0
  then 1
  else n * fact (n-1)
```

Question 2 Écrire une fonction `pow2 : int -> int` telle que `pow2 n` renvoie 2^n où $n \geq 0$ de manière naïve.

Corrigé

```
let rec pow2 n =
  if n = 0
  then 1
  else 2 * pow2 (n-1)
```

Question 3 Écrire une fonction `pow2_rap : int -> int` telle que `pow2_rap n` renvoie 2^n où $n \geq 0$ en remarquant que :

$$2^{2p} = (2^p)^2 \qquad 2^{2p+1} = 2 \times 2^{2p}$$

Corrigé

```
let rec pow2_rap n =
  if n = 0
  then 1
  else if n mod 2 = 1
  then 2 * pow2_rap (n-1)
  else let a = pow2_rap (n/2) in a * a
```

Attention ici, si on écrit `pow2_rap (n/2) * pow2_rap (n/2)` cela va faire deux appels bêtement et la complexité sera exponentielle en n.

Question 4 Écrire une fonction récursive `suplog2 : int -> int` telle que `suplog2 n` renvoie le plus petit $d \in \mathbb{N}$ tel que $0 \leq n < 2^d$ avec la contrainte suivante :

1. en utilisant `pow2`
2. sans utiliser `pow2`

Corrigé

1. En utilisant `pow2` :

```
let rec suplog2_aux n k =
  if pow2 k > n
  then k
  else suplog2_aux n (k+1)

let suplog2 n = suplog2_aux n 0
```

2. Sans utiliser `pow2` :

```
let rec suplog2 n =
  if n = 0
  then 0
  else 1 + suplog2 (n/2)
```

Question 5 Écrire une fonction `applique : ('a -> 'b) -> 'a list -> 'b list` telle que l'appel à `applique f l` renvoie la liste obtenue en appliquant à chaque élément de `l` la fonction `f`.

Corrigé

```
let rec applique f l =
  match l with
  | [] -> []
  | t::q -> f t :: applique f q
```

Question 6 Écrire une fonction `nieme : 'a list -> int -> 'a` telle que l'appel à `nieme l n` renvoie `x` où `x` est le n-ième élément de `l` et déclenche une erreur s'il n'y en a pas.

Corrigé

```
let rec nieme l n =
  match l, n with
  | [], _ -> failwith "Liste vide"
  | t::_, 0 -> t
  | _::q, _ -> nieme q (n-1)
```

62.1.2 Fonctions et typage

Question 7 Écrire une fonction `evalue` telle que `evalue f x` renvoie la valeur de la fonction `f` évaluée en `x`.

Attention la fonction est vraiment très simple.

Corrigé

```
let evalue f x = f x
```

Question 8 Quel est le type de la fonction précédente ?

Corrigé

```
('a -> 'b) -> 'a -> 'b
```

Question 9 Écrire une fonction `compose` telle que `compose f g x` calcule la valeur de $(f \circ g)(x)$ si on utilisait la notation mathématique (utiliser la notation OCaml revient ici à répondre à la question).

Corrigé

```
let compose f g x = f (g x)
```

Question 10 Quel est le type de la fonction précédente ?

Corrigé

```
('a -> 'b) -> ('c -> 'a) -> 'c -> 'b
```

62.1.3 Animal de compagnie

On considère ici le type suivant :

```
type pet = Cat | Dog
```

Question 11 Écrire une fonction `ennemy` qui renvoie l'animal opposé. Par exemple, `ennemy Cat` renvoie `Dog`.

Corrigé

```
let ennemy p = match p with
  | Dog -> Cat
  | Cat -> Dog
```

On dit que l'age, converti en années humaines, d'un chien est de 7 fois son age réel.

Question 12 Écrire une fonction `dog_years` qui prend en entrée l'age réel d'un chien et renvoie son age converti.

Corrigé

```
let dog_years n = n * 7
```

On dit que l'age, converti en années humaines, d'un chat est :

- de 12 ans si le chat a 1 an
- de 24 ans si le chat a 2 ans
- et chaque année supplémentaire à partir de la troisième compte comme 5 années humaines

Ainsi, un chat de 4 ans a $24 + 2 \times 5 = 34$ années humaines.

Question 13 Écrire une fonction `cat_years` qui prend en entrée l'age réel d'un chat et renvoie son age converti.

Corrigé

```
let cat_years n =
  match n with
  | 1 -> 12
  | _ -> 24 + 5 * (n-2)
```

Question 14 En déduire une fonction `pet_years : pet -> int -> int` qui calcule l'age converti d'un animal selon sa race.

Corrigé

```
let pet_years p n =
  match p with
  | Cat -> cat_years n
  | Dog -> dog_years n
```

Notons qu'ici on peut omettre `n` :

```
let pet_years p =
  match p with
  | Cat -> cat_years
  | Dog -> dog_years
```

Cela revient à imaginer qu'on calcule une fonction qui pourra être utilisée dans tous les cas avec un entier.

62.1.4 Couvertures

On considère une ligne de n cases carrées :



que l'on souhaite recouvrir complètement avec des tuiles de différentes longueurs.

Par exemple, si on dispose de tuiles de longueur 1 ou 2, voici une couverture valide avec deux tuiles de longueur 1 et deux de longueur 2.



Question 15 Écrire une fonction OCaml `couvertures12 : int -> int` telle que `couvertures12 n` calcule le nombre de couvertures de n cases avec des tuiles de longueurs 1 ou 2.

Indication : on pourra raisonner sur la tuile la plus à gauche d'une telle couverture.

Corrigé

Dans le cas dégénéré $n = 0$ il n'y a qu'une couverture : c'est l'ensemble vide. Dans le cas $n = 1$ on ne peut placer qu'une tuile de taille 1.

Sinon on place une tuile de taille 1 et on couvre les $n - 1$ restants ou on place une tuile de taille 2 et on couvre les $n - 2$ restants.

```
let rec couvertures12 n =
  if n = 0 || n = 1
  then 1
  else couvertures12 (n-1) + couvertures12 (n-2)
```

Question 16 En relisant la fonction précédente, reliez ce nombre de couvertures à une suite classique.

Corrigé

Il s'agit de la suite de Fibonacci !

Question 17 Montrer qu'il est équivalent de couvrir n cases ainsi et de couvrir un échiquier de $2 \times n$ cases avec des dominos.

Rappel : un domino est un rectangle de deux cases de long, on peut donc le placer verticalement ou horizontalement pour couvrir deux cases de l'échiquier.

Corrigé

Pour couvrir un échiquier $2 \times n$ on a le choix sur la position du premier domino : vertical ou horizontal. Si on le place verticalement, il ne reste comme espace libre que de quoi placer un domino parallèle exactement à côté. Le choix que l'on fait est donc entre un domino horizontal ou deux dominos verticaux. C'est exactement le même choix qu'avec les tuiles précédentes. D'ailleurs on passe de l'un à l'autre en coupant l'échiquier en deux.

Question 18 Déterminer une condition sur n et m pour qu'un échiquier de $n \times m$ cases puisse être recouvert par des dominos.

Corrigé

Si on colorie l'échiquier comme aux échecs, chaque domino est à cheval sur une case blanche et une case noire. Il faut donc que le nombre de cases blanches de l'échiquier soit égal au nombre de cases noires.

Si $n = 2p + 1$, les lignes sont tantôt à p cases noires et $p + 1$ cases blanches, tantôt l'inverse. Pour que cela s'équilibre, il faut que le nombre de lignes soit pair.

Si n est pair, l'équilibre est automatique.

La condition est donc d'avoir n ou m pair. Dans ce cas on peut découper l'échiquier en bandes de tailles $2 \times m$ ou $n \times 2$ et on sait que l'échiquier restant est couvrable (par exemple uniquement avec des dominos parallèles).

On a donc : $n \times m$ admet une couverture $\Leftrightarrow$ nm pair (car n ou m pair $\Leftrightarrow$ nm pair).

Question 19 **(difficile)** Écrire une fonction OCaml `couvertures : int list -> int -> int` telle que `couvertures l n` calcule le nombre de couvertures de n cases avec des tuiles de longueur données par la liste l.

Par exemple, la fonction précédente correspondra à `couvertures [1;2]`.

On ne suppose rien sur l. Si un nombre est présent plusieurs fois, c'est comme si on considérait des tuiles de même longueur mais de couleurs différentes, ce qui augmente le nombre de couvertures.

Corrigé

```
let rec couvertures tl n =
  let rec aux l = match l with
    | [] -> 0
    | t::q -> (if n >= t then couvertures tl (n-t) else 0) + aux q
  in if n = 0 then 1 else aux tl
```

Question 20 Calculer le nombre de couvertures pour la liste `[1;1;1;1]` qui contient donc quatre fois une tuile de longueur 1 pour quatre couleurs différentes.

Corrigé

Ici on a quatre tuiles identiques de longueur 1, donc on a quatre choix par case. Ainsi, s'il y a n cases alors on a 4^n couvertures distinctes.



Figure 62.1 — Mème sur la récursivité

DEVOIR 63

DS 2 - À la recherche du M-PIece

63.1 DS 2 — Correction de programmes, C et OCaml

MP2I, année scolaire 2022-2023. Durée totale 4h.

Tout document interdit. Tout dispositif électronique interdit (calculatrice, cafetière, Francky…)

On vous demande d'écrire lisiblement et de bien présenter. Les programmes en particulier doivent être bien indentés et lisibles.

Bon courage et *à l'abordage !* 🏴‍☠️

À la recherche du M-PIece



Figure 63.1 — Le bateau

Résumé des épisodes précédents

*À bord de votre bateau, le **Coding Merry**, vos coéquipiers et vous, Camel C. Civvy, êtes partis en quête du trésor légendaire : le M-PIece ! Dans ce sujet, vous allez résoudre des problèmes pour y arriver. Et, qui sait, ce trésor se trouve peut-être au CIV.*

Ce problème est constitué de quatre parties indépendantes.

Remarque sur le corrigé

Les preuves de correction ou de terminaison présentées ici sont un objectif de rédaction à atteindre. Des réponses partielles ou moins détaillées sont acceptées.

63.1.1 Passer en revue l'équipage

Avant de partir, c'est le moment de passer en revue l'équipage et de s'assurer que tout le monde est qualifié.

Plusieurs membres de l'équipage vous montrent ainsi des programmes qu'ils ont réalisés et vous allez devoir comprendre ce qu'ils font et montrer qu'ils sont corrects.

On considère le programme `C` suivant :

```c
bool palindrome(char str[]){
    int i = 0,
    j = 0;
    while (str[j] != '\0') {
        j = j + 1;
    }
    j = j - 1;
    while (j >= i) {
        if (str[i] != str[j]) {
            return false;
        }
        i = i+1;
        j = j-1;
    }
    return true;
}
```

On suppose que `str` sera toujours une chaîne de caractère valide en `C`.

Question 1 1. Dérouler l'exécution de ce programme sur les chaînes `"kayak"` et `"canoe"`. 2. Montrer que toutes les boucles terminent. 3. Montrer que ce programme est correct, c'est-à-dire qu'il renvoie un booléen indiquant si une chaîne est un palindrome.

Indication : On rappelle qu'un *palindrome* est une chaine de caractère pouvant se lire de la même manière de gauche à droite et de droite à gauche.

Corrigé

1. Pour `"kayak"`, première boucle :

j	`str[j]`
0	`'k'`
1	`'a'`
2	`'y'`
3	`'a'`
4	`'k'`
5	0

ensuite on passe j à 4 et on entre dans la seconde boucle.

i	j	`str[j]`	`str[i]`
0	4	`'k'`	`'k'`
1	3	`'a'`	`'a'`
2	2	`'y'`	`'y'`
3	1		

Comme on n'est pas sorti avant, on renvoie `true`.

Pour `"canoe"`, première boucle :

j	`str[j]`
0	`'c'`
1	`'a'`
2	`'n'`
3	`'o'`
4	`'e'`
5	0

ensuite on passe j à 4 et on entre dans la seconde boucle.

i	j	`str[j]`	`str[i]`
0	4	`'e'`	`'c'`

Ici, on sort tout de suite en renvoyant `false`.

2. Comme la chaine est valide, on sait que le caractère `'\0'` est présent à un indice n. On a alors la quantité $n - j$ qui est un variant de la première boucle car elle est :

   - entière ;
   - positive tout au long de la boucle vu que la condition implique $j < n$ quand on boucle ;
   - diminue strictement à chaque itération car j augmente.

   Ainsi la première boucle termine.

   La seconde termine car la quantité $j - i$ est un variant :

   - elle est entière ;
   - positive tout au long de la boucle car la condition $i \leq j$ implique $j - i \geq 0$;
   - diminue strictement à chaque itération car $j - 1 - (i + 1) = j - i - 2 < j - i$.

3. La première boucle termine et en sortie on a $j = n$ où n est la première occurrence de `'\0'` dans la chaine, c'est-à-dire sa longueur.

   On va montrer que la propriété

   $$I(i, j) = n - i - 1 = j \wedge \forall k \in [\![0, i-1]\!], s[k] = s[n-k-1]$$

   est un invariant.

   - Initialisation : au départ, $i = 0$ donc la plage $[\![0, i-1]\!] = \emptyset$ et l'invariant $I(0, n-1)$ est vérifié.
   - Hérédité : si $I(i, j)$ est vérifié en début d'itération et qu'on ne sort pas par le `return false`, on a $s[i] = s[j] = s[n-1-i]$ donc la propriété d'égalité est vérifiée pour $k = i$, on a donc $\forall k \in [\![0, i]\!]$. De plus, $n - (i+1) - 1 = n - i - 1 - 1 = j - 1$ donc $I(i+1, j-1)$ est vérifié.

   Si on sort par le `return false`, c'est que $s[i] \neq s[j] = s[n-1-i]$ donc la chaîne n'est pas un palindrome : le résultat est correct.

   Sinon, l'invariant est validé et en sortie de boucle, on a $n - i - 1 = j < i \Leftrightarrow i > \frac{n-1}{2} \geq \lfloor \frac{n-1}{2} \rfloor$ donc $i \geq \lfloor \frac{n}{2} \rfloor$ et l'égalité assure ainsi que la première moitié se retrouve à l'envers dans la seconde moitié : c'est un palindrome.

On considère le programme `OCaml` suivant :

```
let div a b =
  let q = ref 0 in
  let r = ref a in
  while !r >= b do
```

```
    r := !r - b;
    q := !q + 1
  done;
  (!q, !r)
```

On suppose ici que a, b sont des entiers naturels et que $b > 0$.

Question 2 1. Dérouler l'exécution de ce programme pour $a = 42$ et $b = 4$. 2. Montrer que ce programme termine. 3. Montrer que le couple (q, r) renvoyé vérifie $a = bq + r$ et $0 \leq r < b$.

Corrigé

1. Valeurs successives :

q	r
0	42
1	38
2	34
3	30
4	26
5	22
6	18
7	14
8	10
9	6
10	2

   La fonction renvoie donc `(10, 2)`.

2. La quantité r est un variant :
   - elle est entière (uniquement des soustractions) ;
   - elle est positive tout au long de la boucle car si on rentre dans une itération c'est que $r \geq b$, donc en fin d'itération $r - b \geq 0$ et c'est la nouvelle valeur de r ;
   - comme $b > 0$, elle diminue strictement à chaque itération.

   Ainsi, comme elle admet un variant, la boucle termine.

3. On a l'invariant $I(q, r) \equiv a = bq + r$:
   - Initialisation : $a = b \times 0 + a$ donc $I(0, a)$ est vérifié.
   - Itération : si $I(q, r)$ est vérifié en début d'itération, alors $a = bq + r = b(q + 1) + r - b$ donc $I(q + 1, r - b)$ est vérifié en fin d'itération ce qui correspond aux nouvelles valeurs de (q, r).

   Ainsi, en sortie de boucle, on a $a = bq + r$ et comme on est sorti $r < b$. De plus, on a vu à la question précédente que $r \geq 0$.

63.1.2 Trouver la bonne vitesse - C

Afin de planifier la route, votre navigatrice, Cami, étudie les données qu'elle a *empruntées* à d'autres équipages. Elle a ainsi obtenu une suite de n nombres entiers indiquant la vitesse du bateau à intervalle régulier.

Par exemple, on aurait pu obtenir cette suite exemple de 10 nombres :

199 200 208 210 200 207 240 269 260 263

Dans toutes les questions, on suppose que cette suite est donnée sous la forme d'un tableau d'entiers du type `int v[]` et du nombre `int n` de vitesses enregistrées.

En premier lieu, on va s'intéresser au nombre de moments où le bateau a accéléré, indiqué ici avec une flèche ↗ :

$$199 \nearrow 200 \nearrow 208 \nearrow 210 \searrow 200 \nearrow 207 \nearrow 240 \nearrow 269 \searrow 260 \nearrow 263$$

On voit sur cet exemple que le bateau a accéléré 7 fois.

Attention : on considère qu'on accélère quand la vitesse augmente **strictement**.

Question 3 Écrire une fonction de prototype

```
int accelerations(int v[], int n);
```

qui prend un tableau de vitesse en entrée et renvoie le nombre de fois où le bateau a accéléré.

Corrigé

```
int accelerations(int v[], int n) {
    int accel = 0;
    for(int i = 0; i < n-1; i++) {
        if (v[i] < v[i+1]) {
            accel = accel + 1;
        }
    }
    return accel;
}
```

Question 4 Prouvez la correction de votre fonction.

Indication : il sera sûrement nécessaire de s'assurer qu'à tout moment, vous avez compté le nombre de fois où le bateau a accéléré **dans la partie étudiée** du tableau.

Corrigé

Conformément à l'indication, on remarque que `accel` contient toujours le nombre d'accélérations du préfixe étudié du tableau.

On considère ainsi le candidat invariant $I(\texttt{accel}, i) \equiv$ le nombre d'accélérations dans $v[0], ..., v[i]$ est `accel`. On va le valider :

- Initialisation : au départ, $i = 0$ donc le préfixe contient un seul élément, il ne peut être question d'accélération, et comme initialement $\texttt{accel} = 0$, on a bien $I(\texttt{accel}, i)$.
- Itération : si $I(\texttt{accel}, i)$ est vérifié en début d'itération, on a deux cas :
  - $v[i] \geq v[i+1]$, alors il n'y pas de nouvelle accélération et comme on ne change pas `accel`, on a bien $I(\texttt{accel}, i+1)$.
  - $v[i] < v[i+1]$, on a alors une accélération de plus et ainsi on a bien $I(1 + \texttt{accel}, i+1)$.

  Dans tous les cas, I est vérifié en fin d'itération.

Ainsi, en sortie de boucle $I(\texttt{accel}, n-1)$ est vérifié donc `accel` contient les accélérations du tableau complet.

Question 5 Écrire une suite d'instructions qui pourraient être placées dans la fonction `main` d'un programme pour :

- définir le tableau de l'exemple
- calculer le nombre d'accélérations `N` avec la fonction précédente
- afficher une ligne indiquant `Le bateau a acceleré N fois` où `N` est affiché sous forme décimale.

Corrigé

```
int v[] = { 199, 200, 208, 210, 200, 207, 240, 269, 260, 263 };
int N = accelerations(v, 10);
printf("Le bateau a acceleré %d fois.\n", N);
```

Dans l'exemple précédent, on peut voir que le bateau accélère au plus trois fois d'affilée avant de ralentir, cela est réalisé deux fois : 199 ↗ 200 ↗ 208 ↗ 210 et 200 ↗ 207 ↗ 240 ↗ 269. Une série d'accélérations sera appelée ici une **poussée**.

Question 6 Écrire une fonction de prototype

```
int poussee(int v[], int n, int i);
```

où v et n décrivent un tableau vitesse comme précédemment, $0 \leq i < n$ et qui renvoie le nombre de fois où le bateau a accéléré à partir de la vitesse $v[i]$.

Dans l'exemple précédent, pour $i = 0$ on renverra 3, pour $i = 2$ on renverra 1 et pour $i = 3$ on renverra 0.

Corrigé

```
int poussee(int v[], int n, int i) {
    int j = i;
    while(j < n-1 && v[j] < v[j+1]) {
        j += 1;
    }
    return j-i;
}
```

Question 7 Prouver la correction de votre fonction, c'est-à-dire que si elle renvoie N pour i alors la suite de valeurs $v[i], v[i+1], ..., v[i+N]$ est croissante et, soit $i + N + 1 = n$ soit $v[i+N] \geq v[i+N+1]$.

Attention il sera sûrement nécessaire ici de s'assurer de la terminaison.

Corrigé

On remarque rapidement que $n - j$ est un variant, donc la boucle termine.

On va valider l'invariant $I(j) \equiv \forall k \in [\![i, j-1]\!], v[k] < v[k+1]$.

- Initialisation : au départ $j = i$, donc $[\![i, j-1]\!] = \emptyset$ et $I(j)$ est trivialement vérifié.
- Itération : si on a $I(j)$ en début d'itération, comme on est resté dans la boucle, c'est que $j < n - 1$ et $v[j] < v[j+1]$, comme on sait que

$$\forall k \in [\![i, j-1]\!], v[k] < v[k+1]$$

on en déduit

$$\forall k \in [\![i, j]\!], v[k] < v[k+1]$$

donc $I(j+1)$ est vérifié ce qui correspond à la nouvelle valeur de j.

Ainsi, c'est bien un invariant et en sortie de boucle on a

$$\forall k \in [\![i, j-1]\!], v[k] < v[k+1]$$

En notant $N = j - i$, on a alors

$$\forall k \in [\![i, i+N-1]\!], v[k] < v[k+1]$$

et,

- soit $j = n - 1 \Leftrightarrow i + N + 1 = n$
- soit $v[j] = v[i + N] \geq v[i + N + 1] = v[j + 1]$.

Cela correspond donc à la spécification donnée par la question.

Question 8 En déduire une fonction de prototype

```
int plus_grande_poussee(int t[], int n);
```

qui renvoie la plus grande valeur de la poussée.

Attention votre fonction devra appeler la fonction précédente.

Corrigé

