


# Informatique

Devoir surveillé 3

16 décembre 2021




Durée totale 2h.
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.

## 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
```

**Q01** Montrer que la fonction précédente calcule $x^n$ en considérant sa terminaison acquise.

*Démonstration.* 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 $i = 0$ 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 $p = x^{n+1-1} = x^n$. □

**Q02** Combien de multiplications effectue cette fonction pour calculer $x^n$ ?

*Démonstration.* On effectue une multiplication par itérations 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
```

**Q03** Donner l'état des variables à chaque itération dans l'appel de `expo_binaire 3 15`.

*Démonstration.*

| p | a | k |
|---|---|---|
| 1 | 3 | 15 |
| 3 | 9 | 7 |
| 27 | 81 | 3 |
| 2187 | 6561 | 1 |
| 14348907 | 43046721 | 0 |

□

**Q04** Prouver la correction de cette fonction :

1. sa terminaison ;
2. le fait qu'elle calcule $x^n$.

*Démonstration.* 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 $I(x, 1, k)$ 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 $2^p \leqslant n < 2^{p+1}$.
- $\lceil \log_2 n \rceil$ l'unique entier $p$ tel que $2^{p-1} < n \leqslant 2^p$.
- $\nu(n)$ le nombre de 1 dans l'écriture binaire de $n$. Plus précisément, si $n = a_0 + a_1 2 + a_2 4 + \dots a_{d-1} 2^{d-1}$ avec $a_0, \dots, a_{d-1} \in \{0, 1\}$ alors il y a exactement $\nu(n)$ des $a_i$ qui valent 1.

**Q05** On suppose que $n > 0$. Montrer que la fonction `expo_binaire` effectue $\lceil \log_2 n \rceil + \nu(n)$ multiplications pour calculer $x^n$.

*Démonstration.* On remarque que si $n = 2^d$ alors on a effectue directement $d$ itérations et $k$ n'est impair que lors de la dernière itérations, on effectue donc $d + 1 = \lceil \log_2 n \rceil + \nu(n)$ multiplications.
Sinon $n = a_0 + a_1 2 + \dots + a_{d-2} 2^{d-2} + 2^{d-1}$ avec $a_0, \dots, a_{d-2} \in \{0, 1\}$, i.e. $n = \overline{1 a_{d-2} \dots a_0}^2$ s'écrit avec $d$ chiffres en binaire, et on a $2^{d-1} < n < 2^d$, donc $d = \lceil \log_2 n \rceil$.
A chaque étape, on décale dans $k$ l'écriture binaire de $n$ vers la droite :

$$\overline{1 a_{d-2} \dots a_0}^2 \to \overline{1 a_{d-2} \dots a_1}^2 \cdots \to \overline{1 a_{d-2}}^2 \to \overline{1}^2$$

Donc `k mod 2` va prendre les valeurs $a_0, a_1, \dots, a_{d-2}, 1$ et on aura bien $\nu(n)$ valeurs 1 entrainant une multiplication. Au final, on a donc $\nu(n)$ multiplications issues du test et $\lceil \log_2 n \rceil$ multiplications déduites de la mise au carré de $a$ soit $\lceil \log_2 n \rceil + \nu(n)$ multiplications en tout.
*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 $x = x^1$.

**Q06** Comparer les deux méthodes pour $n = 55$ et $n = 33$. Que peut-on en conclure ?

*Démonstration.* — 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 mise au carré et 4 mise 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.

**Q07** Donner, sans les justifier, les chaines correspondant aux deux méthodes pour 16, 27 et 125.

*Démonstration.* — 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 $(1, 2, 3, 9, 18, 27)$ avec la méthode du facteur.
— pour 125 : $(1, 2, 4, 5, 8, 13, 16, 29, 32, 61, 64, 125)$ avec la méthode binaire et $(1, 2, 3, 5, 10, 20, 25, 30, 60, 120, 125)$ avec la méthode du facteur.

□

**Définition 1.** *Une suite finie* $(n_0, \dots, n_r)$ *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 $x^{n_k} = x^{n_i} \times x^{n_j}$.

