

DS 2 d'informatique MP2I corrigé

2 heures

Calculatrice et documents de cours interdits.

I Questions de cours

On utilise le type suivant pour stocker un vecteur du plan (x étant l'abscisse et y l'ordonnée) :

```
type vect = {x : float; y : float}
```

1. Définir le vecteur $\begin{pmatrix} 3 \\ -2 \end{pmatrix}$.

Solution :

```
let v = {x = 3.0; y = -2.0}
```

2. Écrire des fonctions `add : vect -> vect -> vect` et `prod : vect -> vect -> float` permettant de calculer l'addition de vecteurs et le produit scalaire, respectivement.

Solution :

```
let add u v = {x = u.x + v.x; y = v.x + v.y};;  
let prod u v = u.x * v.x + u.y + v.y;;
```

3. Quelles sont les opérations permises par une file ? Décrire brièvement 2 implémentations possibles de file, une persistante (c'est-à-dire fonctionnelle) et l'autre mutable (c'est-à-dire impérative).

Solution : Voir cours.

II Liste de profondeur quelconque

On considère le type `list_or_elem` suivant, permettant de définir des listes de profondeur quelconque (des listes de listes, ou des listes de listes de listes...) :

```
type 'a list_or_elem = E of 'a | L of 'a list_or_elem list
```

Ainsi une valeur de type `list_or_elem` est soit `E e` (représentant un élément `e`), soit `L l` (représentant une liste de `list_or_elem`).

Écrire une fonction `flatten : 'a list_or_elem -> 'a list` telle que, si `l` est de type `'a list_or_elem`, `flatten l` renvoie la liste des éléments de `l`. Par exemple, `flatten (L [E 1; E 2])` doit renvoyer `[1; 2]` et `flatten (L [E 5; L [L [E 2; E 0]; E 7]])` doit renvoyer `[5; 2; 0; 7]`.

Solution :

```
let rec flatten = function  
  | E e -> [e]  
  | L l -> let rec aux = function (* concatène les listes de l *)  
            | [] -> []  
            | e::q -> (flatten e) @ (aux q) in  
            aux l
```

Ou, en utilisant `List.fold_left f acc l` qui applique `f` sur une liste `l` en accumulant le résultat dans `acc` :

```
let rec flatten = function
| E e -> [e]
| L l -> List.fold_left (fun l ll -> l @ flatten ll) [] l
```

III Prochain plus grand élément

Étant donné un tableau t d'entiers, on veut calculer un tableau tp des **prochains plus grand éléments**, c'est-à-dire un tableau de même taille que t et tel que, pour tout indice i , $tp.(i)$ soit le plus petit indice j tel que $j > i$ et $t.(j) > t.(i)$. Si cet indice n'existe pas, on mettra -1 dans $tp.(i)$.

Par exemple, si $t = [4; 2; 7; 0; 6]$ alors $tp = [2; 2; -1; 4; -1]$. En effet :

- Le premier élément supérieur à $t.(0) = 4$, après l'indice 0, est $t.(2) = 7$. Donc $tp.(0) = 2$.
 - Le premier élément supérieur à $t.(1) = 2$, après l'indice 1, est $t.(2) = 7$. Donc $tp.(1) = 2$.
 - Il n'y a pas d'élément supérieur à $t.(2) = 7$ après l'indice 2. Donc $tp.(2) = -1$.
 - Le premier élément supérieur à $t.(3) = 0$, après l'indice 3, est $t.(4) = 6$. Donc $tp.(3) = 4$.
 - Il n'y a pas d'élément supérieur à $t.(4) = 6$ après l'indice 4. Donc $tp.(4) = -1$.
1. Écrire une fonction `next : int array -> int -> int` telle que, si t est un tableau d'entiers et i un indice de t , `next t i` renvoie le plus petit indice j tel que $j > i$ et $t.(j) > t.(i)$. Si cet élément n'existe pas, on renverra -1 .

Solution : En utilisant une exception (il y a bien d'autres façons de faire...) :

```
exception Found of int;;

let next t i =
  try for j = i + 1 to Array.length t - 1 do
    if t.(j) > t.(i) then raise (Found j)
  done;
  -1
with Found j -> j
```

2. Écrire une fonction `next_greater : int array -> int array` telle que `next_greater t` renvoie tp . Quelle est sa complexité ?

Solution : Si t est de taille n , on applique n fois `next` ce qui donne une complexité $O(n^2)$.

```
let next_greater t =
  let n = Array.length t in
  let tp = Array.make n (-1) in
  for i = 0 to n - 1 do
    tp.(i) <- next t i
  done;
  tp
```

Dans la suite, on va utiliser une pile avec le module `Stack` d'OCaml dont voici un exemple d'utilisation :

```
let s = Stack.create ();; (* créer une pile vide *)
Stack.push s 1;; (* ajouter 1 à la pile *)
Stack.top ();; (* affiche l'élément du dessus de la pile, sans le supprimer *)
Stack.pop s;; (* supprime et renvoie l'élément du dessus de la pile *)
```

On considère maintenant l'algorithme suivant :

```

let f t =
  let n = Array.length t in
  let tp = Array.make n (-1) in
  let s = Stack.create () in
  for i = 0 to n - 1 do
    while not (Stack.is_empty s) && t.(Stack.top s) <= t.(i) do
      tp.(Stack.pop s) <- i
    done;
    Stack.push i s
  done;
  tp

```

3. Exécuter `f [14; 2; 7; 0; 6]` en donnant le contenu de `tp` et de `s` après chaque itération de la boucle `for`.

Solution : Voici le contenu de `tp` et `s` (représenté sous forme de liste, avec le dessus de la pile tout à droite) à la fin de l'itération i de la boucle `for` :

i	<code>tp</code>	<code>s</code>
0	<code>[-1; -1; -1; -1; -1]</code>	<code>[0]</code>
1	<code>[-1; -1; -1; -1; -1]</code>	<code>[0; 1]</code>
2	<code>[2; 2; -1; -1; -1]</code>	<code>[2]</code>
3	<code>[2; 2; -1; -1; -1]</code>	<code>[2; 3]</code>
4	<code>[2; 2; -1; 4; -1]</code>	<code>[2; 4]</code>

4. Quelles est la complexité de `f t` ? On donnera le plus petit terme possible dans le $O(\dots)$.

Solution : Soit n la taille de `t`.