```
int plus_grande_poussee(int t[], int n) {
    int m = 0;
    for(int i = 0; i < n-1; i++) {
        int v = poussee(t, n, i);
        if (v > m) {
            m = v;
        }
    }
    return m;
}
```

Question 9 Prouvez la correction de la fonction précédente en supposant la fonction `poussee` correcte.

Corrigé

On remarque qu'il n'est pas question de terminaison vu qu'il s'agit d'une boucle inconditionnelle.

On va valider l'invariant $I(m, i) \equiv \forall k \in [\![0, i-1]\!], m \geq \texttt{poussee}\ (t, n, k)$.

- Initialisation : au départ $m = 0$ et $i = 0$, donc la plage est vide et la propriété est vérifiée. Ainsi, $I(0, 0)$ est vérifiée.
- Itération : si $I(m, i)$ est vérifié en début d'itération, on a deux cas, soit $\texttt{poussee}\ (t, n, i) \leq m$ et m ne change pas, soit $\texttt{poussee}\ (t, n, i) > m$ et m devient $\texttt{poussee}\ (t, n, i)$. Dans tous les cas la valeur en fin d'itération de m est plus grande que tous les $\texttt{poussee}\ (t, n, k)$ pour $k \in [\![0, i]\!]$.

Ainsi, en sortie de boucle, on a $I(m, n-1)$ vérifié, et comme $\texttt{poussee}\ (t, n, n-1) = 0 \leq m$, m est plus grand que toutes les valeurs $\texttt{poussee}\ (t, n, k)$.

De plus, m vaut toujours 0 ou l'une des valeurs $\texttt{poussee}\ (t, n, k)$: comme $0 = \texttt{poussee}\ (t, n, n-1)$, m est une valeur atteinte, c'est donc bien le maximum.

> **Initialiser le maximum courant pour $i = 0$ ou avec une valeur par défaut ?** Ici, on a fait le choix de l'initialiser à 0 car on sait que la poussée est dans $\mathbb{N}$. Mais quand on est dans $\mathbb{Z}$, comme c'est non minoré, il n'y a pas de bonne valeur pour initialiser, et dans ce cas, on place la valeur pour le premier indice comme valeur initiale.
>
> Par contre, cela complique la preuve de correction, donc, quand on peut l'éviter, comme ici, on se contente d'initialiser avec un minimum absolu.

Question 10 Justifiez rapidement que si le tableau comporte n valeurs, alors la fonction précédente effectue de l'ordre de n^2 opérations élémentaires (affectations, accès mémoire, opérations arithmétiques ou de comparaison).

Corrigé

La fonction `poussee` effectue une boucle d'au plus $n-i$ itérations effectuant chacune des opérations élémentaires. Ainsi, chaque appel fait au maximum de l'ordre de n opérations élémentaires. Comme `plus_grande_poussee` effectue $n-1$ appels à `poussee` on a de l'ordre n^2 opérations élémentaires.

Plutôt que la vitesse instantanée, on va s'intéresser maintenant à la vitesse moyenne sur une suite de mesures, ce qui revient à s'intéresser aux sommes de vitesses. On parle de *largeur de fenêtre* pour le nombre de mesures que l'on considère.

Ainsi, sur l'exemple précédent et avec une largeur de fenêtre de 3, la première somme vaut $199+200+208=607$ et la seconde 618. Il y avait 10 valeurs dans cet exemple, mais avec une fenêtre de largeur 3, on ne peut calculer que 8 sommes, on obtient ainsi la suite de sommes suivantes :

$$607 \quad 618 \quad 618 \quad 617 \quad 647 \quad 717 \quad 769 \quad 792$$

On peut alors considérer de même les accélérations entre ces sommes, moments où la somme augmente **strictement**. Pour l'exemple on a donc :

$$607 \nearrow 618 = 618 \searrow 617 \nearrow 647 \nearrow 717 \nearrow 769 \nearrow 792$$

Question 11 Écrire une fonction de prototype

```
int accelerations_fenetre(int v[], int n, int k);
```

qui prend en entrée un tableau de vitesses donné par v et n ainsi qu'une largeur de fenêtre k et renvoie le nombre d'accélérations associé.

Exemple : pour $k=3$ et les vitesses de l'exemple précédent, votre fonction renverra la valeur 5.

Indication : il n'y a pas besoin de stocker les sommes de fenêtre dans un tableau, on peut tout faire uniquement avec un nombre fini de variables locales entières.

Corrigé

```
int accelerations_fenetre(int v[], int n, int k) {
    if (k+1 > n) // pas de fenetres
        return 0;
    int s = 0;
    for(int i = 0; i < k; i++) {
        s = s + v[i];
    }
    int accel = 0;
    for(int j = 0; j < n-k; j++) {
        int s2 = s - v[j] + v[j+k];
        if(s2 > s) {
            accel += 1;
        }
        s = s2;
    }
    return accel;
}
```

Question 12 Prouvez soigneusement la correction de votre fonction.

Corrigé

Il est facile de se tromper d'un cran dans la fonction précédente. Il y a un terme consacré pour ces problèmes : *off by one*. Ici, faire la preuve de correction permet d'être certain de ne pas s'être trompé.

La fonction est directement correcte quand il y a 0 ou 1 fenêtre uniquement, car on renvoie 0 et qu'il y a bien 0 accélération possible.

On remarque rapidement que la structure du compte est la même que dans la fonction `accelerations` donc, il suffit de vérifier que `s` contient bien la somme de la fenêtre.

On suppose assez rapidement que la valeur initiale est $s = \sum_{i=0}^{k-1} v[i]$.

On va ainsi montrer que la propriété $I(j) \equiv s = \sum_{i=j}^{j+k-1} v[i]$ est un invariant.

Avant la boucle, $j = 0$ et on a supposé $I(0)$. Si en début d'itération, on a $I(j)$, alors $s = \sum_{i=j}^{j+k-1} v[i]$ donc $s2 = s - v[j] + v[j+k] = \sum_{i=j+1}^{j+k} v[i]$ et ainsi la valeur de s en fin d'itération vérifie bien $I(j+1)$.

En sortie de boucle, on a $I(n-k)$ vérifié donc $s = \sum_{i=n-k}^{n-1} v[i]$ est la dernière somme de fenêtre considérée et c'est bien la dernière fenêtre.

63.1.3 Planifier sa route - OCaml

Pour votre route, vous devrez passer d'île en île. Afin de passer à une île suivante, il est nécessaire d'attendre une certaine durée pour que votre *log2pos*, une sorte de boussole, puisse vous indiquer la direction de la prochaine île.

On vous a fourni une `int list` contenant les temps qu'il faut attendre pour chaque île.

Question 13 Écrire une fonction de type :

```
temps_total : int list -> int
```

qui détermine le temps total qu'il faudra attendre.

Corrigé

```
let rec temps_total l =
  match l with
  | [] -> 0
  | t::q -> t + temps_total q
```

Question 14 Écrire une fonction de type :

```
plus_dattente : int list -> int
```

qui détermine l'indice dans la liste de l'île où il faudra attendre le plus longtemps.

Corrigé

Cette fonction est plutôt délicate à écrire en récursif car on a besoin de beaucoup de variables temporaires qui deviennent des paramètres de fonctions auxiliaires.

```
let plus_dattente l =
  let rec aux l max_courant ind_max_courant indice_actuel =
    match l with
    | [] -> ind_max_courant
    | t::q ->
      if t > max_courant
      then aux q t indice_actuel (indice_actuel+1)
      else aux q max_courant ind_max_courant (indice_actuel+1)
  in match l with
  | [] -> failwith "Vide"
  | t::q -> aux q t 0 1
```

Une bonne manière de programmer une telle fonction est de penser à une version impérative et à transformer l'itération en une fonction : valeurs avant → valeurs après.

Ainsi, en Python, on aurait pu écrire :

```python
def plus_dattente(l):
    max_courant = l[0]
    ind_max_courant = 0

    for indice_actuel in range(1, len(l)):
        t = l[indice_actuel]
        if t > max_courant:
            max_courant = t
            ind_max_courant = indice_actuel

    return ind_max_courant
```

Il se trouve que le temps à attendre n'est qu'une approximation grossière de votre traversée car il faut prendre en compte le trajet entre chaque île.

On considère alors le type suivant :

```
type donnee_nav =
    Trajet of int | Ile of int
```

Votre route est alors une liste alternant entre valeurs `Trajet` et valeurs `Ile`.

Question 15 Écrire une fonction de type :

```
alternante : donnee_nav list -> bool
```

qui indique si la liste alterne ainsi.

Corrigé

On peut écrire cette fonction de plusieurs manières. Avancer en regardant deux éléments est sûrement le plus simple.

```
let rec alternante l =
  match l with
  | [] | [_] -> true
  | Trajet _ :: Ile v :: q -> alternante (Ile v :: q)
  | Ile _ :: Trajet v :: q -> alternante (Trajet v :: q)
  | _ -> false
```

On pouvait aussi faire deux fonctions auxiliaires mutuellement récursives permettant de distinguer les cas où on commence avec une île ou un trajet.

```
let alternante l =
  let rec ile_alt l =
    match l with
    | [] -> true
    | Trajet _ :: q -> false
    | Ile _ :: q -> trajet_alt q
  and trajet_alt l =
    match l with
    | [] -> true
    | Ile _ :: q -> false
    | Trajet _ :: q -> ile_alt q
  in ile_alt l || trajet_alt l
```

Question 16 Écrire une fonction de type :

```
ile_depart_fin : donnee_nav list -> bool
```

qui indique si la liste commence et finit par une `Ile`.

Corrigé

```
let ile_depart_fin l =
  let rec dernier_ile l =
    match l with
    | [] -> false
    | [Ile _] -> true
    | _::q -> dernier_ile q in
  match l with
  | Ile _ :: _ -> dernier_ile l
  | _ -> false
```

Question 17 Écrire une fonction de type :

```
temps_total_nav : donnee_nav list -> int
```

qui détermine le temps total, trajet et attente, de la route.

Corrigé

```
let rec temps_total_nav l =
  match l with
  | [] -> 0
  | Trajet t :: q | Ile t :: q ->
    t + temps_total_nav q
```

Il se trouve que votre route n'est pas forcément linéaire : il peut y avoir des choix entre plusieurs îles.

On considère alors le type suivant :

```
type route =
  | Finie
  | Simple of donnee_nav * route
  | Double of (donnee_nav * route)
              * (donnee_nav * route)
```

Un couple du type `donnee_nav * route` représente ainsi un bout du parcours, soit d'une île, soit d'un trajet, et on continue la route. On a alors une route qui est soit finie, soit simple quand il n'y a pas de choix, soit double quand on a le choix entre deux parcours.

Par exemple, la valeur :

```
Simple(Ile 3,
  Double(
    (Trajet 3,
      Simple(Ile 2,
        Double((Trajet 2, Simple(Ile 1, Finie)),
          (Trajet 3, Simple(Ile 2, Finie))))),
    (Trajet 1, Simple(Ile 5, Finie))))
```

représente la route suivante :

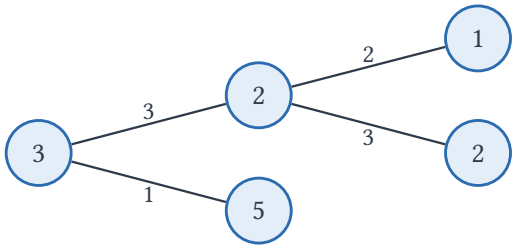


Figure 63.2 — La route représentée : îles dans les cercles, numéros de trajet sur les arêtes.

Question 18 Écrire une fonction de type

```
temps_total_route : route -> int
```

qui renvoie la somme des temps de l'ensemble de la route.

Corrigé

```
let rec temps_total_route l =
  match l with
  | Finie -> 0
  | Simple(Trajet t, q)
  | Simple(Ile t, q) ->
    t + temps_total_route q
  | Double(r1,r2) ->
    let aux c =
      match c with
      | (Ile t, r)
      | (Trajet t, r) -> t + temps_total_route r
    in aux r1 + aux r2
```

Question 19 Écrire une fonction de type

```
alternant_route : route -> bool
```

qui détermine si une route alterne toujours comme sur l'exemple, entre île et trajet.

Corrigé

Ici, la version avec récursivité croisée semble plus intéressante.

```
let alternant_route l =
  let rec alt_ile l =
    match l with
    | Finie -> true
    | Simple(Ile _, r) -> alt_trajet r
    | Double( (Ile _, r1), (Ile _, r2) ) ->
      alt_trajet r1 && alt_trajet r2
    | _ -> false
  and alt_trajet l =
    match l with
    | Finie -> true
    | Simple(Trajet _, r) -> alt_ile r
    | Double( (Trajet _, r1), (Trajet _, r2) ) ->
      alt_ile r1 && alt_ile r2
    | _ -> false
  in alt_ile l || alt_trajet l
```

Question 20 Écrire une fonction de type

```
temps_min_route : route -> int
```

qui renvoie le temps du chemin le plus court dans cette route.

Indication : sur l'exemple, on constate que le chemin le plus court est donné par le chemin du bas `Ile 3 - Trajet 1 - Ile 5` pour une durée de 9.

Corrigé

```
let rec temps_min_route l =
  match l with
  | Finie -> 0
```

```
  | Simple(Trajet t, q)
  | Simple(Ile t, q) ->
    t + temps_min_route q
  | Double(r1,r2) ->
    let aux c =
      match c with
      | (Ile t, r)
      | (Trajet t, r) -> t + temps_min_route r
    in min (aux r1) (aux r2)
```

Question 21 Écrire une fonction de type

```
est_chemin : route -> donnee_nav list -> bool
```

qui vérifie qu'un chemin donné sous la forme d'une liste de données de navigation est dans la route en partant du départ et en finissant complètement.

Indication : dans la route précédente, on peut constater que `[Ile 3; Trajet 3; Ile 2; Trajet 2; Ile 1]` est un chemin de la route, mais pas `[Ile 3; Trajet 1]` (pas finie) ou `[Ile 3; Trajet 4]` (données non présentes).

Corrigé

```
let rec est_chemin r l =
  match r, l with
  | Finie, [] -> true
  | Simple(Ile a, r'), Ile b::q when a = b
    -> est_chemin r' q
  | Simple(Trajet a, r'), Trajet b::q when a = b
    -> est_chemin r' q
  | Double((d1, r1), (d2, r2)), d::q ->
    (d1 = d && est_chemin r1 q) || (d2 = d && est_chemin r2 q)
  | _ -> false
```

Note : dans le cas `Double`, il faut bien essayer les deux branches (avec un `||`) : si elles commencent par la même donnée, le chemin peut se trouver dans la seconde.

Question 22 Écrire une fonction de type

```
chemins : route -> donnee_nav list list
```

qui renvoie la liste de tous les chemins présents dans la route.

Indication : pour l'exemple, cette fonction va renvoyer :

```
[[Ile 3; Trajet 3; Ile 2; Trajet 2; Ile 1];
 [Ile 3; Trajet 3; Ile 2; Trajet 3; Ile 2];
 [Ile 3; Trajet 1; Ile 5]]
```

Corrigé

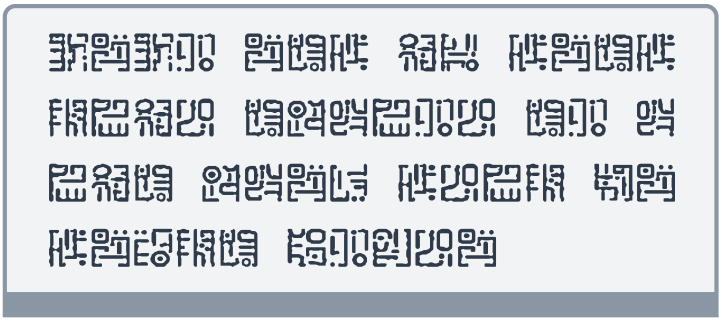
```
let rec ajoute x l =
  match l with
  | [] -> []
  | t::q -> (x::t) :: ajoute x q

let rec chemins r =
  match r with
  | Finie -> [ [] ]
  | Simple(v,r') -> ajoute v (chemins r')
  | Double((v1,r1),(v2,r2)) ->
```

```
    ajoute v1 (chemins r1)
    @ ajoute v2 (chemins r2)
```

63.1.4 Décoder les Codeglyphes

Tout au long de votre périple, votre exploratrice, Nicode, déchiffre des stèles contenant d'étranges caractères, les *Codeglyphes* :



Après avoir passé beaucoup de temps sur le déchiffrage, vous comprenez que tout repose sur votre interprétation de quatre Codeglyphes que l'on vous donne ici sous forme déchiffrée :

Premier Codeglyphe :

```
#include <stdio.h> int m
ain(void) {int t[]={1,1,
0,4,1,2,1,5,0,8,0,4,0,2,
0,4};for(int i=0;i<16;i=
i+2)printf("%c",'A'+10*t
[i]+t[i+1]);}
```

Deuxième Codeglyphe :

```
#include <stdio.h> int m
ain(void){int t[]={3,0,1
3,18,11,21,23,21,24,24,4
};int b=t[3]/2;int a=t[b
]-t[b+1];for (int i=0;i<
=b+1;i++){int b=t[i];if(
t[i]>a)b=t[t[i]-a];prin
tf("%c",'A'+b);}}
```

Troisième Codeglyphe :

```
let rec f x xx=if x=0 th
en [] else x mod xx::f(x
/xx)xx and g ll l1 l=mat
ch ll with []->0|x::xx->
x*l1+g xx(l*l1) l in pri
nt_int(g(f 73 4)1 3)
```

Quatrième Codeglyphe (une partie n'est pas déchiffrée) :

```
#include <stdio.h> void d(ch
ar s[], int k){for(int i=0;i
<10;i++)s[i]='A'+(((s[i]-'A'
)+k)%26);}int main(void){cha
r s[]="MLDLCXZTCP";d(s,
;printf(s);}
```

Question 23 On vous demande, pour ces quatre programmes :

- de les présenter sous une forme lisible et acceptée par un compilateur. Cela peut passer par le renommage des variables ou fonctions.
- de dérouler leur exécution, attention pour le quatrième programme, vous devez deviner le bout de code non déchiffré.
- d'expliquer ce que fait le programme dans des termes simples.

Indication : on rappelle que les caractères sont dans l'ordre alphabétique depuis `'A'`. Ainsi `'A'+3` est le caractère `'D'`.

Corrigé

1. Premier programme :

```c
#include <stdio.h>

int main(void) {
    int t[]={1,1,0,4,1,2,1,5,0,8,0,4,0,2,0,4};

    for(int i=0;i<16;i=i+2) {
        printf("%c",'A'+10*t[i]+t[i+1]);
    }
}
```

On comprend ainsi que ce programme va lire le numéro dans l'alphabet, en commençant à $A = 0$, écrit en décimal sur deux cases consécutives de t. Ainsi, les numéros de lettres sont 11, 4, 12, 15, 8, 4, 2, 4 ce qui correspond à LEMPIECE.

2. Deuxième programme :

```c
#include <stdio.h>

int main(void) {
    int t[]={3,0,13,18,11,21,23,21,24,24,4};
    int b=t[3]/2; // 18/2=9
    int a=t[b]-t[b+1]; // t[9]-t[10]=24-4=20

    for (int i=0;i<=b+1;i++){
        int c=t[i]; // on change le nom
        if(t[i]>a)
            c=t[t[i]-a];
        printf("%c",'A'+c);
    }
}
```

Ici, le plus simple semble de faire un tableau de valeurs :

i	$t[i]$	$t[i]-a$	$t[t[i]-a]$	c	'A' + c
0	3			3	D
1	0			0	A
2	13			13	N
3	18			18	S
4	11			11	L
5	21	1	0	0	A
6	23	3	18	18	S
7	21	1	0	0	A
8	24	4	11	11	L
9	24	4	11	11	L
10	4			4	E

et on lit donc DANSLASALLE.

3. Troisième programme :

```
let rec f x xx =
  if x=0
  then []
  else x mod xx
       :: f (x/xx) xx
and g ll l1 l =
  match ll with
  | []->0
  | x::xx-> x * l1 + g xx (l*l1) l
in print_int (g (f 73 4) 1 3)
```

qu'on peut rendre plus lisible en nommant les variables de manière raisonnable :

```
let rec decompose n base =
  if n=0
  then []
  else n mod base :: decompose (n/base) base
and compose ecriture puissance base =
  match ecriture with
  | []->0
  | chiffre::ecriture'-> chiffre * puissance
                         + compose ecriture' (base*puissance) base
in print_int (compose (decompose 73 4) 1 3)
```

On voit alors qu'il s'agit de décomposer 73 en base 4 et d'en déduire le nombre en base 3 en lisant cette écriture.

Ainsi 73 donne la liste `[1;2;0;1]` qui correspond en base 3 à $1 + 2 \times 3 + 27 = 34$.

4. Quatrième programme :

```
#include <stdio.h>

void d(char s[], int k) {
    for(int i=0;i<10;i++)
        s[i]='A'+(((s[i]-'A')+k)%26);
}

int main(void){
    char s[]="MLDLCXZTCP";
```

```
    d(s,???);
    printf(s);
}
```

On comprend que la fonction `d` est un décalage vers la droite de k unités. Il suffit alors de chercher tous les décalages jusqu'à obtenir une valeur signifiante.

k	s
0	MLDLCXZTCP
1	NMEMDYAUDQ
2	ONFNEZBVER
3	POGOFACWFS
4	QPHPGBDXGT
5	RQIQHCEYHU
6	SRJRIDFZIV
7	TSKSJEGAJW
8	UTLTKFHBKX
9	VUMULGICLY
10	WVNVMHJDMZ
11	XWOWNIKENA
12	YXPXOJLFOB
13	ZYQYPKMGPC
14	AZRZQLNHQD
15	BASARMOIRE
16	CBTBSNPJSF
17	DCUCTOQKTG
18	EDVDUPRLUH
19	FEWEVQSMVI
20	GFXFWRTNWJ
21	HGYGXSUOXK
22	IHZHYTVPYL
23	JIAIZUWQZM
24	KJBJAVXRAN
25	LKCKBWYSBO

On pouvait donc lire BASARMOIRE (pour $k = 15$).

5. Au final, la phrase cachée est LEMPIECEDANSLASALLE34BASARMOIRE.

Une fois les quatre programmes compris, vous avez pu découvrir l'emplacement du M-PIece ! Comme un vrai pirate, vous avez alors partagé le trésor avec tout l'équipage.

DEVOIR 64

DS 4 - C et mémoire

64.1 DS 4 — C et mémoire

MP2I, année scolaire 2022-2023. Durée totale 2h30.

Tout document interdit. Tout dispositif électronique interdit.

On vous demande d'écrire lisiblement et de bien présenter. Les programmes en particulier doivent être bien indentés et lisibles.

64.1.1 C et mémoire

On considèrera dans ce sujet que tous les en-têtes ont déjà été inclus en C.

Question 1 Expliquer brièvement l'organisation de la mémoire d'un programme compilé.

On tachera de faire un diagramme indiquant les différentes zones mémoire ainsi que leur rôle.

Corrigé

Voir cours.

Question 2 Si `i` est une valeur du type `int` et que `p` et `q` sont des pointeurs du type `int *`, indiquez les instructions d'affectation licites :

a) `p = i;` b) `*p = &i;` c) `&p = q;`

b) `p = &q;` e) `p = *&q;` f) `p = q;`

c) `p = *i;` h) `*p = q;` i) `*p = *q;`

Corrigé

Les instructions e, f et i sont licites.

Pour s'en convaincre, on pouvait faire un tableau en indiquant les types à gauche et à droite du symbole `=` :

	gauche		droite
a	`int *`	=	`int`
b	`int`	=	`int *`
c	`int **`	=	`int *`
d	`int *`	=	`int **`
e	`int *`	=	`int *`
f	`int *`	=	`int *`
g	`int *`	=	invalide
h	`int`	=	`int *`
i	`int`	=	`int`

Question 3 - Écrire une fonction de prototype

```
```c
void max_min(int a[], int n, int *min, int *max);
```

qui prend en entrée un tableau `t` de `n` entiers,
supposé non vide,
et place la plus petite valeur dans `min` passé par pointeur, et
sa plus grande valeur dans `max`.
```

- Écrire une fonction `main` typique qui testerait cette fonction sur le tableau `{1, 5, -1, 3}` en affichant la réponse sous la forme

```
min=-1 max=5
```

Corrigé

```c
void max_min(int a[], int n, int *min, int *max)
{
    *min = a[0];
    *max = a[0];
    for (int i = 1; i < n; i++)
    {
        if (*min > a[i])
            *min = a[i];
        if (*max < a[i])
            *max = a[i];
    }
}
```

```c
int main(void) {
    int t[] = { 1, 5, -1, 3 };
    int max, min;
    max_min(t, 4, &min, &max);
    printf("min=%d max=%d\n", min, max);
}
```

Question 4 Écrire une fonction de prototype

```c
void find_two_largest(int a[], int n,
    int *largest, int *second_largest);
```

qui prend en entrée un tableau `t` de `n` entiers, en supposant $n > 1$, et place la plus grande valeur présente dans `largest` et la seconde plus grande dans `second_largest`.

Exemple : pour `{ 1, 4, 5, 2, 4 }` on va donc avoir 5 dans `largest` et 4 dans `second_largest`.

Corrigé

Un classique des fausses questions faciles. Il y a deux pièges :

- on ne peut pas initialiser aux deux premières valeurs parce qu'elles peuvent être égales. Il faut donc chercher les deux premières valeurs **différentes**.
- il faut bien faire attention à ne pas faire deux `if` successifs dans la boucle car le premier `if` perturbe le second.