**Q08**

1. Montrer par récurrence sur $r$ que toute chaine $(n_0, \dots, n_r)$ de calcul pour $x^n$ vérifie
$$\forall k \in [\![0, r]\!], n_k \leqslant 2^k$$
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.

*Démonstration.* 1. **Init** Si $r = 0$ la seule chaine est $(1)$ et on a bien $1 \leqslant 2^0$.

**Her** Si c'est vrai pour $r - 1 > 0$, soit $(n_0, \dots, n_r)$ une chaine de calcul on a $(n_0, \dots, n_{r-1})$ chaine de calcul donc $\forall k \in [\![0, r-1]\!], n_k \leqslant 2^k$ et ainsi par hypothèse on a une décomposition $n_r = n_i + n_j \leqslant 2^i + 2^j \leqslant 2 \times 2^{r-1} = 2^r$. 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 $n_r = n \leqslant 2^r$ on a $\log_2 n \leqslant r$ et comme $r$ est un entier $\lceil \log_2 n \rceil \leqslant r$.
3. C'est atteint pour les $2^r$ où on a une chaine $(1, 2, 4, \cdots, 2^{r-1}, 2^r)$ de longueur $r = \log_2(2^r)$.

□

## 2 Breizhsort

Si `t` est un tableau et $0 \leqslant$ `i` $\leqslant$ `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 `retourne t 2 4` change `t` en `[| 1; 2; 5; 4; 3; 6 |]` car `t[2:4] = [| 3; 4; 5 |]`.

**Q09** 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`.

*Démonstration.* `[| 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
```

**Q10** Montrer que `retourner` termine.

*Démonstration.* On remarque que $\mathtt{j}-\mathtt{i}$ est entière, si elle est négative ou nulle on termine, sinon on effectue un appel tel que $(\mathtt{j}-1)-(\mathtt{i}+1)=\mathtt{j}-\mathtt{i}-2<\mathtt{j}-\mathtt{i}$ donc elle diminue strictement.
Ainsi, il ne peut il y avoir qu'un nombre fini d'appel récursifs, la fonction termine. □

**Q11** Montrer que `retourner` correspond bien à la spécification de l'énoncé. On pourra montrer par récurrence sur $\mathtt{j}-\mathtt{i}+1$ [a] que

$$\forall \mathtt{k} \in [\![\mathtt{i},\mathtt{j}]\!], \texttt{t.(k)} = \texttt{t'.(i+j-k)}$$

où `t'` est la valeur initiale de `t`.

a. le $+1$ n'est là que pour déplacer l'initialisation au cas de base

*Démonstration.* On suit l'énoncé.

**Init** Pour $\mathtt{j}-\mathtt{i}+1=0$ on a $\mathtt{j}=\mathtt{i}-1<\mathtt{i}$ donc $[\![\mathtt{i},\mathtt{j}]\!]=\emptyset$ et la propriété est vérifiée directement.

**Her** Si la propriété est vérifiée pour $\mathtt{n}\geqslant 0$ on considère $\mathtt{i},\mathtt{j}$ tels que $\mathtt{j}-\mathtt{i}+1=\mathtt{n}+1$, c'est-à-dire que $\mathtt{j}-\mathtt{i}=\mathtt{n}\geqslant 0$. Par hypothèse, on a

$$\forall \mathtt{k} \in [\![\mathtt{i}+1,\mathtt{j}-1]\!], \texttt{t.(k)} = \texttt{t'.(i+1+j-1-k)t'.(i+j-k)}$$

De plus, on effectue l'échange entre les indices `i` et `j` et 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
```

**Q12** Montrer qu'en sortie de l'appel à `placer t k` le tableau `t` vérifie :

$$\forall \mathtt{i} \in [\![0, \mathtt{k}]\!], \mathtt{t.(i)} \leqslant \mathtt{t.(k)}$$