Chaque élément de `t` est ajouté exactement une fois dans `s`, et supprimé au plus une fois. Donc le nombre total de `Stack.push i s` est n et le nombre total de `tp.(Stack.pop s) <- i` est au plus n .

Comme `Array.make n (-1)` est aussi en $O(n)$, la complexité totale est $O(n) + O(n) + O(n) = \boxed{O(n)}$.

5. Si `s` contient les éléments i_1, \dots, i_p dans cet ordre (où i_p est l'élément sur le dessus de la pile), que peut-on conjecturer sur `t.(i1)`, ..., `t.(ip)` (dans quel ordre sont-ils ordonnés) ? Le démontrer.

Solution : On considère l'invariant de boucle suivant :

\mathcal{H} : Si `s` contient i_1, \dots, i_p alors `t.(i1)`, ..., `t.(ip)` est décroissant.

Démontrons \mathcal{H} par récurrence sur le nombre de passage dans la boucle `for`.

Avant que l'on passe dans le `for` pour la 1ère fois, `s` est vide. Donc \mathcal{H} est vraie.

Supposons que \mathcal{H} soit vraie à la fin de la i ème itération du `for`.

Lors de la $(i + 1)$ ème itération, on supprime dans le `while` tous les indices du dessus de la pile dont les valeurs (dans `t`) sont supérieures à `t.(i + 1)`. Comme \mathcal{H} était vraie à l'itération précédente, on peut enlever "du dessus de la pile" dans la phrase précédente.

Ainsi, les valeurs des indices dans `s` restent décroissantes, ce qui montre que \mathcal{H} reste vraie tout au long de l'algorithme.

6. Que peut-on dire de `tp.(i)` si i est dans `s` ?

Solution : `tp.(i)` vaut -1 dans ce cas.

7. (difficile) Montrer que `f t` renvoie bien le tableau des prochains plus grand éléments.

IV Recherche de doublon

Dans tout l'exercice, on considère un tableau `t` d'entiers contenant au plus 2 fois chaque élément.

Par exemple, `t` peut être égal à `[15; -1; 0; -2; 5; -2]` mais pas `[15; -1; 5; 0; -2; 5; -2]` (5 apparaît 3 fois).

On veut trouver tous les doublons dans `t` (c'est-à-dire les éléments apparaissant plusieurs fois).

1. Écrire une fonction `doublons : 'a array -> (int * int) list` renvoyant la liste des couples d'indices correspondants à des doublons dans un tableau `t`, en $O(n^2)$.

Par exemple, `doublons [|-1; 3; 0; -2; 3; -2|]` peut renvoyer `[(1, 4); (3, 5)]`. En effet, `t.(1) = t.(4) = 3` et `t.(3) = t.(5) = -2`.

Solution :

```
let rec doublons t =
  let n = Array.length t in
  let rec aux i j =
    if i = n then []
    else if j = n then aux (i + 1) (i + 2)
    else if t.(i) = t.(j) then (i, j)::aux (i + 1) (i + 2)
    else aux i (j + 1) in
  aux 0 1
```

2. On suppose avoir à disposition une fonction `sort : 'a array -> unit` triant un tableau de taille n en $O(n \log(n))$. En déduire une fonction `doublons2` réalisant la même chose que `doublons`, mais avec une meilleure complexité. Pour trier en conservant les indices du tableau, on pourra utiliser le fait qu'il est possible de comparer deux couples en OCaml : $(a, b) < (c, d)$ si $a < c$ ou $a = c$ et $b < d$.

Solution :

```
let doublons2 t =
  let n = Array.length t in
  let couples = Array.make n (0, 0) in
  for i = 0 to n - 1 do
    couples.(i) <- (t.(i), i)
  done;
  sort couples;
  let rec aux i =
    if i = n-1 then []
    else if fst couples.(i) = fst couples.(i+1)
      then (snd (couples.(i)), snd couples.(i+1))::aux (i+1)
      else aux (i+1) in
  aux 0
```

On utilise maintenant un dictionnaire, avec à disposition les fonctions suivantes en $O(1)$:

```
create : unit -> ('k*'v) dict
(* create () crée un nouveau dictionnaire vide *)
add : ('k*'v) dict -> ('k * 'v) -> unit
(* add d k v ajoute une association de la clé k à la valeur v.
Si la clé k existait déjà, elle est remplacée *)
get : ('k*'v) dict -> 'k -> 'v option
(* get d k renvoie Some v où v est la valeur associée à k,
ou None si la clé k n'existe pas *)
```

3. Écrire une fonction `doublons3 : 'a array -> (int * int) list` réalisant la même chose que `doublons`, mais en complexité linéaire.

Solution :

```
let doublons3 t =
  let d = create () in
  let rec aux i =
    if i = Array.length t then []
    else match get d t.(i) with
      | None -> add d (t.(i), i); aux (i + 1)
      | Some j -> (i, j)::aux (i + 1) in
  aux 0
```