```c
void find_two_largest(int a[], int n,
    int *largest, int *second_largest)
{
    *largest = a[0];
    int i = 1;
    while (a[i] == *largest)
        i++;
    if (a[i] > *largest) {
        *second_largest = *largest;
        *largest = a[i];
    } else {
        *second_largest = a[i];
    }
    for (int j = i + 1; j < n; j++) {
        if (a[j] > *largest) {
            *second_largest = *largest;
            *largest = a[j];
        } else if (a[j] < *largest && a[j] > *second_largest)
            *second_largest = a[j];
    }
}
```

On suppose ici que le tableau contient au moins deux valeurs distinctes, sinon la boucle `while` sort du tableau.

Question 5 On considère les structures suivantes :

```c
struct point {
    int x;
    int y;
};
struct rectangle {
    struct point upper_left;
    struct point lower_right;
};
```

Écrire une fonction de prototype

```c
rectangle *mon_rectangle();
```

qui **alloue** et renvoie un `rectangle` de coin supérieur gauche de coordonnées $(10, 25)$ et de coin inférieur bas $(20, 15)$.

Corrigé

Faute de `typedef`, il faut écrire `struct rectangle` (ou ajouter `typedef struct rectangle rectangle;`).

```c
struct rectangle *mon_rectangle() {
    struct rectangle *r = malloc(sizeof(struct rectangle));
    r->upper_left.x = 10;
    r->upper_left.y = 25;
    r->lower_right.x = 20;
    r->lower_right.y = 15;
```

```
    return r;
}
```

Question 6 Écrire une fonction de prototype

```
int *create_array(int n, int initial_value);
```

qui alloue un tableau de n entiers et initialise chaque case avec la valeur `initial_value`.

Corrigé

```
int *create_array(int n, int initial_value)
{
    int *t = malloc(n * sizeof(int));
    for (int i = 0; i < n; i++)
        t[i] = initial_value;
    return t;
}
```

64.1.2 OCaml et complexité

C'est le grand jour où vous fêtez avec vos amis et votre famille votre réussite aux concours. Vous organisez un grand repas et vous souhaitez, par commodités, placer le maximum de convives sur une même table.

Problème : certains de vos convives ne s'entendent pas (sûrement une dispute sur leur langage de programmation préféré) et ne souhaitent pas être placés à la même table. Comment allouer des tables aux convives en essayant d'en avoir le moins possibles ?

On considère que chacun de vos n convives est identifié de manière unique par un numéro dans $[\![0, n-1]\!]$ et que l'on dispose d'une matrice :

```
val ami : bool array array
```

de dimension (n, n) telle que `ami.(i).(j)` soit vrai lorsque `i` et `j` sont amis. **Cette relation étant considérée symétrique** on a toujours `ami.(i).(j) = ami.(j).(i)`.

Un tableau `table` de longueur n qui associe à chaque convive un numéro dans $[\![0, p-1]\!]$ est appelé une affectation des convives à p tables. Ainsi `table.(i)` vaut `k` quand on place le convive `i` à la table `k`.

Question 7 Écrire une fonction de prototype

```
val valide : int array -> bool
```

qui s'assure que l'on n'a jamais placé à une même table deux personnes qui ne sont pas amies.

Corrigé

```
let valide table =
  let n = Array.length table in
  let is_valide = ref true in
  for i = 0 to n-1 do
    for j = i+1 to n-1 do
      (* pas amis et à la même table *)
      if not ami.(i).(j) && table.(i) = table.(j)
      then is_valide := false
    done
  done;
  !is_valide
```

Question 8 Quelle est la complexité de la fonction précédente ?

Corrigé

On parcourt chaque case de la moitié de la matrice, donc $O(n^2)$ cases et pour chaque case on effectue un traitement en $O(1)$, donc la complexité est en $O(n^2)$.

Note : si les personnes sont presque toutes amies, il serait plus raisonnable de travailler sur la liste des inimitiés.

On va essayer de construire une affectation valide de manière incrémentale ainsi :

- au départ on initialise le tableau `table` avec des valeurs quelconques (prenez `-1` par exemple)
- Pour i allant de 0 à $n-1$
  - on suppose qu'il y a un numéro valide dans `table` pour tous les convives de numéro $< i$ et qu'il n'existe pas de couple (j,k) où $j,k < i$ qui ne soient pas amis et tels que `table.(j) = table.(k)`.
  - on calcule la liste des numéros de tables de tous les convives de numéro $< i$ qui ne sont pas amis avec i.
  - on en déduit le plus petit numéro ≥ 0 de table qui ne soit pas dans cette liste
  - on affecte ce numéro à i en le plaçant dans `table.(i)`.

Question 9 On suppose qu'il y a 6 convives, de 0 à 5, et que les inimitiés sont les suivantes :

- 0 et 2 sont ennemis
- 2 et 3 sont ennemis
- 2 et 4 sont ennemis
- 3 et 4 sont ennemis
- 1 et 5 sont ennemis

1. Dérouler l'algorithme précédent avec précision et en déduire une affectation en trois tables.
2. Peut-on faire mieux ?

Corrigé

1. On va construire `i -> table.(i)` de manière incrémentale :
   - $0 \mapsto 0$
   - $1 \mapsto 0$
   - $2 \mapsto 1$ car il a 0 comme ennemi < 2 et qu'il est déjà à la table 0.
   - $3 \mapsto 0$ car il a 2 comme ennemi < 3.
   - $4 \mapsto 2$ car il a 2 et 3 comme ennemis < 4.
   - $5 \mapsto 1$ car il a 1 comme ennemi < 5.
2. On ne peut pas faire mieux que trois tables, car comme 2, 3 et 4 ne s'apprécient pas deux à deux, il faut les placer sur trois tables différentes. La solution précédente est donc optimale.

Question 10 Écrire une fonction de prototype

```
val tables_ennemis : int array -> int -> int list
```

telle que l'appel à `tables_ennemis table i` renvoie la liste des numéros de table (avec possiblement des répétitions) des ennemis du convive i dont le numéro est $< i$.

Corrigé

```
let tables_ennemis table i =
  let rec cherche j =
    if j < 0
    then []
    else let l = cherche (j-1) in
      if not ami.(i).(j)
      then table.(j) :: l
```

```
    else l
in cherche (i-1)
```

Question 11 Quelle est la complexité de cette fonction ?

Corrigé

C'est la boucle induite par la fonction récursive qui compte car on n'y effectue que des opérations en temps constant, donc la complexité est en $O(i)$.

Question 12 Écrire une fonction de prototype

```
val plus_petit_absent : int list -> int
```

qui prend une liste d'entiers et renvoie le plus petit entier naturel qui n'est pas présent dans la liste.

Votre réponse devra être linéaire pour avoir tous les points, mais un algorithme non linéaire aura au moins la moitié des points.

Corrigé

Si la liste est de longueur n, le plus petit absent est dans $[\![0, n]\!]$: on marque dans un tableau de $n+1$ booléens les éléments de la liste qui sont dans cet intervalle, puis on cherche la première case non marquée.

```
let plus_petit_absent l =
  let n = List.length l in
  (* present.(k) indique si k est dans l, pour k <= n *)
  let present = Array.make (n+1) false in
  List.iter (fun x -> if x >= 0 && x <= n then present.(x) <- true) l;
  let k = ref 0 in
  while present.(!k) do
    incr k
  done;
  !k
```

Chaque étape est en $O(n)$, la fonction est donc linéaire.

Question 13 En déduire une fonction de prototype

```
val affecte : unit -> int array
```

qui réalise l'algorithme d'affectation présenté ci-dessus.

Corrigé

```
let affecte () =
  let n = Array.length ami in
  let table = Array.make n 0 in
  for i = 1 to n-1 do
    let l = tables_ennemis table i in
    table.(i) <- plus_petit_absent l
  done;
  table
```

Question 14 Quelle est sa complexité en supposant que `plus_petit_absent` est linéaire ?

Corrigé

Les deux appels dans la boucle sont en $O(i)$, donc, comme $i \leq n$, la complexité totale est en $O(n^2)$.

Question 15 Établir un petit contre-exemple démontrant que cette fonction ne réalise pas le plus petit nombre de tables.

Corrigé

- Premier contre-exemple :
  - 0 est ennemi avec 1
  - 1 est ennemi avec 0
  - 2 est ennemi avec 3
  - 3 est ennemi avec 1 et 2

  On va dérouler l'algorithme :

  - $0 \mapsto 0$
  - $1 \mapsto 1$ car évite $\{0\}$
  - $2 \mapsto 0$ car évite $\emptyset$
  - $3 \mapsto 2$ car évite $\{0, 1\}$

  On obtient donc $(0, 1, 0, 2)$ en 3 tables alors que $(0, 1, 1, 0)$ en deux tables est optimal.
- Un contre-exemple classique :
  - 0 est ennemi avec 1, 2 et 3
  - 1 est ennemi avec 0, 2 et 4
  - 2 est ennemi avec 0, 1 et 5
  - 3 est ennemi avec 0, 4 et 5
  - 4 est ennemi avec 1, 3 et 5
  - 5 est ennemi avec 2, 3 et 4

  On va dérouler l'algorithme :

  - $0 \mapsto 0$
  - $1 \mapsto 1$ car évite $\{0\}$
  - $2 \mapsto 2$ car évite $\{0, 1\}$
  - $3 \mapsto 1$ car évite $\{0\}$
  - $4 \mapsto 0$ car évite $\{1\}$
  - $5 \mapsto 3$ car évite $\{0, 1, 2\}$

  Or l'affectation $(0, 1, 2, 1, 2, 0)$ est valide et utilise une table de moins.

  Le contre-exemple qu'on vient de voir en reste un en renumérotant les convives.

DEVOIR 65

DS 5 - Listes chaînées

65.1 DS 5 — Listes chaînées

MP2I, année scolaire 2022-2023, devoir surveillé du 11 mars 2023. Durée totale 4h.

Tout document interdit. Tout dispositif électronique interdit.

On vous demande d'écrire lisiblement et de bien présenter. Les programmes en particulier doivent être bien indentés et lisibles. Utilisez de préférence plusieurs couleurs. **Commentez** votre code dès que ce qu'il fait n'est pas évident.

L'introduction et l'utilisation de fonctions auxiliaires est autorisée, et même encouragée, d'autant plus si cela améliore la lisibilité et la compréhension. Les fonctions auxiliaires **doivent être précédées de leur type (en OCaml) et commentées**.

Lorsque l'on pose des hypothèses sur les arguments, il n'est pas nécessaire dans vos fonctions de le vérifier ni de prévoir le comportement si l'hypothèse est violée.

De nombreuses questions comportent une sous-question sur la complexité. On prendra garde à ne pas lire trop vite l'énoncé et à ne pas ainsi oublier de répondre à une partie de la question.

Le sujet comporte un premier exercice de reprise du cours et un problème découpé en plusieurs parties relativement indépendantes.

Attention : on rédigera chaque partie sur des copies séparées pour bien différencier les numéros des questions.

65.1.1 I. A link to a past TP

On va reprendre le type des listes chaînées en `C` :

```c
struct maillon {
    int valeur;
    struct maillon *suivant;
};

typedef struct maillon maillon;

struct liste {
    maillon *debut;
    maillon *fin;
};

typedef struct liste liste;
```

- Dans un maillon, soit `suivant` pointe vers le maillon suivant de la chaîne ou vaut `NULL`.
- La liste vide aura la valeur `NULL` pour `debut` et `fin`.
- Dans une liste non vide, `debut` pointe sur le premier maillon et `fin` sur le dernier.

On suppose définies des fonctions

```
// cree un maillon
maillon *creer_maillon(int valeur, maillon *suivant);
// supprime un maillon
void suppr_maillon(maillon *);
// cree une liste vide
liste *creer_liste(void);
// supprime une liste
void suppr_liste(liste *);
```

Question 1 Écrire une fonction de prototype

```
int longueur(liste *);
```

qui renvoie la longueur d'une liste.

Corrigé

```
int longueur(liste *l)
{
    int n = 0;
    maillon *m = l->debut;
    while (m != NULL)
    {
        n++;
        m = m->suivant;
    }
    return n;
}
```

Question 2 Écrire une fonction de prototype

```
int somme(liste *);
```

qui renvoie la somme des valeurs des éléments d'une liste.

Corrigé

```
int somme(liste *l)
{
    int s = 0;
    maillon *m = l->debut;
    while (m != NULL)
    {
        s = s + m->valeur;
        m = m->suivant;
    }
    return s;
}
```

Question 3 Écrire une fonction de prototype

```
void liste_ajout_en_tete(liste *l, int valeur);
```

qui ajoute un maillon en tête de la liste passée en paramètre en $O(1)$.

On fera attention au cas où la liste est vide.

Corrigé

```c
void liste_ajout_en_tete(liste *l, int valeur)
{
    l->debut = creer_maillon(valeur, l->debut);
    if (l->fin == NULL)
        l->fin = l->debut;
}
```

Question 4 Écrire une fonction de prototype

```c
maillon *liste_cherche_maillon(liste *l, int valeur);
```

qui renvoie un pointeur sur le premier maillon de valeur `valeur` ou renvoie `NULL` s'il n'y en a pas.

Corrigé

```c
maillon *liste_cherche_maillon(liste *l, int valeur)
{
    maillon *m = l->debut;
    while (m != NULL && m->valeur != valeur)
    {
        m = m->suivant;
    }
    return m;
}
```

Question 5 Expliquer comment cette structure permet d'implémenter une pile et donner les différentes complexités des opérations de la structure liste.

Corrigé

L'ajout et la suppression en tête sont en $O(1)$, on s'en sert donc respectivement pour les opérations push et pop.

DEVOIR 66

DS 6 - Arbres n-aires en C

66.1 DS 6 — Arbres n-aires en C

MP2I, année scolaire 2022-2023. Durée totale 4h.

La partie 4 (« Médians ») reprend le premier problème de l'épreuve d'informatique de Polytechnique 2004 (filière MP, option informatique).

Tout document interdit. Tout dispositif électronique interdit.

On vous demande d'écrire lisiblement et de bien présenter. Les programmes en particulier doivent être bien indentés et lisibles.

66.1.1 1. Arbres n-aires en C

On va représenter ici des arbres n-aires en C.

On considère le type suivant :

```c
struct noeud {
    int valeur;
    int arite;
    struct noeud *enfants;
};

typedef struct noeud arbre;
```

Ainsi, chaque nœud contiendra : sa valeur, c'est-à-dire l'étiquette du nœud, son arité, c'est-à-dire son nombre de nœuds enfants et le tableau de ses enfants.

Les nœuds d'arité nulle sont les feuilles.

Question 1 Quelle différence avec le type vu en OCaml pour la même structure ? Expliquer.

Corrigé

On ne peut pas connaître la taille d'un tableau passé par pointeur comme dans cette structure. Il est donc nécessaire d'avoir également l'arité dans les nœuds.

On remarque qu'on aurait aussi pu faire une liste chaînée de nœuds enfants, mais cela rajoute beaucoup de complexité dans la gestion de la mémoire.

Question 2 Expliquer comment on peut représenter l'arbre vide en considérant des éléments du type `arbre *`.

Corrigé

Il suffit de le représenter avec le pointeur `NULL`. Une subtilité ici, l'arbre vide n'est jamais un sous-arbre. On est plus proche d'un type optionnel comme en OCaml.

Question 3 Écrire une fonction de prototype

```
int taille(arbre *a);
```

qui renvoie le nombre de nœuds d'un arbre.

Corrigé

```
int taille(arbre *a) {
    if (a == NULL)
        return 0;

    int t = 1;
    for (int i = 0; i < a->arite; i++) {
        t += taille(&a->enfants[i]);
    }
    return t;
}
```

Question 4 Écrire une fonction qui affiche tous les nœuds dans un parcours préfixe.

```
void affiche(arbre *a);
```

Corrigé

```
void affiche(arbre *a) {
    if (a == NULL)
        return;

    printf("%d\n", a->valeur);
    for (int i = 0; i < a->arite; i++) {
        affiche(&a->enfants[i]);
    }
}
```

Question 5 Écrire une fonction de prototype

```
int somme(arbre *a);
```

qui renvoie la somme des étiquettes des nœuds dans l'arbre.

Corrigé

```
int somme(arbre *a) {
    if (a == NULL)
        return 0;

    int s = a->valeur;
    for (int i = 0; i < a->arite; i++) {
        s += somme(&a->enfants[i]);
    }
    return s;
}
```

Question 6 On souhaite libérer la mémoire occupée par un arbre et Justin Topain, votre camarade, propose le code suivant :

```
void libere(arbre *a) {
    free(a);
    for (int i = 0; i < a->arite; i++) {
        free(a->enfants[i]);
```

```
    }
}
```

Indiquer les erreurs logiques dans ce code en expliquant pourquoi elles en sont. Proposer une version correcte de cette fonction.

Corrigé

Un problème ici est de libérer `a` avant d'accéder à `a->enfants` ou `a->arite`. De plus, il faudrait gérer l'arbre vide.

On a également un problème avec le passage par pointeur, `a->enfants[i]` est un `arbre` et pas un `arbre *`. Notons que c'est le même problème dans les questions précédentes.

Autre problème, cette fonction doit être récursive pour ne pas libérer qu'un nœud et ses enfants.

Enfin, les enfants d'un nœud sont rangés dans **un seul** tableau alloué `a->enfants` : c'est ce tableau qu'il faut libérer (après avoir libéré les tableaux des descendants), et non chacune de ses cases. On suppose ici que la racine a été allouée par `malloc` et que chaque tableau `enfants` est alloué par `malloc` (ou vaut `NULL` pour une feuille).

```c
// libère les tableaux d'enfants de tous les descendants de a, sans libérer a
void libere_enfants(arbre *a) {
    for (int i = 0; i < a->arite; i++) {
        libere_enfants(&a->enfants[i]);
    }
    free(a->enfants);
}

void libere(arbre *a) {
    if (a == NULL)
        return;
    libere_enfants(a);
    free(a);
}
```

66.1.2 2. Recherche trichotomique

On considère ici un tableau de valeurs entières triées dans l'ordre croissant. Pour rechercher si un élément x est dans un tel tableau de taille $n \geq 2$, on propose de prendre les deux éléments d'indice $k = \left\lfloor \frac{n}{3} \right\rfloor$ et $p = \left\lfloor \frac{2n}{3} \right\rfloor$:

- Si $x = t[k]$ on renvoie k
- Si $x = t[p]$ on renvoie p
- Si $x < t[k] \leq t[p]$, on continue la recherche dans le tableau des indices 0 à $k - 1$
- Si $t[k] < x < t[p]$, on continue la recherche dans le tableau des indices $k + 1$ à $p - 1$.
- Si $x > t[p]$, on continue la recherche dans le tableau des indices $p + 1$ à $n - 1$.

Quand le tableau a 0 ou 1 élément, on se contente de répondre faux ou de vérifier que le seul élément est x.

Question 7 La manière dont on a rédigé cet algorithme considère qu'on va extraire des sous-tableaux pour faire des appels récursifs. Justifier brièvement pourquoi ce n'est pas une bonne idée ici.

Corrigé

Ici, on a un algorithme où la fusion est clairement en $O(1)$. Or, les extractions de sous-tableaux induisent une fusion linéaire, ce qui est forcément plus coûteux que l'algorithme lui-même.

Question 8 Décrire, sans écrire de code mais avec précision, un algorithme récursif de recherche n'effectuant pas d'extraction de sous-tableaux.

Corrigé

Petite imprécision du sujet qui parle de renvoyer un indice dans les premiers cas puis change d'avis en demandant juste le booléen.

On considère une fonction récursive d'interface $Rech(t, i, j, x)$ qui va chercher x dans t aux indices entre i et j (inclus).

- Si $j < i$ on renvoie faux.
- Si $j = i$, on teste si $t[i] = x$.
- Sinon, on note $m = j - i + 1$ la taille de la zone de recherche et on pose $k = i + \left\lfloor \frac{m}{3} \right\rfloor$ et $p = i + \left\lfloor \frac{2m}{3} \right\rfloor$.
  - Si $x = t[k]$ ou $x = t[p]$ on renvoie vrai.
  - Si $x < t[k] \leq t[p]$, on continue avec $Rech(t, i, k - 1, x)$
  - Si $t[k] < x < t[p]$, on continue avec $Rech(t, k + 1, p - 1, x)$
  - Si $x > t[p]$, on continue avec $Rech(t, p + 1, j, x)$

On note $T(n)$ le nombre d'opérations élémentaires effectuées pour un tableau de n valeurs.

Question 9 Montrer qu'on a, pour $n \geq 2$,

$$T(n) \leq T\left(\left\lceil \frac{n}{3} \right\rceil\right) + O(1)$$

et $T(0) = T(1) = 1$.

Corrigé

Pour les cas de base, le 1 est un peu grossier, on remarque juste qu'on effectue un $O(1)$ opérations élémentaires.

Si $n \geq 2$, on va couper en trois. Les fragments sont de taille $\leq \left\lceil \frac{n}{3} \right\rceil$, on continue dans un seul, et le coût du découpage est en $O(1)$, on a donc la majoration de $T(n)$ demandée.

Question 10 On pose $u_n = T(3^n)$.

Exprimer u_n sans récursivité.

Corrigé

Imprécision encore, c'est une majoration qu'on exprime ici.

$u_0 = T(1) = 1$ et $u_{n+1} = T(3^{n+1}) \leq u_n + O(1)$.

Ainsi $u_n \leq nO(1) = O(n)$.

Question 11 En déduire la complexité de cette recherche sous la forme d'un $O(\cdots)$.

Corrigé

Soit p tel que $3^{p-1} < n \leq 3^p$, donc $p = \lceil \log_3 n \rceil$.

On a $T(n) \leq T(3^p) \leq O(p) = O(\log_3 n)$.

Question 12 Cette recherche est-elle meilleure que la recherche dichotomique ?

Corrigé

On a $O(\log_3 n) = O(\log_2 n)$ donc on obtient exactement la même complexité asymptotique.

66.1.3 3. Table de hachage avec chaînage

On va réaliser dans cette partie une table de hachage avec chaînage pour gérer les collisions en OCaml.

On va utiliser ici la fonction `Hashtbl.hash` qui a le prototype suivant :

```
val hash : 'a -> int
```

et qui à toute valeur `x` va associer un entier strictement positif `hash x`. On a la garantie que si `x = y` alors `hash x = hash y`.

Ensemble

On considère le problème plus simple vu en classe consistant à représenter un ensemble fini de valeurs.

Question 13 Quelle relation entre ce problème et celui de la création d'un dictionnaire ?

Corrigé

Il suffit de rajouter des couples `(clé,valeur)` et de ne considérer que la `clé` pour le hachage.

On va réaliser ici une table de hachage où les cases contiennent des listes de valeurs pour gérer les collisions.

On considère donc le type suivant :

```
type 'a table = 'a list array
```

Question 14 Écrire une fonction

```
val contient : 'a table -> 'a -> bool
```

qui indique si un élément est dans la table.

Corrigé

L'entier `hash x` est quelconque : on le réduit modulo la taille de la table pour obtenir un indice valide.