*Démonstration.* — On commence par montrer que la fonction `indice_max` fait ce qu'on attend d'elle. Pour cela, on considère la propriété

$$\mathrm{I}(\mathtt{i}, \mathtt{m}) := \mathtt{t.(m)} = \max_{0 \leqslant \mathtt{j} < \mathtt{i}} \mathtt{t.(j)}$$

dont on montre facilement que c'est un invariant de boucle. En sortie, on a donc $\mathrm{I}(\mathtt{k}+1, \mathtt{m}) := \mathtt{t.(m)} = \max_{0 \leqslant \mathtt{j} \leqslant \mathtt{k}} \mathtt{t.(j)}$ 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 $\mathtt{i} = \mathtt{k}$, 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
```

**Q13** Donner l'état du tableau à chaque itération dans l'appel à `rumman t` où `t = [| 0; 5; 7; 1; 2 |]`.

*Démonstration.* — `[| 0; 5; 7; 1; 2 |]`
- $\mathtt{k} = 4$, → `[| 7; 5; 0; 1; 2 |]` → `[| 2; 1; 0; 5; 7 |]`
- $\mathtt{k} = 3$, pas de changement
- $\mathtt{k} = 2$, → `[| 2; 1; 0; 5; 7 |]` → `[| 0; 1; 2; 5; 7 |]`
- $\mathtt{k} = 1$, pas de changement
- $\mathtt{k} = 0$, pas de changement

□

**Q14** En utilisant un invariant de boucle, montrer que la fonction `rumman` trie le tableau passé en argument dans l'ordre croissant.

*Démonstration.* On commence par remarquer que les seules modifications du tableau sont effectuées par `retourne` 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 boucles :

$$\mathrm{I}(\mathtt{k}) := \mathtt{t}[\mathtt{k}+1 : \mathtt{n}-1] \text{ est trié et } \forall \mathtt{i} \in \mathtt{range0k}, \forall \mathtt{j} \in [\![\mathtt{k}+1, \mathtt{n}-1]\!], \mathtt{t.(i)} \leqslant \mathtt{t.(k)}$$

où $|\mathtt{t}| = \mathtt{n}$, en raisonnant sur l'indice comme auparavant : on l'incrémente 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 $\mathtt{k}+1$ est invalide au départ.

**Init** Au départ $\mathtt{k} = \mathtt{n}-1$ et donc $\mathtt{t}[\mathtt{k}+1 : \mathtt{n}-1] = \emptyset$ et la seconde partie de la propriété est triviale.

**Her** Si la propriété est vérifiée au début d'une itération, On sait que $\mathtt{t}[\mathtt{k}+1 : \mathtt{n}-1]$ 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 $\mathtt{t}(\mathtt{k}+1)$ par hypothèse. Donc on a bien la propriété en fin d'itération.

En sortie de boucle, on a donc `t` trié. □

**Q15** Donner, en les justifiant, les complexités temporelles dans le pire des cas des fonctions `retourne`, `placer`, `indice_max` et `rumman`.

*Démonstration.*
— `retourne i j` effectue des appels en enlevant 2 à $j-i$ à chaque appel, on a donc $O(j-i)$ appels.
— `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 $i \leqslant k$.
— `rumman` est en $\sum_{k=0}^{n-1} O(k) \leqslant C\sum_{k=0}^{n-1} = C\frac{n(n-1)}{2}$ donc en $O(n^2)$.

□

## 3 Récursif

**Q16** 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)
```

**Q17** 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
```

**Q18** Ici, on tire partie 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 auxillaire 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 []
```

**Q19** Question très simple mais qui demande de bien comprendre l'enoncé !

```
let ajoute_assoc l a b =
    (a,b) : : l
```

**Q20** On effectue une simple recherche dans la liste mais en tirant parti du fait que les éléments sont des couples. A 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
```

**Q21**

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
```

**Q22**

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
    | _ -> []
```

**Q23**

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
```

**Q24**

```
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
```

**Q25**

```
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
```

# 4 Impératif

**Q26**

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

**Q27**

```
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
```

**Q28**

```
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
```

**Q29**

Attention aux cas de bases 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
```

**Q30**

```
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
```

**Q31**

```
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
```

**Q32**

```
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
```

**Q33**

```
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
```

**Q34**

```
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;
    incr r
```

```
    done;
    !r
     end else 0
```

**Q35**

```
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
```