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 $\begin{pmatrix} 3 \\ -2 \end{pmatrix}$.
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.
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).

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_of_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]`.

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

Dans la suite, on va utiliser une pile avec le module Stack d'OCaml dont voici les opérations (en O(1)) :

```
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. Éxecuter `f [|4; 2; 7; 0; 6|]` en donnant le contenu de `tp` et de `s` après chaque itération de la boucle `for`.
4. Quelles est la complexité de `f t` ? On donnera le plus petit terme possible dans le O(...).
5. Si `s` contient les éléments $i_1, ..., i_p$ dans cet ordre (où i_p est l'élément sur le dessus de la pile), que peut-on conjecturer sur `t.`(i_1), ..., `t.`(i_p) (dans quel ordre sont-ils ordonnés) ? Le démontrer.
6. Que peut-on dire de `tp.(i)` si `i` est dans `s` ?
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 `[|5; -1; 0; -2; 5; -2|]` mais pas `[|5; -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`.
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 $b < d$.

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

```
create : unit -> ('k*'v) dict
(* create () créé 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.