```
let contient t x = List.mem x t.(hash x mod Array.length t)
```

Question 15 Écrire une fonction

```
val ajoute : 'a table -> 'a -> unit
```

qui rajoute un élément dans la table.

Corrigé

```
let ajoute t x =
  let h = hash x mod Array.length t in
  t.(h) <- x :: t.(h)
```

Dictionnaire

On considère ici le type précédent enrichi pour des dictionnaires associant des `'a` à des `'b`.

```
type ('a, 'b) dico = ('a * 'b) list array
```

On continue encore avec le même principe de chaînage en cas de collision mais on ne le fait que sur la première composante du couple.

Question 16 Écrire une fonction

```
val assoc : ('a * 'b) list -> 'a -> 'b
```

qui prend une liste de couples `l` et une valeur `x` et renvoie le premier `y` tel que `(x,y)` soit dans `l`.

Si l'élément n'est pas dedans, on fera une erreur.

Corrigé

```
let rec assoc l x =
  match l with
  | [] -> failwith "Non présent"
  | (y,v)::q ->
    if x = y
    then v
    else assoc q x
```

Question 17 En déduire une fonction

```
val dico_recherche : ('a, 'b) dico -> 'a -> 'b
```

qui prend un dico `d` et une valeur `x` et renvoie un `y` associé.

Corrigé

```
let dico_recherche d x = assoc d.(hash x mod Array.length d) x
```

Question 18 Quels sont les défauts de cette implémentation comparativement à l'adressage ouvert vu en TP ?

Corrigé

Le problème est la fragmentation mémoire induite par les listes chaînées. Sans mécanisme particulier, chaque collision risque d'engendrer des lenteurs d'accès mémoire qui sont évitées par un tableau qui peut tenir complètement dans le cache du processeur.

66.1.4 4. Médians

Un *médian* d'un ensemble $X = \{e_1, e_2, ..., e_n\}$ de n nombres entiers distincts est un nombre e dans X tel que les nombres d'éléments strictement plus petits et strictement plus grands que e dans X diffèrent d'au plus 1 ($n > 0$). Si n est impair, le médian est unique ; si n est pair, il y a deux médians possibles.

Le problème de la *sélection* consiste à trouver l'élément de rang k dans X, c'est-à-dire l'élément se trouvant en k-ième position quand X est trié en ordre croissant ($1 \leq k \leq n$).

Nous supposerons l'ensemble X représenté par la liste de ses éléments, c'est-à-dire par le type ensemble défini par :

```
type ensemble = int list
```

On supposera que les `ensemble` sont sans répétitions mais pas nécessairement triés.

Question 19 Écrire une fonction `card` qui prend un ensemble et renvoie son nombre d'éléments.

```
val card : ensemble -> int
```

Quelle est la complexité de cette fonction par rapport à n ?

Corrigé

Vu la question, il fallait réécrire la fonction `List.length`.

```
let rec card e =
  match e with
  | [] -> 0
  | _::q -> 1 + card q
```

Elle est linéaire en n : un appel par élément.

Soit p un entier quelconque, on sera amené à partitionner l'ensemble X en éléments plus grands, égaux ou plus petits que p.

Question 20 Écrire la fonction `nPetits` qui prend en arguments un entier p et l'ensemble X ; et qui renvoie le nombre d'éléments strictement inférieurs à p dans X.

```
val nPetits : int -> ensemble -> int
```

Quelle est la complexité de cette fonction par rapport à n ?

Corrigé

```
let rec nPetits p x =
  match x with
  | [] -> 0
  | t::q -> (if t < p then 1 else 0) + nPetits p q
```

Cette fonction est linéaire en n car elle effectue un parcours de la liste.

Pour sélectionner le k-ième élément de X, on prend un nombre p bien choisi dans X, appelé *pivot* ($p \in X$), et on effectue une partition de X par rapport à p.

Question 21 Écrire la fonction `partitionP` qui prend en arguments un entier p et l'ensemble X ; et qui renvoie l'ensemble de tous les éléments de X strictement plus petits que p.

```
val partitionP : int -> ensemble -> ensemble
```

Modifier cette fonction pour obtenir la fonction `partitionG` qui retourne l'ensemble de tous les éléments de X strictement plus grands que p. Quelles sont les complexités en temps de ces fonctions par rapport à n ?

Corrigé

```
let rec partitionP p x =
  match x with
  | [] -> []
  | t::q -> let q' = partitionP p q in
    if t < p then t::q' else q'

let rec partitionG p x =
  match x with
  | [] -> []
  | t::q -> let q' = partitionG p q in
    if t > p then t::q' else q'
```

Exactement comme la question précédente : linéaire en n.

Question 22 Écrire la fonction récursive `elementDeRang` qui prend en arguments un nombre k entier naturel et l'ensemble X ; et qui renvoie l'élément de rang k dans X, en choisissant comme pivot le premier élément de la liste représentant X.

```
val elementDeRang : int -> ensemble -> int
```

Corrigé

```
let rec elementDeRang k x =
  match x with
  | [] -> failwith "Vide"
  | p::q ->
    let petits, grands =
      partitionP p q, partitionG p q in
```

```
    let m = card petits in
    if k <= m
    then elementDeRang k petits
    else if k = m+1
    then p
    else elementDeRang (k-m-1) grands
```

Le nombre d'opérations pris par la fonction précédente varie en fonction du choix du pivot.

Question 23 Donner un ordre de grandeur, par rapport à n, du nombre maximum $M(n)$ d'opérations pris par la fonction précédente.

Corrigé

En pire cas, on effectue un appel récursif avec un seul élément de moins, comme les opérations sont linéaires dans la longueur des listes, on retrouve une complexité quadratique.

Plus précisément : $M(n) \leq M(n-1) + O(n)$ donc $M(n) = O(n^2)$.

En supposant équiprobables tous les rangs possibles pour le pivot choisi dans X, on peut démontrer que le temps moyen pris par `elementDeRang` est borné supérieurement par $T(n)$ où T est une fonction croissante, vérifiant $T(0) = 0$ et la formule de récurrence suivante :

$$T(n) \leq \ell n + \frac{1}{n}\sum_{i=1}^{n} \max\{T(i-1), T(n-i)\}$$

(ℓ est une constante indépendante de n et k).

Question 24 En déduire une constante c, qu'on déterminera en fonction de ℓ, telle que pour tout n entier naturel, on a $T(n) \leq cn$.

Corrigé

Un peu pénible vu la manière dont l'énoncé le présente. On peut couper en deux la somme pour résoudre le maximum par croissance de T :

Si $n = 2p + \epsilon$ avec $\epsilon \in \{0, 1\}$, on a

$$\sum_{i=1}^{n} \max(T(i-1), T(n-i)) = \sum_{i=1}^{p} T(n-i) + \epsilon T(p) + \sum_{i=p+1+\epsilon}^{n} T(i-1) = \epsilon T(p) + 2\sum_{i=p+1+\epsilon}^{n} T(i-1)$$

On montre par récurrence forte sur n que $T(n) \leq cn$ avec $c = 4\ell$. C'est vrai pour $n = 0$. Soit $n \geq 1$, on suppose que $T(m) \leq cm$ pour tout $m < n$.

Dans le cas où $n = 2p$:

$$\begin{aligned}
T(n) &\leq \ell 2p + \frac{2}{2p} c \sum_{i=p+1}^{2p} (i-1) \\
&= \ell 2p + \frac{c}{p}\left(\frac{2p(2p-1)}{2} - \frac{(p-1)p}{2}\right) \\
&= \ell 2p + \frac{c}{2}(2(2p-1) - (p-1)) \\
&= \ell 2p + \frac{c}{2}(3p-1) \\
&\leq n(\ell + \frac{3c}{4})
\end{aligned}$$

Dans le cas où $n = 2p + 1$:

$$\begin{aligned} T(n) &\leq \ell(2p+1) + c\frac{p}{2p+1} + \frac{2c}{2p+1}\sum_{i=p+2}^{2p+1}(i-1) \\ &= \ell(2p+1) + \frac{c}{2p+1}(p + 2p(2p+1) - p(p+1)) \\ &= \ell(2p+1) + \frac{c}{2p+1}p(3p+2) \\ &\leq n(\ell + \frac{3c}{4}) \end{aligned}$$

la dernière inégalité venant de $4p(3p+2) \leq 3(2p+1)^2$.

Et on a dans les deux cas, $T(n) \leq n(\ell + \frac{3c}{4}) = cn$ pour $c = 4\ell$.

On s'intéresse à présent à optimiser le coût dans le pire cas $M(n)$. Considérons d'abord les sous-ensembles

$$X_i = \{e_{5i+1}, e_{5i+2}, e_{5i+3}, e_{5i+4}, e_{5i+5}\}$$

de 5 éléments consécutifs dans X pour $0 \leq 5i + 5 \leq n$.

Soit Y l'ensemble des médians des X_i.

Question 25 Écrire une fonction `medians` qui prend en arguments l'ensemble X ; et qui renvoie l'ensemble Y.

```
val medians : ensemble -> ensemble
```

Quelle est la complexité en temps pris par cette fonction par rapport à n ?

Corrigé

Les éléments d'un dernier paquet incomplet (moins de 5 éléments) ne sont pas pris en compte, puisque seuls les X_i avec $5i + 5 \leq n$ sont considérés.

```
let rec medians x =
  match x with
  | x1::x2::x3::x4::x5::q ->
    elementDeRang 3 [x1;x2;x3;x4;x5] :: medians q
  | _ -> []
```

Comme chaque calcul d'une médiane de 5 peut être supposé en temps constant, on a une complexité linéaire en n.

Pour améliorer la vitesse de la fonction de sélection, on prend à présent comme pivot le médian de l'ensemble Y.

Question 26 Écrire la fonction `elementDeRangBis` qui prend en entrée un entier naturel k et l'ensemble X ; et qui renvoie l'élément de rang k dans X, en prenant comme pivot le médian de l'ensemble Y.

```
val elementDeRangBis : int -> ensemble -> int
```

Question 27 Montrer que son temps maximum $M'(n)$ vérifie la formule

$$M'(n) \leq \ell' n + M'\left(\left\lfloor \frac{n}{5} \right\rfloor\right) + M'\left(7\left\lfloor \frac{n}{10} \right\rfloor + 4\right)$$

où $\lfloor x \rfloor$ est la partie entière de x, et ℓ' est une constante que l'on ne cherchera pas à expliciter. En déduire que pour n suffisamment grand, on a $M'(n) \leq c'n$ où c' est une constante à déterminer en fonction de ℓ'.

Question 28 Expliquer pourquoi l'ordre de grandeur du nombre maximal d'opérations pris par la fonction de sélection serait différent si on avait regroupé les éléments de X par groupes de 3, plutôt que par groupes de 5.

DEVOIR 67

DS 7 - Sac à dos fractionnaire

67.1 DS 7 — Sac à dos fractionnaire

MP2I, année scolaire 2022-2023. Durée totale 4h.

Tout document interdit. Tout dispositif électronique interdit.

On vous demande d'écrire lisiblement et de bien présenter. Les programmes en particulier doivent être bien indentés et lisibles.

Chaque problème est à traiter sur une copie séparée pour ne pas avoir de confusion entre les numérotations.

67.1.1 Sac à dos fractionnaire

On considère le problème suivant :

Problème 67.1 — : SacADosFractionnaire

Entrées :

- n objets numérotés de 1 à n.
- à chaque objet on associe deux entiers :
  - w_i son poids
  - v_i sa valeur
- Un poids maximal $W \in \mathbb{N}$

Pour $\sigma : [\![1, n]\!] \to \mathbb{Q}_+ \cap [0; 1]$, on note $w(\sigma) = \sum_{x=1}^{n} w_x \sigma(x)$, le poids de σ, et $v(\sigma) = \sum_{x=1}^{n} v_x \sigma(x)$ sa valeur.

On cherche

$$\max\{v(\sigma) \mid \sigma : [\![1, n]\!] \to \mathbb{Q}_+, w(\sigma) \leq W\}$$

C'est-à-dire que l'on cherche la répartition d'objets maximisant la valeur en restant dans la limite de poids.

Chacun des $\sigma(x)$ représente la *proportion* de l'objet x que l'on rajoute dans le sac. Pour que cette proportion soit représentable informatiquement, on se restreint à des rationnels, mais dans la suite, on considérera des flottants sans se poser de questions.

Pour résoudre ce problème, on se propose d'effectuer un algorithme glouton : on identifie l'objet qui représente le choix glouton et on place le maximum de cet objet dans le sac, c'est-à-dire soit l'objet en entier s'il rentre, soit la proportion suffisante pour saturer le sac.

Question 1 Rappeler brièvement le principe d'un algorithme glouton et du choix glouton.

Implémentation et tri

On va implémenter ce problème en C en représentant un problème par un élément du type :

```c
struct sacados {
    int nobjets;
    int *poids;
    int *valeurs;
    int poids_maximal;
};
typedef struct sacados sacados;
```

On suppose ainsi que toutes les valeurs sont entières. Les tableaux représentant les poids et les valeurs doivent donc être alloués en plus de la structure.

Attention : afin de simplifier l'implémentation les objets sont numérotés de 0 à n-1 dans les questions de code uniquement.

Question 2 Écrire des fonctions de prototype :

```c
sacados *sacados_alloue(int n);
void sacados_libere(sacados *sac);
```

qui vont respectivement allouer une instance du sac à dos pour n objets et libérer un sac à dos précédemment alloué.

Les deux fonctions utilisées successivement ne doivent pas produire de fuite de mémoire.

Corrigé

```c
sacados *sacados_alloue(int n)
{
    sacados *s = malloc(sizeof(sacados));
    s->nobjets = n;
    s->poids = malloc(sizeof(int) * n);
    s->valeurs = malloc(sizeof(int) * n);
    return s;
}

void sacados_libere(sacados *sac)
{
    free(sac->poids);
    free(sac->valeurs);
    free(sac);
}
```

On va étudier différents critères pour trier les objets afin d'élaborer un algorithme glouton. Comme on ne peut pas passer des fonctions en argument, dans les limites du programme, on ne peut pas faire en sorte facilement de trier selon différents paramètres. On va considérer ici qu'on a défini une fonction

```c
bool sacados_cmp(sacados *sac, int i, int j);
```

qui compare selon le critère retenu les deux objets d'indices i et j et renvoie un booléen indiquant si i est plus petit strictement que j.

Question 3 Écrire une fonction de prototype

```c
int *tri_objets(sacados *sac);
```

qui renvoie un tableau de n éléments `t` où `t[i]` est l'indice du $i$ème objet dans l'ordre croissant induit par la fonction de comparaison.

Note : on pourra implémenter le tri que l'on souhaite ici. Notamment, un tri de complexité non optimale sera accepté.

Corrigé

On trie par insertion le tableau des indices $0, \ldots, n-1$.

```
void echange(int *t, int i, int j)
{
    int temp = t[i];
    t[i] = t[j];
    t[j] = temp;
}

int *tri_objets(sacados *sac)
{
    int n = sac->nobjets;
    int *t = malloc(sizeof(int) * n);
    for (int i = 0; i < n; i++) {
        t[i] = i;
    }
    // tri par insertion
    for (int i = 1; i < n; i++) {
        for (int j = i; j > 0 && sacados_cmp(sac, t[j], t[j-1]); j--) {
            echange(t, j, j-1);
        }
    }
    return t;
}
```

Question 4 En déduire une fonction de prototype

```
float *sacados_resout(sacados *sac);
```

qui alloue et renvoie une solution gloutonne au sac à dos fractionnaire.

Le ième élément du tableau sera la proportion du ième objet.

Indication : les valeurs dans ce tableau sont toutes des 0 ou des 1, sauf au plus une valeur dans $]0; 1[$.

Corrigé

```
float *sacados_resout(sacados *sac)
{
    float *p = malloc(sizeof(float) * sac->nobjets);
    for (int i = 0; i < sac->nobjets; i++) {
        p[i] = 0.;
    }
    int *t = tri_objets(sac);

    float reserve = sac->poids_maximal;
    for (int i = 0; i < sac->nobjets; i++)
    {
        int k = t[i];
        if (sac->poids[k] > reserve) {
            p[k] = reserve / sac->poids[k];
            break;
        }
        p[k] = 1.0;
        reserve -= sac->poids[k];
    }

    free(t);
    return p;
}
```

Des mauvais choix gloutons

67.1.1.2.1 Privilégier les grandes valeurs

Dans ce paragraphe, on considère que le choix glouton est l'objet de plus grande valeur.

Question 5 Écrire le corps de la fonction `sacados_cmp` avec cette approche.

Attention, à ce qu'on a supposé quand on a défini `tri_objets`.

Corrigé

Comme `tri_objets` trie dans l'ordre croissant et que `sacados_resout` prend les objets dans cet ordre, l'objet « le plus petit » doit être celui de plus grande valeur.

```
bool sacados_cmp(sacados *sac, int i, int j)
{
    return sac->valeurs[i] > sac->valeurs[j];
}
```

Question 6 Dérouler l'algorithme glouton qu'on peut en déduire sur l'exemple suivant :

- $n = 4$ et $W = 50$
- $w_1 = 10$ et $v_1 = 60$
- $w_2 = 10$ et $v_2 = 60$
- $w_3 = 20$ et $v_3 = 100$
- $w_4 = 30$ et $v_4 = 120$

Question 7 Que peut-on déduire de l'exemple précédent ?

67.1.1.2.2 Privilégier les petits poids

Dans ce paragraphe, on considère que le choix glouton est l'objet de plus petit poids.

Question 8 Écrire le corps de la fonction `sacados_cmp` avec cette approche.

Corrigé

```
bool sacados_cmp(sacados *sac, int i, int j)
{
    return sac->poids[i] < sac->poids[j];
}
```

Question 9 Dérouler l'algorithme glouton qu'on peut en déduire sur l'exemple du paragraphe précédent.

Question 10 Montrer que ce n'est quand même pas un algorithme optimal.

Efficacité relative

Pour chaque objet, on calcule son efficacité $e_i = v_i/w_i$. On trie alors les objets par efficacité décroissante et on sélectionne tous les objets tant qu'on reste sous le poids, le premier objet qui force un dépassement est placé de manière fractionnelle.

Question 11 Écrire le corps de la fonction `sacados_cmp` avec cette approche.

Corrigé

L'objet i est « plus petit » que j quand il est plus efficace : $\frac{v_i}{w_i} > \frac{v_j}{w_j}$, ce qu'on teste sans division (les poids sont strictement positifs).

```
bool sacados_cmp(sacados *sac, int i, int j)
{
```

```
    return sac->valeurs[i] * sac->poids[j] > sac->valeurs[j] * sac->poids[i];
}
```

On va se placer dans le cas où $\sum_{i=1}^{n} w_i > W$, sinon la solution triviale consistant à prendre tous les objets fonctionne.

Question 12 En déduire que si τ est une solution optimale, alors $w(\tau) = W$.

On suppose qu'on a obtenu une solution gloutonne donnée par σ et qu'on dispose également d'une solution optimale τ. Pour simplifier, on peut supposer les objets rangés par efficacité décroissante, ainsi 1 est l'objet d'efficacité maximale.

Question 13 Montrer que si $\sigma(1) > \tau(1)$, alors on peut en déduire une solution optimale τ' telle que $\tau'(1) = \sigma(1)$.

Question 14 Justifier que si τ est optimale pour les objets de 1 à n, alors τ^* la restriction de τ à $[\![2, n]\!]$ est également optimale pour cette instance avec $W - \tau(1)w_1$ comme poids limite.

Question 15 Justifier que σ^*, la restriction de σ à $[\![2, n]\!]$, est également donnée par l'algorithme glouton pour le poids limite $W - \sigma(1)w_1$.

Question 16 En déduire, par récurrence sur le nombre d'objets, que le glouton renvoie une solution optimale.

PARTIE XI

Devoirs MPSI, MP et tronc commun

68 Devoir DS 1 - Typage, listes et récursivité 927
69 Devoir DS 2 - Autour de la décomposition de Zeckendorf 933
70 Devoir DS 4 - Logique et tas 939
71 Devoir DS 1 - Nombres narcissiques et inversions de permutations 945
72 Devoir DS 1 - Poteaux télégraphiques 951
73 Devoir Interro 1 - Programmation Python 957
74 Devoir DS 2 - Correction (circulation automobile) 961
75 Devoir Interro de rentrée - Révisions Python 967

DEVOIR 68

DS 1 - Typage, listes et récursivité

68.1 DS 1 — Typage, listes et récursivité

MPSI option informatique, année scolaire 2021-2022 — devoir surveillé du 12 mars 2022 — durée totale 2h.

Corrigés

La source ne comporte pas de corrigé. Les corrigés des questions de programmation ci-dessous sont des propositions (testées), à relire.

68.1.1 I. Typage

Question 1 Dans une expression conditionnelle `if expr1 then expr2 else expr3`, quelles sont les contraintes de types sur `expr1`, `expr2` et `expr3` ? Quel est alors le type de cette expression et sa valeur ?

Question 2 Quelles sont les contraintes de types sur `expr1` et `expr2` pour que l'expression `(expr1 expr2)` soit une expression bien typée en CAML et quel est alors son type et sa valeur ?

Question 3 Déterminer le type des fonctions suivantes. Certaines définitions sont équivalentes et ne diffèrent que par la syntaxe utilisée. Préciser quelles sont les définitions équivalentes.

