

Informatique

Devoir surveillé 4 - Corrigé

1 février 2022



1 A link to a past TP

Q01 Écrire une fonction de prototype

```
int longueur(liste *);
```

qui renvoie la longueur d'une liste.

Démonstration.

```
int longueur(liste *l)
{
    int n = 0;
    maillon *m = l->debut;
    while(m != NULL)
    {
        n++;
        m = m->suivant;
    }
    return n;
}
```

□

Q02 Écrire une fonction de prototype

```
int somme(liste *);
```

qui renvoie la somme des valeurs des éléments d'une liste.

Démonstration.

```
int somme(liste *l)
{
    int s = 0;
    maillon *m = l->debut;
    while(m != NULL)
    {
        s = s + m->valeur;
        m = m->suivant;
    }
    return s;
}
```

□

Q03 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.

Démonstration.

```
void liste_ajout_en_tete(liste *l, int valeur)
{
    l->debut = creer_maillon(valeur, l->debut);
    if (l->fin == NULL)
        l->fin = l->debut;
}
```

□

Q04 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.

Démonstration.

```
maillon *liste_cherche_maillon(liste *l, int valeur)
{
    maillon *m = l->debut;
    while(m != NULL && m->valeur != valeur)
    {
        m = m->suivant;
    }
    return m;
}
```

□

Q05 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.

Démonstration.

```
void liste_ajout_maillon_apres(liste *l, maillon *m, int valeur)
{
    m->suivant = creer_maillon(valeur, NULL);
    if (l->fin == m)
        l->fin = m->suivant;
}
```

□

Q06 Expliquer comment cette structure permet d'implémenter une pile et donner les différentes complexités des opérations de la structure liste.

Démonstration. L'ajout et la suppression en tête sont en $O(1)$. □

2 Domino's Tiling

Q07 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.

Démonstration. 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. □

Q08 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).

Démonstration.

```
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 i (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 -> []
```

□

Q09 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.

Démonstration.
```
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
```

□

Q10 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

Démonstration.
```
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
```

□

Q11 Écrire une fonction de signature

```
val complet : case array array -> bool
```

qui indique si une grille est complètement pavée.

Démonstration.
```
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
```

□

Q12 Écrire une fonction récursive de signature

```
val enumere : case array array -> unit
```

qui enumere 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 domino correspondante telle qu'obtenue avec `extrait_pavage`.
L'exception est définie par :

```
exception Solution of domino list
```

Démonstration.
```
exception Solution of domino list
let enumere a =
    let rec aux l =
        if complet a
        then raise (Solution l)
        else List.iter
            (fun m ->
                place a m;
                aux (m::l);
                enleve a m) (mouvements a)
    in aux []
```

□

Q13 Écrire une fonction de signature :

```
val pavage : int -> int -> domino list
```

telle que `pavage w h` renvoie une liste de domino correspondant à un pavage d'une grille de dimension $w \times h$.

Démonstration.
```
let pavage w h = enumere (Array.make_matrix w h E)
```

□

Q14 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.

Démonstration.
```
let enumere_tous a =
    let rec aux l =
        if complet a
        then [ l ]
        else let acc = ref [] in
        begin
            List.iter
                (fun m ->
                    place a m;
                    acc := aux (m::l) @ !acc;
                    enleve a m) (mouvements a);
            !acc
```

```
        end
    in aux []

let pavages w h = enumere_tous (Array.make_matrix w h E)
```

□

3 Matoucheespaceestcassée

Q15 Écrire une fonction de signature :

```
val est_mot : char array -> int -> int -> bool
```

tel que l'appel à `est_mot t i j` renvoie `true` lorsque le `t[i..j]` est dans le dictionnaire.

Démonstration.

```
let est_mot c i j =
    let m = Array.init (j-i+1) (fun k -> c.(i+k)) in
    List.mem m dico
```

□

Q16 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 $n = |t|$.

Démonstration.

```
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
```

□

Q17 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'élément 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 $T_p = O(2^p)$.

Démonstration.

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 à 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, $T_p = O(2^p)$.

□