```
let f01 x y = x
let f02 x = fun y -> y
let f03 x y = x y
let f04 x y = y x
let f05 x y = x y y
let f06 x y = x (x y)
let f07 = fun x -> fun y -> y (x y)
let f08 x y = (x y) y
let f09 x y = x (y x)
let f10 x y = y (x y)
```

Question 4 Prévoir et donner la réponse de CAML pour chacune des phrases dans la séquence suivante :

```
let a = fun b -> (b 4) / 5
let b a = fun c -> a (fun d -> c + d) + c
let c = b a
let d = c 11
```

68.1.2 II. Type somme

On considère le type entier suivant :

```
type entier =
  | Zero
  | Successeur of entier
```

Question 5 Écrire une fonction `add : entier -> entier -> entier` réalisant l'addition de deux entiers avec ce type.

Corrigé

```
let rec add a b =
  match a with
  | Zero -> b
  | Successeur a' -> Successeur (add a' b)
```

Question 6 Écrire une fonction `mult : entier -> entier -> entier` réalisant la multiplication de deux entiers avec ce type.

Corrigé

```
let rec mult a b =
  match a with
  | Zero -> Zero
  | Successeur a' -> add b (mult a' b)
```

68.1.3 III. Listes

Question 7 Écrire une fonction `produit : int list -> int` qui renvoie le produit des éléments d'une liste d'entiers.

Corrigé

```
let rec produit l =
  match l with
  | [] -> 1
  | x :: q -> x * produit q
```

Question 8 Écrire une fonction `supprime : 'a -> 'a list -> 'a list` qui renvoie une liste identique à celle passée en argument à ceci près que toutes les occurrences éventuelles de l'élément passé en argument ont été supprimées.

Corrigé

```
let rec supprime x l =
  match l with
  | [] -> []
  | y :: q -> if x = y then supprime x q else y :: supprime x q
```

Question 9 Écrire une fonction `avant_dernier : 'a list -> 'a` qui renvoie l'avant-dernier élément d'une liste de taille au moins deux. Le filtrage devra être exhaustif et on lèvera une exception (avec `failwith`) si la liste est trop courte.

Corrigé

```
let rec avant_dernier l =
  match l with
  | [] | [_] -> failwith "liste trop courte"
  | [x; _] -> x
  | _ :: q -> avant_dernier q
```

68.1.4 IV. Tri fusion

Question 10 Implémenter une fonction `division : 'a list -> 'a list * 'a list` qui divise une liste passée en argument en deux sous-listes de même taille, à un près. La première sous-liste doit contenir les éléments

d'indices pairs et la deuxième ceux d'indices impairs, dans le même ordre relatif que dans la liste initiale. Par exemple :

```
division [1; 4; 2; 6; 7; 1; 2; 4; 2]
- : int list * int list = ([1; 2; 7; 2; 2], [4; 6; 1; 4])
```

✍ **Corrigé**

```
let rec division l =
  match l with
  | [] -> [], []
  | [x] -> [x], []
  | x :: y :: q ->
    let l1, l2 = division q in
    x :: l1, y :: l2
```

Question 11 Implémenter une fonction `fusion : 'a list -> 'a list -> 'a list` qui prend en argument deux listes, supposées triées, et qui renvoie la liste triée formée des éléments des deux listes (avec doublons). Par exemple :

```
fusion [1; 2; 2; 2; 7] [1; 4; 4; 6]
- : int list = [1; 1; 2; 2; 2; 4; 4; 6; 7]
```

✍ **Corrigé**

```
let rec fusion l1 l2 =
  match l1, l2 with
  | [], _ -> l2
  | _, [] -> l1
  | x :: q1, y :: q2 ->
    if x <= y
    then x :: fusion q1 l2
    else y :: fusion l1 q2
```

Question 12 Implémenter une fonction `tri_fusion : 'a list -> 'a list` qui trie la liste reçue en argument. Par exemple :

```
tri_fusion [1; 4; 2; 6; 7; 1; 2; 4; 2]
- : int list = [1; 1; 2; 2; 2; 4; 4; 6; 7]
```

✍ **Corrigé**

```
let rec tri_fusion l =
  match l with
  | [] | [_] -> l
  | _ ->
    let l1, l2 = division l in
    fusion (tri_fusion l1) (tri_fusion l2)
```

68.1.5 V. Dénombrement de $\mathbb{N}^2$

On se propose dans cette partie d'étudier une fonction permettant de numéroter les éléments de $\mathbb{N}^2$.

On considère la fonction *carre* définie par :

$$\forall (x,y) \in \mathbb{N}^2, \quad carre(x,y) = (\max(x,y))^2 + \max(x,y) + y - x$$

Question 13 Définir en CAML la fonction `carre : int -> int -> int`.

✍ **Corrigé**

```
let carre x y =
  let m = max x y in
  m * m + m + y - x
```

La fonction *carre* permet d'effectuer une numérotation dite en carré des éléments de $\mathbb{N}^2$.

Question 14 Représenter dans le plan les points numérotés entre 0 et 30 et justifier le nom de la fonction.

La fonction *carre_rec* est définie sur $\mathbb{N} \times \mathbb{N}$ par $\forall (m,n) \in \mathbb{N}^2$,

$$carre\_rec(m,n) = \begin{cases} m^2 & \text{lorsque } n = 0 \\ carre\_rec(m, n-1) + 1 & \text{lorsque } 1 \leq n \leq m \\ carre\_rec(m+1, n) + 1 & \text{lorsque } n > m \end{cases}$$

Question 15 Définir en CAML la fonction `carre_rec : int -> int -> int`.

✍ **Corrigé**

```
let rec carre_rec m n =
  if n = 0 then m * m
  else if n <= m then carre_rec m (n-1) + 1
  else carre_rec (m+1) n + 1
```

Question 16 Montrer que la fonction *carre_rec* est égale à la fonction *carre*.

Question 17 Prouver la terminaison de la fonction *carre_rec*.

68.1.6 VI. Ackermann

La fonction d'Ackermann A est définie sur $\mathbb{N} \times \mathbb{N}$ par :

$$\{A(0,p) = p+1$$
$$A(n,0) = A(n-1,1) \text{ si } n > 0$$
$$A(n,p) = A(n-1, A(n,p-1)) \text{ si } n > 0 \text{ et } p > 0$$

Question 18 Calculer $A(0,0)$ puis $A(2,2)$.

Question 19 Implémenter une fonction calculant A en CAML, elle sera du type `int -> int -> int`.

✍ **Corrigé**

```
let rec ackermann n p =
  if n = 0 then p + 1
  else if p = 0 then ackermann (n-1) 1
  else ackermann (n-1) (ackermann n (p-1))
```

68.1.7 VII. Exponentiation rapide

Question 20 Définir une fonction `exp : int -> int -> int` qui étant donné deux entiers x et n calcule x^n de manière naïve, en faisant des multiplications successives par x.

✍ **Corrigé**

```
let rec exp x n =
  if n = 0 then 1
  else x * exp x (n-1)
```

Il est possible d'améliorer ce calcul en remarquant que :

$$x^{2n} = (x^n)^2 \qquad x^{2n+1} = x \times (x^n)^2$$

On peut donc calculer récursivement x^n pour en déduire x^{2n} avec une multiplication de plus et x^{2n+1} avec deux multiplications de plus.

Question 21 Définir une fonction `exp_rap : int -> int -> int` qui réalise cette méthode de calcul.

✍ **Corrigé**

```
let rec exp_rap x n =
  if n = 0 then 1
  else begin
    let y = exp_rap x (n / 2) in
    if n mod 2 = 0 then y * y
    else x * y * y
  end
```

Question 22 Justifier, avec précision et rigueur, la terminaison et la correction de cette fonction.

Question 23 Déterminer, en fonction n, le nombre total d'appels à la fonction pour calculer x^n. On pourra noter C_n le nombre d'appels pour calculer `exp_rap n` et chercher une relation de récurrence vérifiée par la suite $(C_n)_{n\in\mathbb{N}}$.

DEVOIR 69

DS 2 - Autour de la décomposition de Zeckendorf

69.1 DS 2 — Autour de la décomposition de Zeckendorf

MPSI option informatique, année scolaire 2021-2022 — durée totale 2h.

Consignes

Tout document et dispositif électronique interdit.

69.1.1 I. Nombres de Fibonacci

On rappelle la définition de la suite de Fibonacci $(F_n)_{n\geq 0}$:

$$\begin{cases} F_0 = 0 \\ F_1 = 1 \\ F_n = F_{n-1} + F_{n-2} \text{ pour } n \geq 2 \end{cases}$$

Question 1 Écrire une fonction `fibonacci : int -> int` telle que l'appel `fibonacci n` renvoie la valeur de F_n.

Cette fonction ne devra pas être récursive mais elle devra utiliser des boucles et des références.

Corrigé

```
let fibonacci n =
  let f = ref 0 in
  let f' = ref 1 in
  for i = 1 to n do
    let t = !f' in
    f' := !f + !f';
    f := t
  done;
  !f
```

Question 2 Justifier la correction de cette fonction.

Corrigé

On a l'invariant de boucle $I(i) := f = F_{i-1} \wedge f' = F_i$. On suppose, comme on l'a fait en classe, que la valeur de i change en fin d'itération (équivalent à un `while`).

- Initialisation : au départ, on suppose $i = 1$ et on a bien $f = F_0 = 0$ et $f' = 1 = F_1$.
- Hérédité : si $I(i)$ est vérifié au début d'une itération, on a en fin d'itération $f = F_i$ et $f' = F_{i-1} + F_i = F_{i+1}$. On a donc bien $I(i+1)$ en sortie d'itération.

En sortie de boucle, on a donc $I(n+1)$ et ainsi $f = F_n$.

Question 3 Déterminer la complexité de cette fonction en fonction de n.

Corrigé

Cette fonction effectue n itérations effectuant un nombre constant d'opérations, elle est donc en $O(n)$.

Question 4 Écrire une fonction `sous_fibonacci : int -> int` telle que `sous_fibonacci n` renvoie l'unique F_p tel que

$$F_p \leq n < F_{p+1}$$

Corrigé

Rien n'était demandé sur la complexité, mais il semble préférable de repartir de la question précédente :

```
let sous_fibonacci n =
  let f = ref 0 in
  let f' = ref 1 in
  while !f' <= n do
    let t = !f' in
    f' := !f + !f';
    f := t
  done;
  !f
```

69.1.2 II. Arbres de Fibonacci

On considère le type Caml suivant pour représenter des arbres binaires stricts :

```
type arbre =
  | Feuille
  | Noeud of arbre * arbre
```

Question 5 Écrire une fonction `nb_feuilles : arbre -> int` qui renvoie le nombre de feuilles d'un arbre.

Corrigé

```
let rec nb_feuilles a =
  match a with
  | Feuille -> 1
  | Noeud (g, d) -> nb_feuilles g + nb_feuilles d
```

On définit les arbres de Fibonacci $(A_n)_{n\in\mathbb{N}}$ de la manière récursive suivante :

- A_0 et A_1 sont des feuilles ;
- si $n \geq 2$, A_n est un nœud comportant A_{n-1} comme fils gauche et A_{n-2} comme fils droit.

Question 6 Dessiner les arbres A_2, A_3, A_4 et A_5.

Corrigé

Les nœuds sont représentés par des disques et les feuilles par des carrés.

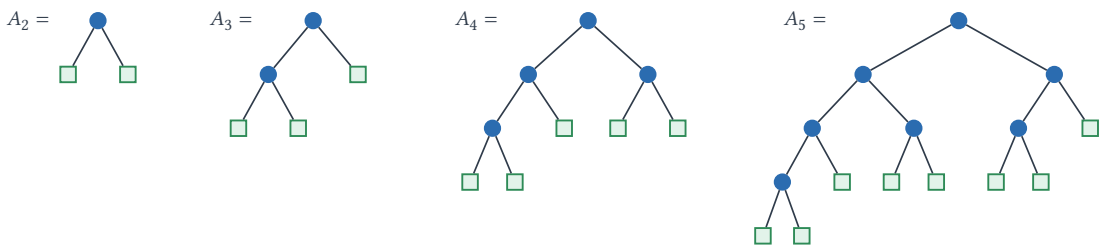


Figure 69.1 — Les arbres de Fibonacci A_2 à A_5 : nœuds en disques, feuilles en carrés.

Question 7 Écrire en Caml une fonction `arbre_fibonacci : int -> arbre` telle que l'appel `arbre_fibonacci n` renvoie l'arbre A_n.

Corrigé

```
let rec arbre_fibonacci n =
  if n <= 1
  then Feuille
  else Noeud (arbre_fibonacci (n-1), arbre_fibonacci (n-2))
```

Question 8 Montrer que la complexité de cette fonction est en $O(2^n)$.

On pourra remarquer qu'en dehors des appels récursifs tout se fait en temps constant et chercher à majorer le nombre d'appels récursifs.

Corrigé

On procède comme l'explication et on note C_n le nombre d'appels récursifs pour calculer le n-ième arbre. On a $C_0 = C_1 = 0$ et $C_n = 2 + C_{n-1} + C_{n-2}$.

Comme l'appel pour n nécessite de résoudre l'appel pour $n-1$, on a nécessairement C_n croissante et $C_n \leq 2(1 + C_{n-1})$ donc $C_n + 2 \leq 2(C_{n-1} + 2)$ et par une récurrence rapide $C_n + 2 \leq 2^n$ donc $C_n \leq 2^n - 2 = O(2^n)$.

Question 9 Quelle est, pour $n \in \mathbb{N}$, la hauteur de l'arbre A_n ? Justifier rigoureusement.

Corrigé

On a $h(A_0) = h(A_1) = 0$ et pour $n \geq 2$, $h(A_n) = 1 + \max(h(A_{n-1}), h(A_{n-2}))$. On en déduit que $h(A_n)$ est croissante et donc que $h(A_n) = 1 + h(A_{n-1})$ d'où $h(A_n) = n - 1$ pour $n \geq 1$.

Question 10 Exprimer le nombre de feuilles de A_n à l'aide des nombres de la suite de Fibonacci $(F_n)_{n\in\mathbb{N}}$.

Corrigé

De même si $f(A)$ est le nombre de feuilles de A, on a $f(A_0) = f(A_1) = 1$ et $f(A_n) = f(A_{n-1}) + f(A_{n-2})$. Donc $f(A_n) = F_{n+1}$: mêmes deux premières valeurs et même récurrence.

Question 11 Exprimer le nombre de nœuds de A_n à l'aide des nombres de la suite de Fibonacci $(F_n)_{n\in\mathbb{N}}$.

Corrigé

On note $N(A)$ le nombre de nœuds de A.

On a $N(A_0) = N(A_1) = 0$ et $N(A_n) = 1 + N(A_{n-1}) + N(A_{n-2})$.

En posant $u_n = N(A_n) + 1$, on a pour $n \geq 2$, $u_n = u_{n-1} + u_{n-2}$ et $u_0 = u_1 = 1$. Donc $u_n = F_{n+1}$ et $N(A_n) = F_{n+1} - 1$.

Question 12 Montrer que les arbres de Fibonacci sont équilibrés : en tout nœud de l'arbre les hauteurs des sous-arbres gauche et droit diffèrent au plus de 1.

Corrigé

C'est une conséquence directe du calcul de $h(A_n) = n - 1$ et de la définition.

Question 13 Écrire en Caml une fonction `arbre_fibonacci_bis : int -> arbre` de complexité $O(n)$. Comment est-il possible d'avoir une fonction de complexité linéaire en n qui crée un arbre de taille exponentielle en n ?

Corrigé

```
let arbre_fibonacci_bis n =
  let f = ref Feuille in
  let f' = ref Feuille in
  for i = 1 to n do
    let t = !f' in
    f' := Noeud (!f', !f);
    f := t
  done;
  !f
```

En fait, quand on écrit `Noeud (!f', !f)` on ne copie pas les arbres, on se contente d'y faire référence. En mémoire, les valeurs ne sont pas dupliquées.

69.1.3 III. Décomposition de Zeckendorf

Pour $n \in \mathbb{N}^*$, on appelle décomposition de Zeckendorf de n une décomposition de n comme somme de termes distincts de la suite de Fibonacci d'indices supérieurs ou égaux à 2, de manière à ce qu'il n'y ait pas deux termes d'indices consécutifs dans la somme.

Théorème 69.1 — (Zeckendorf - 1972)

Soit $n \in \mathbb{N}^*$, il existe une unique suite $c_0, c_1, ..., c_{k-1} \in \mathbb{N} \setminus \{0, 1\}$ vérifiant $\forall i \in [\![0, k-2]\!], c_{i+1} > c_i + 1$ telle que $n = \sum_{i=0}^{k-1} F_{c_i}$.

Par exemple, $2 + 5 = F_3 + F_5$ est une décomposition de Zeckendorf de 7, avec $c_0 = 3$ et $c_1 = 5$. En revanche, $3 + 5 = F_4 + F_5$ n'est pas une décomposition de Zeckendorf de 8 car F_4 et F_5 sont deux termes consécutifs de la suite de Fibonacci.

Question 14 Déterminer une décomposition de Zeckendorf de 20, 21 et 22.

Corrigé

Les premiers termes de Fibonacci sont $F_0 = 0, F_1 = 1, F_2 = 1, F_3 = 2, F_4 = 3, F_5 = 5, F_6 = 8, F_7 = 13, F_8 = 21$ donc $20 = 13 + 5 + 2 = F_7 + F_5 + F_3$, $21 = F_8$ et $22 = 21 + 1 = F_8 + F_2$.

Question 15 Montrer qu'il n'y a pas unicité de la décomposition si l'on autorise des indices de termes consécutifs.

Corrigé

On a $8 = F_6 = F_4 + F_5$.

On considère le programme `myst` suivant.

```
let myst n res =
  let i = ref 0 in
  let j = ref n in
  while !j > 0 do
    let f = sous_fibonacci !j in
    res.(!i) <- f;
    j := !j - f;
    i := !i + 1
  done;
  !i
```

Question 16 On suppose que `tab` est un tableau d'entiers de taille suffisante. Décrire précisément ce que renvoie `myst 17 tab` et ce que contient le tableau `tab` après l'appel.

Corrigé

On a les valeurs suivantes (au début de chaque itération, puis en sortie) :

j	i	f
17	0	13
4	1	3
1	2	1
0	3	

L'appel renvoie donc 3 et on a placé `[|13;3;1|]` en préfixe de `tab`.

Question 17 Justifier très très soigneusement que ce programme termine.

Corrigé

j est un variant, en effet, comme on boucle, on a $j > 0$ et donc il existe $1 \leq F_p \leq j$ ainsi $j \leftarrow j - F_p$ va faire diminuer j strictement à chaque itération.

Question 18 Expliquer rapidement ce que calcule ce programme en détaillant le rôle des paramètres et de la valeur de retour. On ne demande pas de le justifier précisément ni de trouver un invariant.

Corrigé

Cette fonction va donc décomposer n de manière gloutonne en prenant à chaque fois le plus grand $F_p \leq n$. j est la valeur courante à décomposer, i est le compteur du nombre de termes de la décomposition courante, f est le plus grand $F_p \leq j$.

La valeur de retour est donc le nombre de termes de la décomposition.

69.1.4 IV. Codage de Fibonacci

On appelle codage de Fibonacci d'un entier $n \in \mathbb{N}^*$ un tableau `code`, composé de 0 et de 1, indiquant par des 1 les indices des termes de la suite de Fibonacci utilisés dans la décomposition de Zeckendorf de n. Par exemple `code.(2)` indique si F_2 est utilisé dans la décomposition de Zeckendorf de n ou non et `[|0; 0; 0; 1; 0; 1|]` est le codage de Fibonacci de 7. Pour avoir un seul tableau codage pour la décomposition d'un entier, on impose que la dernière valeur du tableau soit toujours un 1.

On remarque que par définition de la décomposition de Zeckendorf, `code` ne peut contenir deux 1 consécutifs et commence toujours par deux 0.

Question 19 Écrire une fonction `est_code : int -> int array -> bool` telle que l'appel `est_code n tab` pour un tableau `tab`, de taille $n \geq 0$, composé de 0 et de 1, renvoie `true` si le tableau `tab` est un codage de Fibonacci d'un entier strictement positif et `false` sinon.

Corrigé

Un codage a au moins 3 cases (deux 0 initiaux et un 1 final) :

```
let est_code n tab =
  if n < 3 then false
  else begin
    let valide = ref (tab.(0) = 0 && tab.(1) = 0 && tab.(n-1) = 1) in
    for i = 2 to n-2 do
      if tab.(i) = 1 && tab.(i+1) = 1
      then valide := false
    done;
    !valide
  end
```

On remarque que si `code` est un codage de Fibonacci, de taille $k \geq 3$, alors c'est le codage de Fibonacci de l'entier $n = \sum_{i=0}^{k-1} \texttt{code.(i)} \times F_i$.

On propose le programme suivant, qui suppose que le tableau `code` de taille k est un codage de Fibonacci d'un nombre strictement positif et que `fib` est un tableau de taille k contenant les k premiers entiers de la suite de Fibonacci : pour tout $m \in [\![0, k-1]\!]$, `fib.(m)` $= F_m$.

```
let decode k code fib =
  let j = ref 2 in
  let n = ref 0 in
  while !j < k do
    n := !n + code.(!j) * fib.(!j);
    j := !j + 1
  done;
  !n
```

Question 20 Expliquer simplement, en une phrase, ce que fait ce programme.

Corrigé

Il calcule la somme $\sum_{i=0}^{k-1}$ `code.(i)` $\times F_i$.

Question 21 En exhibant un variant de boucle, que l'on justifiera rapidement, montrer que ce programme termine.

Corrigé

Le variant est $k - j$ qui diminue strictement car j s'incrémente mais k est fixe.

Question 22 Montrer très soigneusement que « $j \leq k$ et $n = \sum_{i=0}^{j-1}$ `code.(i)` $\times F_i$ » est un invariant de boucle pour la boucle `while` de ce programme.

Corrigé

On note $I(j, n) \coloneqq j \leq k$ et $n = \sum_{i=0}^{j-1} \texttt{code}_i \times F_i$ où $\texttt{code}_i$ désigne `code.(i)`.

- Initialisation : on a $I(2, 0)$ car $0 = 0 \times F_0 + 0 \times F_1$.
- Si $I(j, n)$ est vérifié en début d'itération, on a également $I(j+1, n + \texttt{code}_j \times F_j)$ en fin d'itération vu que $j < k$ donc $j + 1 \leq k$ et $n + \texttt{code}_j \times F_j = \sum_{i=0}^{j-1} \texttt{code}_i \times F_i + \texttt{code}_j \times F_j = \sum_{i=0}^{j} \texttt{code}_i \times F_i$.

Question 23 Montrer la correction de ce programme.

Corrigé

Le programme termine et en sortie de boucle, on a $j = k$ donc comme $I(k, n)$ est vérifié, c'est que $n = \sum_{i=0}^{k-1} \texttt{code}_i \times F_i$.

DEVOIR 70

DS 4 - Logique et tas

70.1 DS 4 — Logique et tas

MPSI option informatique, année scolaire 2021-2022 — durée totale 2h.

Consignes

Tout document et dispositif électronique interdit.

Corrigés

La source ne comporte pas de corrigé. Les corrigés des questions de programmation ci-dessous sont soit repris du fichier de code de l'auteur (`DS_OI4.ml`), soit des propositions (testées), à relire.

70.1.1 I. Logique

Vous participez à un concours de mathématiques comportant une partie de raisonnement logique. Plusieurs orateurs font des déclarations et vous devez répondre à des questions en vous appuyant sur des informations déduites de ces déclarations. La règle suivante s'applique : « Les orateurs sont de trois natures : les véridiques, les menteurs et les changeants. Les véridiques disent toujours la vérité, les menteurs mentent toujours, et les changeants disent en alternance une vérité et un mensonge (c'est-à-dire, soit une vérité, puis un mensonge, puis une vérité, etc. ; soit un mensonge, puis une vérité, puis un mensonge, etc.). Pendant tout le concours, les orateurs ne peuvent pas changer de nature. »

Les épreuves comportent deux phases :

- Les différents orateurs font plusieurs déclarations dont l'analyse permet de déterminer la nature de chaque orateur (véridique, menteur, changeant commençant par dire la vérité, ou changeant commençant par dire un mensonge).
- Les orateurs font une seconde série de déclarations. Puis, vous devez répondre à des questions en exploitant les informations contenues dans ces déclarations.

Question 1 Dans la première phase, quel est le nombre minimum de déclarations que doit faire chaque orateur pour qu'il soit possible de déterminer sa nature ? Justifier votre réponse.

Question 2 Soit un orateur A qui fait une suite de n déclarations A_i. Proposer des formules du calcul des propositions A_V, A_M, A_{CV} et A_{CM} qui permettent de caractériser la nature de A (respectivement véridique, menteur, changeant commençant par dire la vérité, ou changeant commençant par dire un mensonge).

Vous participez à une première épreuve avec un orateur A qui fait les déclarations suivantes :

- J'aime le rouge mais pas le bleu.
- Soit j'aime le rouge, soit j'aime le vert.
- Si j'aime le rouge et le vert, alors j'aime le bleu.

Nous noterons R, V et B les variables propositionnelles associées au fait que l'orateur aime le rouge, le vert ou le bleu. Nous noterons A_1, A_2 et A_3 les formules propositionnelles associées aux déclarations de A.

Question 3 Représenter les déclarations de l'orateur sous la forme de formules du calcul des propositions A_1, A_2 et A_3 dépendant des variables R, V et B.

Question 4 Appliquer les formules permettant de caractériser la nature des orateurs A_V, A_M, A_{CV} et A_{CM} que vous avez proposées pour la question 2 pour l'orateur A dépendant des variables A_1, A_2 et A_3.

Question 5 En utilisant le calcul des propositions (résolution avec les tables de vérité), déterminer la nature de l'orateur A. Quelles sont les couleurs qu'aime A ?

70.1.2 II. Tas

Dans cette partie, on cherche à créer une structure de données **immuable** appelée un *tas-max* pouvant contenir des éléments d'un type `'a` ordonné **totalement** et défini par le type `'a tas` des tas et les opérations suivantes :

- `tas_vide : 'a tas` une valeur représentant un tas vide ;
- `ajoute : 'a tas -> 'a -> 'a tas` telle que `ajoute t x` ajoute l'élément `x` sur le tas `t` ;
- `retire : 'a tas -> 'a * 'a tas` telle que `retire t` renvoie un couple `(x, t')` où `x` est un plus grand élément parmi les éléments de `t` et `t'` est un tas dans lequel on a retiré `x`.

Tri par tas

Sans même savoir implémenter un tas, on peut l'utiliser pour obtenir un algorithme de tri :

- Étant donné des éléments à trier, on les ajoute tous dans un tas.
- Tant que le tas est non vide, on retire un élément du tas et on le place dans la sortie.

Comme on a la garantie d'avoir un élément plus grand que ceux qui restent à chaque fois qu'on en retire un, on est certain que la sortie sera triée dans l'ordre décroissant.

Question 6 Écrire une fonction `remplit : 'a list -> 'a tas -> 'a tas` qui prend en entrée une liste `l` et un tas `t` et renvoie un tas `t'` obtenu en ajoutant tous les éléments de `l` au tas `t`.

Corrigé

```
let rec remplit l t =
  match l with
  | [] -> t
  | x :: q -> remplit q (ajoute t x)
```

Question 7 Écrire une fonction `vide : 'a tas -> 'a list` qui vide un tas pour obtenir la liste de ses éléments dans l'ordre décroissant.

Corrigé

On compare au tas vide (avec l'implémentation par arbres de la suite, on peut aussi filtrer sur `Nil`) :

```
let rec vide t =
  if t = tas_vide then []
  else begin
    let x, t' = retire t in
    x :: vide t'
  end
```

Question 8 Écrire une fonction `tri : 'a list -> 'a list` qui renvoie une liste triée dans l'ordre décroissant à l'aide d'un tas.

Corrigé

```
let tri l = vide (remplit l tas_vide)
```

Question 9 Si on considère que `ajoute` et `retire` sont en complexité temporelle $f(n)$ où n est la taille du tas et où f est croissante, quelle est la complexité de ce tri ?

Implémentation naïve

Pour représenter un tas, on va ici choisir une implémentation naïve :

```
type 'a tas = 'a list

let tas_vide = []
let ajoute t x = x :: t
```

Pour retirer, il va donc être nécessaire de trouver le maximum et de l'extraire. On décompose cela en deux fonctions.

Question 10 Écrire une fonction `maximum : 'a list -> 'a` qui renvoie la plus grande valeur dans la liste donnée en entrée.

✍ **Corrigé**

```
let rec maximum l =
  match l with
  | [] -> failwith "liste vide"
  | [x] -> x
  | x :: q -> max x (maximum q)
```

Question 11 Écrire une fonction `supprime : 'a list -> 'a -> 'a list` qui supprime la première occurrence d'un élément dans une liste si elle le contient (sinon, on renvoie la liste inchangée).

Exemple : `supprime [2;1;3;1] 1` renvoie `[2;3;1]`.

✍ **Corrigé**

```
let rec supprime l x =
  match l with
  | [] -> []
  | y :: q -> if x = y then q else y :: supprime q x
```

Question 12 Écrire la fonction `retire` avec cette implémentation de tas à l'aide des deux fonctions précédentes.

✍ **Corrigé**

```
let retire t =
  let m = maximum t in
  m, supprime t m
```

Question 13 Quelle est la complexité de la fonction précédente ?

Question 14 Quelle est alors la complexité du tri par tas avec cette implémentation ?

Tas binaires

On va maintenant donner une implémentation des tas avec des arbres binaires.

On considère le type :

```
type 'a tas = Noeud of 'a tas * 'a * 'a tas | Nil

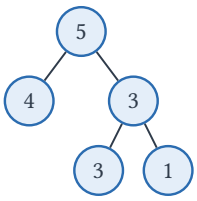
let tas_vide = Nil
```

Et on ne va considérer que des arbres dont l'étiquette de la racine est toujours la plus grande des étiquettes et dont les fils gauches et droits sont aussi des tas.

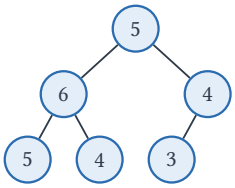
Si on note $\eta(t)$ les étiquettes des nœuds de t, on a donc la propriété *tas* définie ainsi :

$$tas(t) \coloneqq t = Nil \vee (t = Noeud(g, x, d) \wedge x = \max \eta(t) \wedge tas(g) \wedge tas(d))$$

Ainsi l'arbre



est un tas mais



n'en est pas un.

Question 15 Écrire une fonction `arbre_max : 'a tas -> 'a` qui renvoie le plus grand élément d'un arbre non vide.

Attention : on ne suppose pas encore que l'arbre est un tas.

Corrigé

```
let rec arbre_max t =
  match t with
   | Nil -> failwith "arbre vide"
   | Noeud (Nil, x, Nil) -> x
   | Noeud (Nil, x, d) -> max x (arbre_max d)
   | Noeud (g, x, Nil) -> max x (arbre_max g)
   | Noeud (g, x, d) -> max x (max (arbre_max g) (arbre_max d))
```

Question 16 En déduire une fonction `est_un_tas : 'a tas -> bool` qui indique si un arbre est bien un tas.

Corrigé

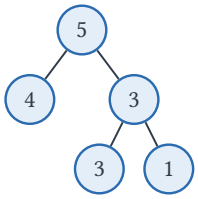
```
let rec est_un_tas t =
  match t with
   | Nil -> true
   | Noeud (g, x, d) -> x = arbre_max t && est_un_tas g && est_un_tas d
```

Pour l'ajout et l'extraction de la racine, on va utiliser une même opération : la fusion de deux tas. Il s'agit d'une opération qui, étant donnés deux tas t_1 et t_2 va renvoyer un tas comportant tous les éléments de t_1 et de t_2.

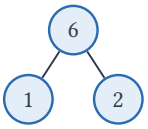
Pour cela, on procède ainsi :

- Si t_1 est vide, on renvoie t_2.
- Si t_2 est vide, on renvoie t_1.
- Si $t_1 = Noeud(g_1, x, d_1)$ et $t_2 = Noeud(g_2, y, d_2)$ et $x > y$, on va fusionner d_1 et t_2 pour obtenir t' et on renvoie $Noeud(g_1, x, t')$.
- On procède symétriquement si $x \leq y$ en renvoyant $Noeud(g_2, y, t')$.

Question 17 Réaliser la fusion des deux tas



et



Question 18 Écrire une fonction `fusionne : 'a tas -> 'a tas -> 'a tas` qui fusionne deux tas.

Corrigé

```
let rec fusionne t1 t2 =
  match t1, t2 with
  | Nil, _ -> t2
  | _, Nil -> t1
  | Noeud (g1, x, d1), Noeud (g2, y, d2) ->
    if x > y
    then Noeud (g1, x, fusionne t2 d1)
    else Noeud (g2, y, fusionne t1 d2)
```

Question 19 Exprimer la complexité de cette fonction en fonction des deux arbres de tas fusionnés.

Pour ajouter un élément à un tas, on va fusionner le tas avec une feuille étiquetée par l'élément.

Question 20 Écrire une fonction `ajoute : 'a tas -> 'a -> 'a tas`.

Corrigé

```
let ajoute t x = fusionne t (Noeud (Nil, x, Nil))
```

Pour retirer un élément d'un tas non vide, on renvoie l'étiquette de la racine et le tas obtenu par fusion de ses deux sous-arbres.

Question 21 Écrire une fonction `retire : 'a tas -> 'a * 'a tas`.

Corrigé

```
let retire t =
  match t with
  | Nil -> failwith "vide"
  | Noeud (g, x, d) -> x, fusionne g d
```

Question 22 Exprimer la complexité de `ajoute` et `retire`.

Question 23 Détailler l'exécution du tri par tas avec la liste `[1;6;2;4;3;7]`. On donnera tous les tas apparaissant au cours du déroulement.

DEVOIR 71

DS 1 - Nombres narcissiques et inversions de permutations

71.1 DS 1 — Nombres narcissiques et inversions de permutations

Informatique tronc commun (MPSI), année scolaire 2021-2022 — durée totale 2h.

Consignes

Tout document et dispositif électronique interdit.

71.1.1 I. Nombres narcissiques

On ne considère dans cette partie que des valeurs entières dans $\mathbb{N}$.

On s'efforcera de ne pas se répéter en utilisant les fonctions précédemment définies.

Question 1 Montrer que si $n, k \in \mathbb{N}$, le k-ième (en partant de la droite) chiffre dans l'écriture décimale de n est donné par $c_k = (n/10^k) \bmod 10$ où l'on rappelle que $a \bmod b$ est le reste dans la division euclidienne de a par b.

Corrigé

Erreur d'énoncé : il manque une partie entière, il faut lire $c_k = \lfloor n/10^k \rfloor \bmod 10$.

On sait qu'on peut écrire $n = \sum_{i=0}^{p} c_i 10^i$ en écriture décimale à $p+1$ chiffres où $n < 10^{p+1}$.

On a donc $\lfloor n/10^k \rfloor = c_k + 10 \sum_{i=k+1}^{p} c_i 10^{i-k-1} \equiv c_k \ [10]$.

Question 2 En déduire une fonction `chiffre` telle que l'appel à `chiffre(n,k)` renvoie la valeur de c_k (on utilisera les opérations usuelles de division euclidienne présentes dans Python, et ainsi, on ne reprogrammera pas la division euclidienne).

Corrigé

Cette fonction est la traduction directe de la formule précédente :

```
def chiffre(n, k):
    return (n // (10**k)) % 10
```

Question 3 Montrer que si $n \in \mathbb{N}$, il existe $N \in \mathbb{N}$ tel que $\forall k \geq N, c_k = 0$ (i.e. que la suite des chiffres est presque nulle).

Corrigé

Il s'agit ici de montrer la finitude de l'écriture d'un entier.

On sait que la suite (10^k) tend vers $+\infty$ donc il existe un N à partir duquel $10^k > n$, et alors $\lfloor n/10^k \rfloor = 0$ donc $c_k = 0$.

Question 4 En déduire une fonction `degre` telle que l'appel à `degre(n)` où $n \neq 0$ renvoie le plus grand k tel que $c_k \neq 0$.

Corrigé

Attention, il ne faut pas chercher le plus petit k pour lequel $c_k = 0$ et prendre le rang d'avant, car il peut très bien y avoir des 0 au milieu de l'écriture du nombre. Par exemple 103 est de degré 2 et non 0.

Il faut donc procéder par division de n et trouver le plus grand k tel que le quotient de n par 10^k soit non nul.

On calcule les quotients de n en divisant progressivement par 10 à chaque étape :

```
def degre(n):
    d = 0
    while n >= 10:
        d = d + 1
        n = n // 10
    return d
```

Question 5 En déduire une fonction `chiffre_dominant` telle que l'appel à `chiffre_dominant(n)` où $n \neq 0$ renvoie le chiffre non nul situé le plus à gauche dans l'écriture décimale de n.

Corrigé

Par définition il suffit donc de renvoyer $c_{\deg n}$ avec le degré obtenu à l'étape précédente :

```
def chiffre_dominant(n):
    return chiffre(n, degre(n))
```

Question 6 Écrire une fonction `somme_chiffres` telle que l'appel à `somme_chiffres(n)` calcule la somme des chiffres dans l'écriture décimale de n.

Corrigé

Dans cette question et les suivantes, on va se servir de `degre` pour fournir la plage d'indices à prendre en compte pour obtenir les chiffres significatifs de n.

```
def somme_chiffres(n):
    s = 0
    for k in range(degre(n)+1):
        s = s + chiffre(n, k)
    return s
```

Question 7 On rappelle que l'on peut créer un tableau t de m cases contenant la valeur 0 à l'aide de l'instruction

```
t = [ 0 ] * m
```

Écrire une fonction `chiffres` telle que l'appel à `chiffres(n)` renvoie le tableau des chiffres dans l'écriture décimale de n, telle qu'à l'indice i on ait le chiffre c_i suivant la numérotation de la question 1.

Corrigé

La structure de la réponse à cette question est proche de la question précédente, une fois le tableau initialisé comme dans l'énoncé (attention il contient $\deg(n) + 1$ cases car on va de 0 à $\deg(n)$ inclus !) on le remplit dans une boucle :

```python
def chiffres(n):
    t = [ 0 ] * (degre(n)+1)
    for k in range(degre(n)+1):
        t[k] = chiffre(n, k)
    return t
```

Question 8 On dit qu'un entier n est narcissique si son écriture décimale comporte p chiffres et que $\sum_{i=1}^{p} c_i^p = n$ où c_i est le i-ème chiffre de n.

Déterminer une fonction `est_narcissique` telle que `est_narcissique(n)` renvoie `True` si n est narcissique et `False` sinon.

On rappelle encore une fois qu'il est demandé de réutiliser les fonctions précédentes (notamment ici `degre` et `chiffre`...).

Corrigé

Ici il faut faire attention car $p = \deg(n) + 1$. On fait donc un calcul de la somme comme précédemment mais en rajoutant la puissance p et ensuite on renvoie le test : n est-il égal à cette somme ?

```python
def est_narcissique(n):
    s = 0
    p = degre(n)+1
    for k in range(p):
        s = s + chiffre(n, k)**p
    return s == n
```

Attention ce n'est pas la peine d'écrire ce genre de choses :

```python
if (s == n) == True:
    return True
else:
    return False
```

le code précédent `return s == n` renvoie directement la même valeur !

Les nombres narcissiques non nuls inférieurs à 10 000 sont $1, \ldots, 9, 153, 370, 371, 407, 1634, 8208$ et 9474.

71.1.2 II. Inversions de permutations

Sujet d'après CCINP 2019.

Pour tout $n \in \mathbb{N}^*$, une **permutation de taille** n est une bijection de l'ensemble $[\![0, n-1]\!] = \{0, 1, \ldots, n-1\}$ dans lui-même. Dans la suite, l'ensemble des permutations de taille n est noté $\mathfrak{S}_n$. Étant donné une permutation σ de taille n, on la représente sous la forme $\sigma_0\sigma_1\ldots\sigma_{n-1}$ où :

$$\forall i \in \{0, 1, \ldots, n-1\}, \sigma_i = \sigma(i)$$

Par exemple, $\sigma = 1032$ représente la permutation envoyant 0 sur 1, 1 sur 0, 2 sur 3 et 3 sur 2.

Soit $\sigma = \sigma_0\ldots\sigma_{n-1}$ une permutation de taille n. On dit que $(i, j) \in \{0, 1, \ldots, n-1\}^2$ est une **inversion** de σ si $\sigma_i > \sigma_j$ et $i < j$. On note $inv(\sigma)$ le nombre d'inversions de σ. Tout d'abord, on relie ce nombre avec un algorithme de tri : le tri à bulles. Puis, on s'intéresse à la table d'inversions d'une permutation, un objet qui caractérise une permutation.

Dans la suite, une permutation de taille n est représentée en Python par un tableau sans répétition contenant tous les entiers compris entre 0 et $n-1$. Par exemple, le tableau `[1, 0, 3, 2]` représente la permutation $\sigma = 1032$.

Si A est un ensemble fini, $Card(A)$ désigne le cardinal de l'ensemble A. Pour tout entier $n \geq 1$, on pose $E_n = \prod_{k=1}^{n} \{0, 1, \ldots, n-k\}$. Par exemple, on a $E_3 = \{0, 1, 2\} \times \{0, 1\} \times \{0\}$.

Tri et inversions

Question 9 Déterminer l'ensemble des inversions de la permutation $\sigma = 140253$.

Corrigé

On a les inversions : $(0,2),(1,2),(1,3),(1,5),(4,5)$ qui correspondent respectivement aux comparaisons $1>0, 4>0, 4>2, 4>3$ et $5>3$.

On rappelle l'algorithme du tri à bulles :

- **Entrée** : un tableau d'entiers L
- **Pour** i allant de (taille de L) -1 à 1 (bornes incluses) **faire**
  - **Pour** j allant de (taille de L) -1 à (taille de L) $-i$ (bornes incluses) **faire**
    - **Si** $L[j] < L[j-1]$ **alors**
      - Échanger $L[j]$ et $L[j-1]$

On rappelle également que `for i in reversed(range(a,b))` permet de parcourir de $b-1$ à a en descendant.

Question 10 Écrire une fonction `tri_bulle` qui prend en argument un tableau d'entiers L et qui trie ce tableau à l'aide du tri à bulles. À l'issue de la fonction, le tableau est trié.

Corrigé

```
def tri_bulle(L):
    for i in reversed(range(1, len(L))):
        for j in reversed(range(len(L)-i, len(L))):
            if L[j] < L[j-1]:
                L[j], L[j-1] = L[j-1], L[j]
```

Dans la suite, on admet que l'algorithme du tri à bulles est bien un algorithme de tri.

Question 11 Montrer que si on effectue exactement un échange dans un tableau lors du tri à bulles, le nouveau tableau a exactement une inversion en moins.

Corrigé

L'algorithme échange $L[j]$ et $L[j-1]$ lorsque $L[j] < L[j-1]$: cela supprime une inversion. Montrons qu'on n'en a pas ainsi créé d'autre. La position relative des $L[i]$ pour $i < j-1$ par rapport aux valeurs initialement en $j-1$ et j ne change pas, donc le nombre d'inversions non plus. De même pour les $L[i]$ avec $i > j$.

Question 12 En déduire une fonction Python `nombre_inversions` qui prend en argument un tableau `L` correspondant à une permutation et renvoyant le nombre d'inversions de celle-ci. Cette fonction sera une légère modification du tri à bulles.

Corrigé

```
def nombre_inversions(L):
    total = 0
    for i in reversed(range(1, len(L))):
        for j in reversed(range(len(L)-i, len(L))):
            if L[j] < L[j-1]:
                L[j], L[j-1] = L[j-1], L[j]
                total = total + 1
    return total
```

Table d'inversions d'une permutation

Définition 71.1 — (table d'inversions)

Soit σ une permutation de taille $n \geq 1$. La **table d'inversions** de σ est le n-uplet $(\alpha_0, \ldots, \alpha_{n-1})$ tel que :

$$\forall i \in \{0, \ldots, n-1\}, \alpha_i = Card\big(\{j \in \{i+1, \ldots, n-1\} \mid \sigma_j < \sigma_i\}\big)$$

Elle est notée Tab_σ. De plus, pour tout $i \in \{0, \ldots, n-1\}$, $Tab_\sigma[i]$ désigne α_i.

Pour tout $n \geq 1$, on désigne par $Tab : \mathfrak{S}_n \to E_n$ l'application qui associe à une permutation de taille n sa table d'inversions.

Question 13 Déterminer la table d'inversions de la permutation $\sigma = 140253$.

Corrigé

Il s'agit de $[1, 3, 0, 0, 1, 0]$.

Question 14 Montrer que pour toute permutation σ de taille $n \geq 1$, Tab_σ est bien un élément de E_n.

Corrigé

Soit σ une permutation de taille $n \geq 1$. Avec les notations de l'énoncé, pour tout $i \in [\![0, n-1]\!]$,

$$\alpha_i = Card\{j \in [\![i+1, n-1]\!] \mid \sigma_j < \sigma_i\} \leq Card\ [\![i+1, n-1]\!] = n - i - 1$$

donc $\alpha_i \in \{0, \ldots, n-i-1\}$, ce qui, avec le décalage d'indice, correspond au terme général du produit cartésien définissant E_n.

Question 15 Soit σ une permutation de taille $n \geq 1$. Montrer que $\sigma_0 = Tab_\sigma[0]$.

Corrigé

On a $\alpha_0 = Card\{j \in [\![1, n-1]\!] \mid \sigma_j < \sigma_0\}$. Mais puisque $\sigma_0 \not< \sigma_0$, c'est aussi $Card\{j \in [\![0, n-1]\!] \mid \sigma_j < \sigma_0\}$. Puisque σ est une permutation, on a donc $\alpha_0 = Card\{j \in [\![0, n-1]\!] \mid j < \sigma_0\}$, ce qui vaut σ_0.

Question 16 Montrer que pour tout entier $n \geq 1$, l'application Tab est bijective.

Corrigé

Compte tenu de l'égalité des cardinaux ($Card(\mathfrak{S}_n) = n! = Card(E_n)$), il suffit de montrer l'injectivité.

Soit $n \geq 1$ et soit σ et σ' deux permutations de taille n qui ont la même table α. On va montrer que $\sigma^{-1} = \sigma'^{-1}$.

En effet, l'indice du zéro est nécessairement $\min\{k \in [\![0, n-1]\!] \mid \alpha_k = 0\}$: c'est forcément un tel k (ce qui assure qu'il existe de tels k), et si $\sigma_k = 0$ et $k' < k$ alors $\alpha_{k'} > 0$ à cause du zéro situé à droite de k'.

Notons donc $k_0 = \sigma^{-1}(0) = \sigma'^{-1}(0)$. S'il existe $k \in [\![0, k_0 - 1]\!]$ tel que $\alpha_k = 1$ alors la plus petite de ces positions est $\sigma^{-1}(1) = \sigma'^{-1}(1)$; sinon cette position est $\min\{k \in [\![k_0 + 1, n-1]\!] \mid \alpha_k = 0\}$.

On note $k_1 = \sigma^{-1}(1) = \sigma'^{-1}(1)$. En traitant comme ci-dessus les différentes possibilités, on montre de même que $\sigma^{-1}(2) = \sigma'^{-1}(2)$. De proche en proche on détermine $\sigma^{-1} = \sigma'^{-1}$ puis $\sigma = \sigma'$.

Question 17 Écrire une fonction Python `permutation_vers_table` qui prend en argument une permutation représentée par le tableau `L` et qui renvoie la table d'inversions correspondante.

Corrigé

```
def permutation_vers_table(L):
    n = len(L)
    res = [0] * n
    for i in range(0, n):
```

```
        compte = 0
        for j in range(i+1, n):
            if L[j] < L[i]:
                compte = compte + 1
        res[i] = compte
    return res
```

Question 18 Écrire une fonction Python `table_vers_permutation` qui prend en argument un tableau `L` qui correspond à une table d'inversions et qui renvoie la permutation qui lui est associée.

Corrigé

```
def table_vers_permutation(L):
    n = len(L)
    sigma = [0] * n
    deja_mis = [False] * n
    for i in range(0, n):
        k = 0
        compte = 0
        while deja_mis[k] or compte < L[i]:
            if not deja_mis[k]:
                compte = compte + 1
            k = k + 1
        sigma[i] = k
        deja_mis[k] = True
    return sigma
```

DEVOIR 72

DS 1 - Poteaux télégraphiques

72.1 DS 1 — Poteaux télégraphiques

Informatique tronc commun, année scolaire 2021-2022.

Ce devoir a pour objectif de choisir où placer des poteaux télégraphiques pour relier le point le plus à gauche d'un paysage unidimensionnel au point le plus à droite en fonction de critères de coût. Nous ferons les simplifications suivantes : les fils sont sans poids et tendus ; ils relient donc en ligne droite les sommets de deux poteaux consécutifs. Les normes de sécurité imposent que les fils soient en tout point à une distance d'au moins Δ (mesurée verticalement) au-dessus du sol. Les poteaux sont tous de longueur identique $\ell \geq \Delta$. Voici par exemple une proposition valide de placement de poteaux pour le paysage ci-dessous avec $\Delta = 0.5$ et $\ell = 2.0$.

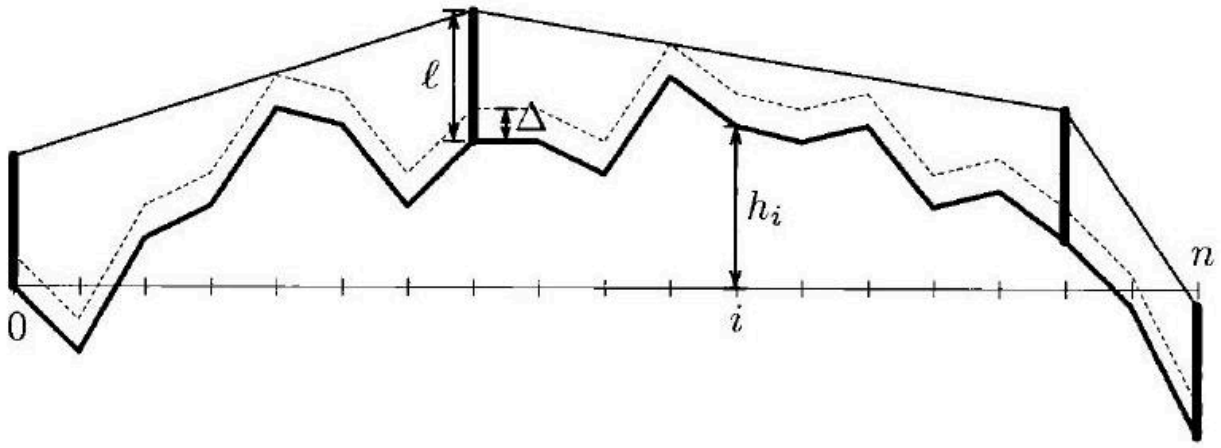


Figure 72.1 — Un exemple de poteaux

Figure 1 - Un exemple de poteaux où le fil est en tout point au moins Δ au-dessus du sol.

72.1.1 I. Planter le paysage

Nous définissons le paysage à partir d'une suite de relevés de dénivelés stockés dans un tableau `deniveles` de nombres flottants (c'est-à-dire des nombres à virgule) de taille $n + 1$.

Le paysage est alors décrit par une ligne brisée de $n + 1$ points, dont le i-ème point P_i est de coordonnées (i, h_i) où :

$$h_0 = 0.0 \quad \text{et} \quad h_i = h_{i-1} + \text{deniveles}\,[i] \quad \text{pour} \quad i \in \{1, \ldots, n\}.$$

Question 1 Écrire une fonction `calculHauteurs(deniveles, n)` qui renvoie un tableau `hauteurs` de taille $n + 1$ tel que `hauteurs[i]` soit la hauteur du i-ème point.

Question 2 Écrire une fonction `calculFenetre(n)` qui calcule les hauteurs minimale et maximale d'un point du paysage et les stocke dans deux variables globales `hMin` et `hMax`. Cette procédure stockera également dans deux variables globales `iMin` et `iMax` les indices des points les plus à gauche de hauteur minimale et maximale respectivement.

On appelle distance au sol de i à j, la longueur de la ligne brisée allant du point P_i au point P_j.

Question 3 Écrire une fonction `distanceAuSol(i, j)` qui calcule et renvoie la distance au sol de P_i à P_j.

On appelle un pic un point P_i d'indice $1 \leq i < n$ tel que $h_i > \max(h_{i-1}, h_{i+1})$. On appelle point remarquable, les pics et les points des bords gauche (point P_0) et droit (point P_n). On appelle bassin toute partie du paysage allant d'un point remarquable au suivant.

Question 4 Écrire une fonction `longueurDuPlusLongBassin(n)` qui calcule et renvoie la longueur au sol maximale d'un bassin dans le paysage.

72.1.2 II. Planter les poteaux

On souhaite relier le point le plus à gauche au point le plus à droite par un fil télégraphique. Pour cela, nous devons choisir à quels points parmi les $(P_i)_{0\leq i\leq n}$ planter les poteaux télégraphiques intermédiaires. Rappelons que le fil est supposé sans poids et tendu et qu'il relie donc les sommets de chacun des poteaux en ligne droite. Les poteaux sont plantés verticalement et ont tous une longueur identique $\ell \geq \Delta$ (ℓ et Δ sont des nombres flottants).

La législation impose que le fil doit rester à une distance supérieure ou égale à Δ (mesurée verticalement) au-dessus du sol. Pour tester si un fil tiré entre un poteau placé au point P_i et un poteau placé au point P_j (avec $j > i$ ou $j < i$) respecte la législation, notons

$$\alpha_{i,k} = \frac{(h_k + \Delta) - (h_i + \ell)}{k - i} \text{ pour } k \neq i \text{, et } \beta_{i,j} = \frac{(h_j + \ell) - (h_i + \ell)}{j - i},$$

les pentes des fils tirés depuis le poteau en P_i jusqu'à une hauteur Δ au-dessus du point P_k d'une part et jusqu'à une hauteur ℓ au-dessus du point P_j d'autre part (voir la Figure 2).

On admet que le fil tiré d'un poteau en P_i à un poteau en P_j respecte la législation si et seulement si : $\beta_{i,j} \geq \max_{i<k<j} \alpha_{i,k}$, lorsque $j > i$; et $\beta_{i,j} \leq \min_{j<k<i} \alpha_{i,k}$, lorsque $j < i$.

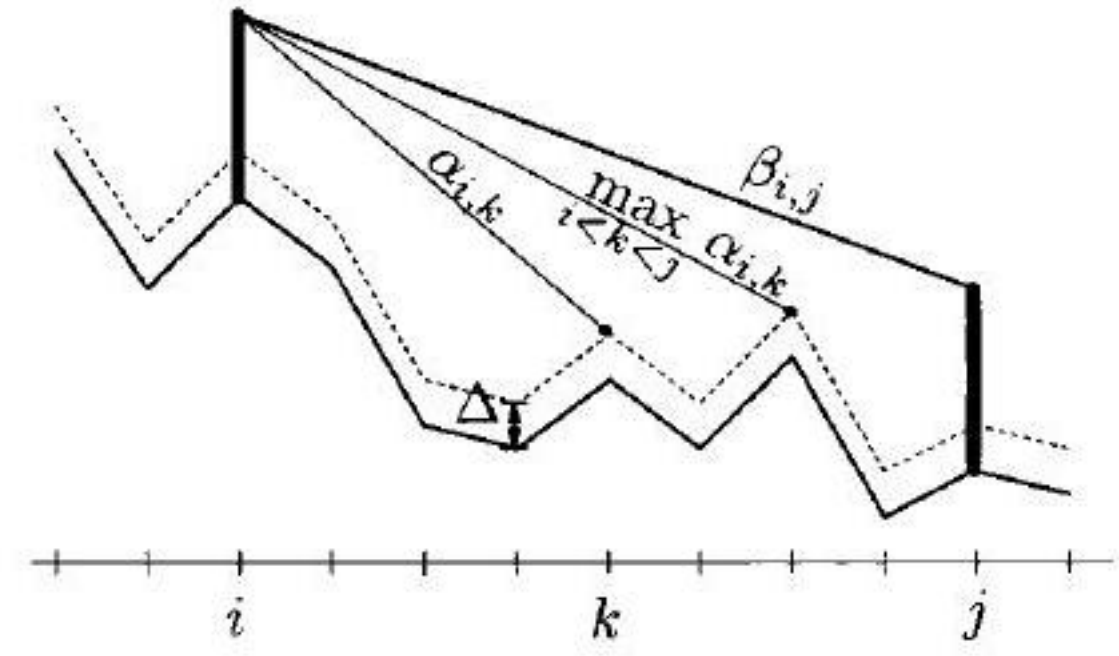


Figure 72.2 — Condition sur les pentes

Figure 2 - Condition sur les pentes pour pouvoir tirer un fil.

Question 5 En utilisant cette méthode, écrire une fonction `estDeltaAuDessusDuSol`(i, j, ℓ, Δ) qui renvoie `true` si un fil tiré entre les sommets d'un poteau placé au point P_i et d'un poteau placé au point P_j respecte la législation, et renvoie `false` dans le cas contraire.

Considérons une première stratégie, dite algorithme glouton en avant. Le premier poteau est planté en P_0. Pour calculer l'emplacement du prochain poteau, on part du dernier poteau planté et on avance (à droite) avec le fil tendu tant que la législation est respectée (et que P_n n'est pas atteint). Un nouveau poteau est alors planté, et on recommence jusqu'à ce que P_n soit atteint.

La figure 3 illustre la solution produite par cet algorithme.

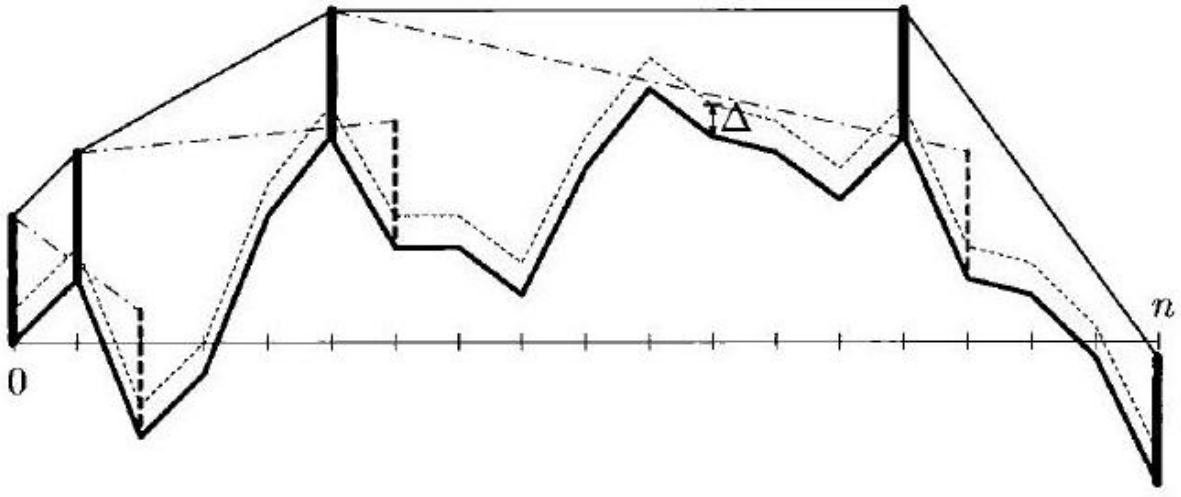


Figure 72.3 — Glouton en avant

Figure 3 - Solution produite par l'algorithme glouton en avant. Les poteaux et les fils en pointillés sont ceux violant la législation.

Question 6 Écrire une procédure `placementGloutonEnAvant`(n, ℓ, Δ) qui calcule la disposition des poteaux selon l'algorithme ci-dessus. La solution sera stockée dans un tableau global `poteaux` de la façon suivante :

- la case `poteaux[0]` contiendra le nombre de poteaux utilisés ;
- la case `poteaux[t]`, pour $t \geq 1$, contiendra l'indice i indiquant que le t-ème poteau, s'il existe, est planté en P_i.

Question 7 Donner une majoration de la complexité du temps de calcul de votre algorithme en fonction de n. Justifier qu'on peut se contenter du calcul de $O(n)$ pentes pour implémenter l'algorithme glouton en avant. Expliquer succinctement comment modifier votre procédure (si nécessaire) pour obtenir une complexité en $O(n)$. Aucune implémentation n'est demandée dans cette question.

L'algorithme glouton en avant a tendance à placer beaucoup trop de poteaux, en particulier dans les vallées alors qu'il suffirait de relier les deux extrémités par un unique fil. Nous considérons donc une alternative, dite glouton au plus loin, qui consiste à planter le prochain poteau le plus à droite possible de la position courante. Le premier poteau est toujours planté en P_0.

La figure 4 illustre la solution produite par cet algorithme sur le même paysage que celui de la figure 3.

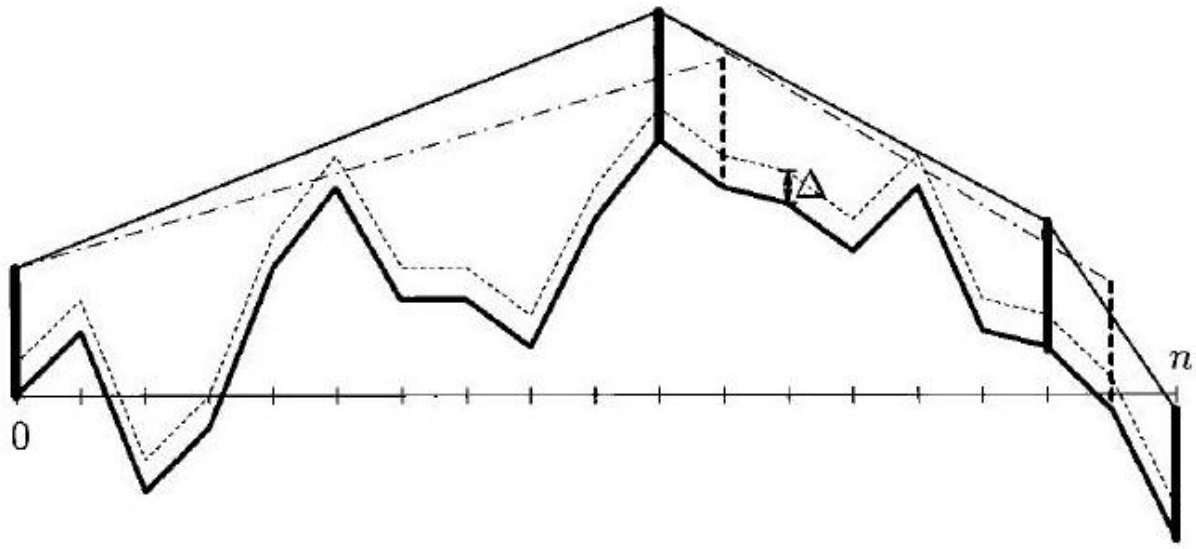


Figure 72.4 — Glouton au plus loin

Figure 4 - Solution produite par l'algorithme glouton au plus loin. Les poteaux et fils en pointillés sont ceux violant la législation.

Question 8 Écrire une procédure `placementGloutonAuPlusLoin`(n, ℓ, Δ) qui calcule la disposition des poteaux selon l'algorithme ci-dessus. La solution sera stockée dans un tableau global `poteaux` de la façon suivante :

- la case `poteaux[0]` contiendra le nombre de poteaux utilisés ;
- la case `poteaux[t]`, pour $t \geq 1$, contiendra l'indice i indiquant que le t-ème poteau, s'il existe, est planté en P_i.

72.1.3 III. Minimiser la longueur du fil

L'objectif est maintenant de calculer un placement optimal des poteaux en terme de longueur totale du fil (le nombre de poteaux pouvant être maintenant arbitraire). Un tableau `optL` peut être calculé de proche en proche tel que `optL[i]` désigne la longueur minimale de fil à utiliser si l'on souhaitait relier le point P_0 au point P_i.

Question 9 Soit $L_{i,j}$ la longueur du segment d'extrémités P_i et P_j.

Montrer que le tableau `optL` vérifie : optL $[0] = 0$, et pour $1 \leq i \leq n$,

$$\text{optL}\,[i] = \min\{L_{i,j} + \text{optL}\,[j],\ \text{où}\ \ 0 \leq j < i\ \ \text{et il est possible de tirer un fil de}\ \ P_j\ \ \text{à}\ \ P_i\}.$$

La case optL $[n]$ contient alors la longueur minimale de fil pour relier le bord gauche au bord droit.

Question 10 Écrire une fonction `longueurMinimale`(n, ℓ, Δ) qui renvoie la longueur minimale de fil pour relier le bord gauche au bord droit, en respectant la législation.

Question 11 Modifier votre procédure `longueurMinimale`(n, ℓ, Δ) pour qu'elle remplisse également un tableau `precOptL` de taille $n+1$, où `precOptL[i]` contient l'indice du poteau précédant le poteau placé en P_i dans une solution de longueur minimale reliant P_0 à P_i. (Par convention, `precOptL[0] = -1`.)

Question 12 Écrire une procédure qui produit un placement des poteaux minimisant la longueur du fil de P_0 à P_n à partir du tableau `precOptL`. La solution sera stockée dans un tableau global `poteaux` de la façon suivante :

- la case `poteaux[0]` contiendra le nombre de poteaux utilisés ;
- la case `poteaux[t]`, pour $t \geq 1$, contiendra l'indice i indiquant que le t-ème poteau, s'il existe, est planté en P_i.

Quelle est la complexité de votre procédure en fonction de n ?

DEVOIR 73

Interro 1 - Programmation Python

73.1 Interro 1 — Programmation Python

MPSI tronc commun, année scolaire 2021-2022 — deux sujets (A et B).

Corrigés

La source ne comporte pas de corrigé. Les corrigés ci-dessous sont des propositions (testées), à relire.

73.1.1 Sujet A

Exercice 1. Écrire une fonction `somme` qui prend en paramètre un entier n et calcule $\sum_{i=1}^{n^2} \prod_{j=1}^{i} (i^3 + (j-i)^2)$.

Corrigé

```
def somme(n):
    s = 0
    for i in range(1, n**2 + 1):
        p = 1
        for j in range(1, i + 1):
            p = p * (i**3 + (j - i)**2)
        s = s + p
    return s
```

Exercice 2. Écrire une fonction `maximum` qui prend en entrée un tableau t, deux indices i et j, on supposera qu'elle sera toujours appelée avec $0 \leq i \leq j < len(t)$, et renvoie l'indice de la plus grande valeur de t comprise entre les indices i et j.

Dans le cas où plusieurs indices conviennent on renverra le **plus petit**.

Corrigé

L'inégalité stricte garantit qu'on conserve le premier indice rencontré.

```
def maximum(t, i, j):
    m = i
    for k in range(i + 1, j + 1):
        if t[k] > t[m]:
            m = k
    return m
```

Exercice 3. Écrire une fonction `diviseur_carre` qui prend en entrée un entier n et renvoie un booléen indiquant si n a un diviseur de la forme p^2.

Corrigé

On exclut le cas trivial $p = 1$ et on teste les $p \geq 2$ tels que $p^2 \leq n$.

```python
def diviseur_carre(n):
    p = 2
    while p * p <= n:
        if n % (p * p) == 0:
            return True
        p = p + 1
    return False
```

Exercice 4. Écrire une fonction `decale` qui prend en entrée le caractère c d'une lettre de l'alphabet en minuscule et un entier d et renvoie la lettre déduite de c par un décalage de d lettres modulo la longueur de l'alphabet : $y \to z \to a$ est décalé de 2.

Exemple : `decale('c',3)` renvoie `'f'` et `decale('y',2)` renvoie `'a'`.

Rappel : les caractères sont tous associés à des numéros dans une table, `ord` renvoie le numéro associé à un caractère et `chr` le caractère associé à un numéro.

Corrigé

```python
def decale(c, d):
    return chr(ord('a') + (ord(c) - ord('a') + d) % 26)
```

73.1.2 Sujet B

Exercice 1. Écrire une fonction `somme` qui prend en paramètre un entier n et calcule $\sum_{i=-n}^{n} \prod_{j=i}^{2i} j(i-j)^3$.

Corrigé

Pour $i < 0$, on a $2i < i$: le produit est vide et vaut 1, ce que donne naturellement la boucle sur `range(i, 2*i + 1)`.

```python
def somme(n):
    s = 0
    for i in range(-n, n + 1):
        p = 1
        for j in range(i, 2 * i + 1):
            p = p * j * (i - j)**3
        s = s + p
    return s
```

Exercice 2. Écrire une fonction `minimum` qui prend en entrée un tableau t, deux indices i et j, on supposera qu'elle sera toujours appelée avec $0 \leq i \leq j < len(t)$, et renvoie l'indice de la plus petite valeur de t comprise entre les indices i et j.

Dans le cas où plusieurs indices conviennent on renverra le **plus grand**.

Corrigé

L'inégalité large garantit qu'on conserve le dernier indice rencontré.

```python
def minimum(t, i, j):
    m = i
    for k in range(i + 1, j + 1):
        if t[k] <= t[m]:
            m = k
    return m
```

Exercice 3. Écrire une fonction `diviseur_cube` qui prend en entrée un entier n et renvoie un booléen indiquant si n a un diviseur de la forme p^3.

Corrigé

```python
def diviseur_cube(n):
    p = 2
    while p * p * p <= n:
        if n % (p * p * p) == 0:
            return True
        p = p + 1
    return False
```

Exercice 4. Écrire une fonction `decale` qui prend en entrée le caractère c d'une lettre de l'alphabet en minuscule et un entier d et renvoie la lettre déduite de c par un décalage de d lettres modulo la longueur de l'alphabet : $y \to z \to a$ est décalé de 2.

Exemple : `decale('c',3)` renvoie `'f'` et `decale('y',2)` renvoie `'a'`.

Rappel : les caractères sont tous associés à des numéros dans une table, `ord` renvoie le numéro associé à un caractère et `chr` le caractère associé à un numéro.

✍ **Corrigé**

```python
def decale(c, d):
    return chr(ord('a') + (ord(c) - ord('a') + d) % 26)
```

DEVOIR 74

DS 2 - Correction (circulation automobile)

74.1 DS 2 — Correction : modélisation de la circulation

MP, année scolaire 2021-2022 — corrigé (informatique commune, Python et SQL).

✍ **Sujet**

Seule la correction est disponible : l'énoncé n'est pas reproduit ici. Sujet d'après le concours Mines-Ponts 2016, informatique commune (modélisation de la circulation automobile : files de voitures, croisements, espace des configurations, base de données de croisements et de voies).

74.1.1 Partie I

Question 1 ::: {.adm kind=« corrige » title=« Corrigé »}

On peut faire une liste `[v0,...,vn-1]` où `vi` vaut `True` quand une voiture est présente à l'indice i et `False` sinon. :::

Question 2 ::: {.adm kind=« corrige » title=« Corrigé »}

```
A = [ True, False, True, True, False,
    False, False, False, False, False, True ]
```

:::

Question 3 ::: {.adm kind=« corrige » title=« Corrigé »}

```
def occupe(L, i):
    return L[i]
```

:::

Question 4 ::: {.adm kind=« corrige » title=« Corrigé »}

Chacune des n cases peut prendre deux valeurs, il y en a donc 2^n. :::

Question 5 ::: {.adm kind=« corrige » title=« Corrigé »}

Question mal posée, on peut répondre légitimement avec `==`...

```
def egal(L1, L2):
    if len(L1) != len(L2):
        return False
    for i in range(len(L1)):
        if L1[i] != L2[i]:
            return False
    return True
```

:::

Question 6 ::: {.adm kind=« corrige » title=« Corrigé »}

Complexité linéaire en temps en pire cas, ce qui correspond au fait d'avoir des listes égales. :::

Question 7 ::: {.adm kind=« corrige » title=« Corrigé »}

Le type de retour est un booléen donc de type `bool`. *Note : cette question est étrange...* :::

74.1.2 Partie II

Question 8 ::: {.adm kind=« corrige » title=« Corrigé »}

On a vu que `avancer(A,False)` renvoie

```
[False,True,False,True, True, False, False, False, False, False, False]
```

si on appelle sur ce résultat `avancer` avec `True` on obtient :

```
[True, False,True,False,True, True, False, False, False, False, False]
```

:::

Question 9 ::: {.adm kind=« corrige » title=« Corrigé »}

On a plusieurs choix ici, si on considère que la fonction `avancer` est définie, on peut l'utiliser directement. Sinon, on peut le faire à la main.

```
def avancer_fin(L, m):
    La = [ False ] * len(L)
    for i in range(m):
        La[i] = L[i]
    for i in range(m+1, len(L)):
        La[i] = L[i-1]
    return La
```

:::

Question 10 ::: {.adm kind=« corrige » title=« Corrigé »}

```
def avancer_debut(L, b, m):
    La = [ False ] * len(L)
    for i in range(m+1, len(L)):
        La[i] = L[i]
    for i in range(1, m+1):
        La[i] = L[i-1]
    La[0] = b
    return La
```

:::

Question 11 ::: {.adm kind=« corrige » title=« Corrigé »}

Ici, il faut procéder en reculant à partir de la position `m`.

```
def avancer_debut_bloque(L, b, m):
    # on cherche la première case libre à gauche de m (strictement)
    i = m-1
    while i > 0 and L[i]:
        i = i-1
    return avancer_debut(L, b, i)
```

:::

74.1.3 Partie III

Question 12 ::: {.adm kind=« corrige » title=« Corrigé »}

```
def avancer_files(L1, b1, L2, b2):
    p = len(L1)//2
    La1 = avancer_file(L1, b1)
    if La1[p]:
        La2 = avancer_fin(avancer_debut_bloque(L2, b2, p), p)
    else:
        La2 = avancer_file(L2, b2)
    return [La1, La2]
```

:::

Question 13 ::: {.adm kind=« corrige » title=« Corrigé »}

L'appel renvoie

```
[ [ False, False, True, False, True ], [ False, True, False, True, False ] ]
```

:::

74.1.4 Partie IV

Question 14 ::: {.adm kind=« corrige » title=« Corrigé »}

Il suffit que toutes les cases de `L1` soient occupées. :::

Question 15 ::: {.adm kind=« corrige » title=« Corrigé »}

On va y passer en deux temps, d'abord toutes les voitures de `L1` passent à droite, ensuite les voitures de `L2` peuvent descendre et au passage on rajoute des voitures à gauche de `L1`.

On a donc 5 étapes pour libérer le croisement puis comme sur la dernière étape on a commencé à faire descendre les voitures de `L2` on a juste besoin de 4 étapes ensuite, donc 9 étapes en tout. :::

Question 16 ::: {.adm kind=« corrige » title=« Corrigé »}

La configuration 4c n'a pas d'antécédent car il faudrait avoir une voiture de `L1` et une voiture de `L2` sur la position centrale. :::

74.1.5 Partie V

Question 17 ::: {.adm kind=« corrige » title=« Corrigé »}

```
def elim_double(L):
    Le = []
    for x in L:
        if Le == [] or x != Le[-1]:
            Le.append(x)
    return Le
```

:::

Question 18 ::: {.adm kind=« corrige » title=« Corrigé »}

```
[ 1, 2, 3, 5 ]
```

:::

Question 19 ::: {.adm kind=« corrige » title=« Corrigé »}

Non car elle se contente de supprimer les doublons côte à côte. :::

Question 20 ::: {.adm kind=« corrige » title=« Corrigé »}

- `recherche` renvoie un `bool` ;
- `successeur` renvoie une `list` de couples de configurations `[L1,L2]` ;
- `espace` a le même type que le retour de `successeur` ;
- `but` est une configuration, donc une liste `[L1,L2]` où les `Li` sont des `list` de `bool` de même longueur impaire. :::

Question 21 ::: {.adm kind=« corrige » title=« Corrigé »}

La première recherche correspond à une recherche linéaire et la seconde à une recherche dichotomique, donc en temps logarithmique. La seconde est préférable.

Cependant, remarquons que dans la mesure où on effectue un tri à chaque étape en $O(n \log_2 n)$, la question est peu pertinente... :::

Question 22 ::: {.adm kind=« corrige » title=« Corrigé »}

```
def versEntier(L):
    p = 1
    s = 0
    for b in reversed(L):
        if b:
            s = s + p
        p = p * 2
    return s
```

:::

Question 23 ::: {.adm kind=« corrige » title=« Corrigé »}

Il faut une taille correspondant au nombre de chiffres de n en binaire, soit $\lfloor \log_2 n \rfloor + 1$.

La condition est $n > 0$. :::

Question 24 ::: {.adm kind=« corrige » title=« Corrigé »}

La fonction termine car elle continue tant qu'elle rajoute des configurations et que le nombre de configurations est borné par 2^{2n-1}. :::

Question 25 ::: {.claude kind=« modifie » raison=« coquille « retrurn » corrigée en « return ». »}

Corrigé

On rajoute à chaque fois l'ensemble des successeurs aux précédents. Donc le test de présence du but ne va en fait porter que sur ces nouveaux successeurs. Il suffit de compter le nombre d'itérations dans la boucle `while`.

```
def recherche(but, init):
    espace = [ init ]
    stop = egal(but, init)
    if stop:
        return 0
    nb_etapes = 0
    while not stop:
        ancien = espace
        espace = espace + successeur(espace)
        espace.sort()
        espace = elim_double(espace)
        stop = egal(ancien, espace)
        nb_etapes += 1
        if but in espace:
            return nb_etapes
    return -1
```

:::

74.1.6 Partie VI

Question 26 ::: {.adm kind=« corrige » title=« Corrigé »}

```
SELECT id_croisement_fin FROM voie
WHERE id_croisement_debut = c;
```

:::

Question 27 ::: {.adm kind=« corrige » title=« Corrigé »}

```
SELECT longitude, latitude
FROM croisement JOIN voie ON croisement.id = id_croisement_fin
WHERE id_croisement_debut = c;
```

:::

Question 28 ::: {.adm kind=« corrige » title=« Corrigé »}

Cette requête renvoie les croisements accessibles en suivant deux voies à partir du croisement *c*. :::

DEVOIR 75

Interro de rentrée - Révisions Python

75.1 Interro de rentrée — Révisions Python

MP tronc commun, année scolaire 2022-2023 — six questions à traiter sur l'énoncé en 30 min.

> **Objectif** Faire un état des lieux et préparer le premier TP. La correction est faite ensemble ensuite : les corrigés et les rappels ci-dessous sont ceux de la séance de révision rapide de Python.

Question 1 Écrire les instructions permettant de placer dans la variable `P` la valeur de $\prod_{k=1}^{n}(1+k^2)$ en considérant que la variable `n` est définie.

Corrigé

```
P = 1
for k in range(1, n + 1):
    P = P * (1 + k * k)
```

Rappels.

- On déclare et on modifie une variable avec une instruction `nom = expression`.
- `range(a, b + 1)` énumère les entiers de `a` à `b`.
- `for k in iterable: corps` va appliquer le corps de boucle à chaque valeur de l'itérable en l'affectant à `k`.
- Si on déroule, c'est équivalent à :

```
P = P * (1 + 1*1)
P = P * (1 + 2*2)
P = P * (1 + 3*3)
...
P = P * (1 + n*n)
```

Question 2 En déduire une fonction `produit` qui prend en argument `n` et renvoie cette valeur.

Corrigé

```
def produit(n):
    P = 1
    for k in range(1, n + 1):
        P = P * (1 + k * k)
    return P
```

Rappels.

- Une fonction est une abstraction sur un ensemble de lignes, en identifiant :
  - des valeurs en entrée, les arguments ou paramètres ;

- **une** valeur en sortie, la valeur de retour.
- `def mafonction(arg1, ..., argn)`: pour donner son nom `mafonction` ; `arg1, ..., argn` sont des noms de variables qu'on va utiliser dans le corps.
- `return v` pour renvoyer la valeur `v`. Cela interrompt l'exécution de la fonction.
- Quand on vous dit que la fonction prend un entier en argument, il n'y a rien à vérifier.
- On **appelle** la fonction en passant des valeurs (des expressions) à la fonction : `mafonction(v1, ..., vn)`. Tout se passe comme si le corps de la fonction était exécuté après avoir défini `arg1 = v1, ..., argn = vn`.

Question 3 On considère le programme suivant pour créer une matrice carrée de zéros :

```
def zeros(n):
    l = [ 0 ] * n
    m = [ l ] * n
    return m
m = zeros(2)
m[0][0] = 1
print(m)
```

Qu'est-ce qui est affiché ? Expliquez.

Corrigé

```
[[1, 0], [1, 0]]
```

- Quand on écrit `m = [ l ] * n`, on ne copie pas n fois le tableau `l`.
- Python se contente de placer dans `m` n références vers le **même** tableau `l` : modifier `m[0][0]` modifie donc aussi `m[1][0]`.
- Pour corriger, le plus simple est le code suivant. Chaque `l` est un nouveau tableau créé à chaque itération.

```
def zeros(n):
    m = []
    for i in range(n):
        l = [ 0 ] * n
        m.append( l )
    return m
```

Question 4 On considère une matrice **carrée** représentée par un tableau à deux dimensions. Écrire une fonction `compte` qui prend en arguments une telle matrice `m` et une valeur `x` et renvoie le nombre de coefficients égaux à `x` dans `m`.

Ainsi `compte([[1,2],[1,3]], 1)` renverra `2` car il y a deux `1`.

Corrigé

```
def compte(m, x):
    nb = 0
    for ligne in m:
        for coeff in ligne:
            if coeff == x:
                nb = nb + 1
    return nb
```

Rappels.

- Tableau 1D : une boucle `for`.
- Tableau 2D : deux boucles `for` imbriquées :

- imbrication = produit cartésien ;
- à la suite = union.

- Matrice carrée donc `len(m) == len(m[0])`.

Question 5 Écrire une fonction `occurrences` qui prend en entrée une matrice **carrée** et renvoie un dictionnaire dont les clés sont les coefficients de la matrice et les valeurs associées sont leur nombre d'occurrences.

Exemple : `occurrences([[1,2],[1,3]])` renvoie le dictionnaire `{ 1 : 2, 2 : 1, 3 : 1 }`.

Corrigé

```
def occurrences(m):
    occ = {}
    for ligne in m:
        for coeff in ligne:
            if coeff not in occ:
                occ[coeff] = 1
            else:
                occ[coeff] = occ[coeff] + 1
    return occ
```

Rappels.

- Un dictionnaire est une sorte de tableau dont les indices ne sont pas 0, 1, … mais ce qu'on veut (ou presque).
- `{}` est le dictionnaire vide.
- `d[cle] = valeur` rajoute ou modifie une association.
- `d[cle]` permet d'accéder à la valeur.
- Ici `occ[coeff]` n'existe pas forcément : on teste avec `coeff in occ` si c'est le cas.

Question 6 À l'aide de la fonction `zeros` précédente en considérant qu'elle a été corrigée, écrire une fonction `complemente` qui prend en entrée une matrice carrée `m` dont les coefficients sont `0` ou `1` et renvoie une **nouvelle matrice** telle que son coefficient `(i,j)` soit égal à 1 si et seulement si celui de `m` est égal à `0` et réciproquement.

Exemple : `complemente([[1,0],[1,1]])` renvoie `[[0,1],[0,0]]`.

Corrigé

```
def complemente(m):
    n = len(m)
    mc = zeros(n)
    for i in range(n):
        for j in range(n):
            mc[i][j] = 1 - m[i][j]
    return mc
```

Rappels.

- Avec les tableaux, ce type de questions où on demande de créer un **nouveau** tableau est assez classique.
- Le plus simple :
  - créer un tableau trivial (des zéros ici) ;
  - le remplir.

PARTIE XII

Annexes

76 Complexité et chansons algorithmiques 973

CHAPITRE 76

Complexité et chansons algorithmiques

DANS CE CHAPITRE

76.1 99 DM à rendre pour lundi...
76.2 Alouette
76.3 N days of Christmas
76.4 The TELNET song
76.5 Chanter des entiers

Dans ce problème, on va s'intéresser à des chansons dont les paroles peuvent être générées par des algorithmes, et on va analyser le temps qu'il faut pour les chanter. Les notions utilisées sont présentées dans le chapitre Introduction à l'analyse des algorithmes.

76.1 99 DM à rendre pour lundi...

On considère la chanson suivante :

Algorithme 76.1 — : DMàRendre

Entrée : $n \in \mathbb{N}$

Paroles :

- Pour i allant de n à 1
  - **Chanter** « i DM à rendre pour lundi »
  - **Chanter** « prends-en un, fais-le, i-1 DM à rendre pour lundi »
- **Chanter** « plus de DM à rendre pour lundi »
- **Chanter** « je retourne en cours, n DM à rendre pour lundi »

On considère ici que chaque instruction **Chanter** a une complexité linéaire dans le **nombre de mots** chantés.

On supposera dans tout ce sujet, sauf contre-indication, qu'un entier est un unique mot.

Question 1

Quelle est la complexité de cette chanson ?

Corrigé

On peut noter a le nombre de mots chantés dans une itération de la boucle et b le nombre de mots de la conclusion, qui ne dépendent pas de n.

L'appel `DMàRendre(n)` va donc chanter $b + na = O(n)$ mots, et même $\Theta(n)$ mots. Sa complexité est linéaire.

76.2 Alouette

Algorithme 76.2 — : Alouette

Entrées :

- $n \in \mathbb{N}$;
- un tableau `t` de n chaînes de caractères comportant chacune **un mot**.

Paroles :

- **Chanter** « Alouette, gentille alouette, alouette, je te plumerai. »
- Pour i allant de 0 à n-1
  - **Chanter** « Je te plumerai `t[i]`. Je te plumerai `t[i]`. »
  - Pour j allant de i-1 à 0
    - **Chanter** « Et `t[j]` ! Et `t[j]` ! »
  - **Chanter** « Alouette ! Alouette ! Aaaaaaaa.... »
  - **Chanter** « ...alouette, gentille alouette, alouette, je te plumerai. »

Question 2

Quelle est la complexité de cette chanson en fonction de n ?

Corrigé

On note :

- a le nombre de mots de la première ligne ;
- b le nombre de mots chantés dans une itération de la boucle principale, sans compter ceux de la boucle intérieure.

Pour une valeur de i, la boucle intérieure effectue i itérations de 4 mots chacune. On a alors $a + nb + 4(0 + 1 + \ldots + (n-1)) = a + nb + 4\frac{n(n-1)}{2} = a + nb + 2n(n-1) = \Theta(n^2)$ mots chantés : la complexité est quadratique.

76.3 N days of Christmas

Algorithme 76.3 — : NDaysOfChristmas

Entrées :

- $n \in \mathbb{N}^*$;
- un tableau `gifts` de $n-1$ chaînes de caractères comportant chacune **un mot**.

Paroles :

- Pour i allant de 1 à n
  - **Chanter** « On the ith day of Christmas, my true love gave to me »
  - Pour j allant de i à 2
    - **Chanter** « j `gifts[j-2]` »
  - Si i > 1
    - **Chanter** « and »
  - **Chanter** « a partridge in a pear tree. »

Question 3

Quelle est la complexité de cette chanson en fonction de n ?

Corrigé

Si on compte le nombre de mots chantés qui ne sont pas « j » ou un nom de cadeau, on a $12 + 6 = 18$ mots par itération, plus le « and » des $n - 1$ dernières itérations, soit $18n + n - 1 = 19n - 1 = O(n)$ mots.

Le cadeau `gifts[k]`, pour $k \in [\![0, n-2]\!]$, est mentionné lors des itérations pour i allant de $k + 2$ à n, donc $n - k - 1$ fois.

Ainsi, en tout, les cadeaux sont mentionnés $\sum_{k=0}^{n-2}(n-k-1) = \sum_{q=1}^{n-1} q = \frac{n(n-1)}{2}$ fois.

On remarque que chaque mention d'un cadeau est précédée du nombre `j` : on prononce donc autant de nombres que de cadeaux dans la boucle intérieure. Le nombre de mots chantés est donc $19n - 1 + 2 \cdot \frac{n(n-1)}{2} = n^2 + 18n - 1 = \Theta(n^2)$.

76.4 The TELNET song

Remarque 76.4 Petit historique sur `telnet` : il s'agit d'un programme, ancêtre de `ssh`, permettant de se connecter sur d'autres machines en ligne de commande.

Tout ce qu'on écrit est transmis à la machine distante comme des appuis au clavier, sauf si on l'échappe avec un caractère spécial, noté ici `^]` (dans la chanson, il est désigné par *control-uparrow*).

C'est notamment le cas de la commande permettant de quitter une session : `^]Q`.

Imaginons que l'on soit connecté à une machine `A` en `telnet` et que sur cette machine, on se connecte à une machine `B` en `telnet`. Si on tape `^]Q`, c'est interprété localement et cela ferme la connexion à `A`. Comment envoyer des commandes au `telnet` qui tourne sur `A` et qui nous relie à `B` ? Pour cela, on peut doubler l'échappement : `^]^]` est transmis comme un `^]` littéral. Ainsi, quand on tape `^]^]Q`, la machine `A` reçoit `^]Q` et ferme la connexion à `B`, et on peut poursuivre par `^]Q` pour quitter la connexion à `A`.

Et si on s'est connecté sur `C` depuis `B` ? Il faut doubler le doublement : `^]^]^]^]Q^]^]Q^]Q` !

Et ainsi de suite, ce qui a donné à l'informaticien *Guy L. Steele* l'idée de la chanson *The TELNET Song* (1984).

Les paroles de cette chanson, qui dépendent d'un entier N (l'auteur recommande $N = 4$), sont elles-mêmes écrites comme un programme. On en résume ici la structure :

Algorithme 76.5 — : TelnetSong (structure)

Entrée : $N \in \mathbb{N}$

- **Chanter** deux couplets d'introduction, de longueur fixe.
- Pour I allant de 1 à N
  - **Chanter** un couplet de longueur fixe, qui se termine par 2^I fois « control-uparrow » suivis de « Q ».
- **Chanter** un couplet de transition, de longueur fixe.
- Pour I allant de N à 1
  - **Chanter** un autre couplet de longueur fixe, qui se termine lui aussi par 2^I fois « control-uparrow » suivis de « Q ».
- **Chanter** les couplets de conclusion, de longueur fixe (ils contiennent un nombre constant de « control-uparrow »).

Question 4

Quelle est la complexité de cette chanson ?

Corrigé

Ici, on a $O(1)$ mots chantés hors des boucles, et $O(N)$ mots dans les boucles sans compter les `control-uparrow`.

Le nombre de `control-uparrow` dans les boucles est $2\sum_{i=1}^{N} 2^i = 2(2^{N+1}-2) = 2^{N+2}-4$, et chacun compte un nombre constant de mots.

Ainsi, en tout, le nombre de mots chantés est en $\Theta(2^N)$: la complexité est exponentielle.

76.5 Chanter des entiers

L'hypothèse faite sur le fait que chanter $i \in \mathbb{N}$ est en temps constant est peu réaliste. Il suffit de comparer le temps mis entre le fait de prononcer 1 et 123456789.

On va considérer ici le nombre de mots réellement présents dans la lecture d'un entier. Comme il y a des traits d'union, on considère que *saute-mouton* comporte deux mots.

Par exemple,

- 3 se lit *trois* et comporte un mot ;
- 123 se lit *cent vingt-trois* et comporte 3 mots (*vingt-trois* compte comme deux mots) ;
- 123 456 789 se lit *cent vingt-trois millions quatre cent cinquante-six mille sept cent quatre-vingt-neuf* et comporte donc 14 mots.

On pourra utiliser : chiffre en lettre.

Question 5

Exprimer sous la forme $O(f(n))$ le nombre de mots dans la lecture de l'entier n.

Note : on pourra faire une majoration grossière tant qu'elle est tout de même réaliste.

Corrigé

On peut regrouper les chiffres par paquets de trois : un élément de la forme $(10^2a + 10b + c)10^{3k}$, avec $a, b, c \in [\![0, 9]\!]$, se lit avec au plus 7 mots :

- au plus 2 mots pour 10^2a : « a cent » ;
- au plus 4 mots pour $10b + c$: par exemple « quatre-vingt-dix-sept » ou « soixante et onze » ;
- 1 mot pour 10^{3k} : « mille », « millions », etc.

Un entier $n \geq 1$ comporte $\lfloor \log_{1000} n \rfloor + 1$ paquets de trois chiffres. Ainsi, on peut majorer le nombre de mots dans la lecture de n par $7(\lfloor \log_{1000} n \rfloor + 1) = O(\log n)$.

Remarquons qu'on aurait la même complexité en le lisant en binaire, avec $\lfloor \log_2 n \rfloor + 1$ chiffres : c'est une bonne manière de *ressentir* la constante.

Question 6

En déduire des complexités affinées des chansons `DMàRendre` et `NDaysOfChristmas`.

Corrigé

- Dans `DMàRendre` on fait n itérations de boucle où on va prononcer deux nombres inférieurs ou égaux à n. Donc on chante $O(n \log n)$ mots maintenant qu'on prononce réellement les nombres.
- Dans `NDaysOfChristmas` c'est dans la boucle intérieure, qui s'effectue $\frac{n(n-1)}{2}$ fois, qu'on va prononcer un nombre $j \leq n$, donc $O(n^2 \log n)$ mots (le « ith » de la boucle principale n'ajoute que $O(n \log n)$ mots).