

Ce devoir est composé de 5 exercices indépendants. Pour répondre à une question, il est permis de réutiliser une fonction demandée à une question précédente, même sans avoir réussi à coder cette fonction.

**Exercice 1 (listes) :** Dans cet exercice, on veut implémenter nous même certaines fonctions sur les listes OCaml : il est interdit d'utiliser toute fonction du module `List`.

1. Écrire une fonction `longueur` : `'a list -> int` prenant en argument une liste et renvoyant sa longueur.
2. Écrire une fonction `compter` : `'a -> 'a list -> int` prenant en arguments une valeur `x` et une liste `l`, et renvoie le nombre de fois que `x` apparaît dans `l`.
3. Écrire une fonction `iter` : `('a -> unit) -> 'a list -> unit` prenant en arguments une fonction `f` de type `'a -> unit` et une liste `l` d'éléments de type `'a`, et appelle la fonction `f` successivement sur les éléments de `l` (dans l'ordre).  
Autrement dit, `iter f [a1 ; a2 ; ... an]` doit être équivalent à `f a1 ; f a2 ; ... ; f an`.
4. Écrire une fonction `filter` : `('a -> bool) -> 'a list -> 'a list` prenant en arguments une fonction `f` de type `'a -> bool` (`f` est un prédicat), et une liste `l` d'éléments de type `'a`, et renvoyant la liste des éléments de `l` qui vérifient le prédicat `f` (dans l'ordre).
5. Écrire une fonction `append` : `'a list -> 'a list -> 'a list` telle que `append l1 l2` renvoie une nouvelle liste issue de la concaténation de `l1` et `l2` (l'utilisation de l'opérateur `@` est interdit).
6. Écrire une fonction `flatten` : `'a list list -> 'a list` prenant en argument une liste de listes et renvoyant la concaténation de toutes ces listes (dans l'ordre).

**Exercice 2 (tableaux et exceptions) :** Dans cet exercice, les seules fonctions du module `Array` autorisées sont `Array.length` et `Array.make`.

1. Écrire une fonction `somme_produit` : `int array -> int array -> int` prenant en arguments deux tableaux d'entiers `a` et `b` et renvoyant la somme suivante (où `n` est la longueur des tableaux) :

$$\sum_{i=0}^{n-1} a_i \times b_i$$

On lèvera l'exception `Mauvaises_longueurs` (supposée déjà créée) si les tableaux ne sont pas de même longueur.

2. Écrire une fonction `liste_vers_tableau` : `'a list -> 'a array` prenant en argument une liste et renvoyant un tableau contenant les mêmes éléments que la liste, dans le même ordre.
3. Écrire une fonction `tableau_vers_liste` : `'a array -> 'a list` prenant en argument un tableau et renvoyant une liste contenant les mêmes éléments que le tableau, dans le même ordre.
4. Écrire une fonction `test_option` : `('a -> bool) -> 'a array -> 'a option array` prenant en arguments une fonction `f` de type `'a -> bool` et un tableau `t` d'éléments de type `'a`, et renvoyant un tableau `t'` tel que :
  - `t'` est de même longueur que `t` ;
  - pour tout indice `i` de `t`, en notant `x` la valeur contenue dans `t` à l'indice `i`, la case d'indice `i` de `t'` contient :
    - `Some x` si `f(x)` est vrai ;
    - `None` sinon.
5. Écrire une fonction `premier_indice` : `('a -> bool) -> 'a array -> int option` prenant en arguments une fonction `f` de type `'a -> bool` et un tableau `t` d'éléments de type `'a`, et renvoyant :
  - `Some i` où `i` est le plus petit indice d'un élément `x` de `t` tel que `f(x)` soit vrai (si un tel `i` existe) ;
  - `None` si un tel `i` n'existe pas.

On pourra utiliser l'exception ci-dessous pour interrompre la recherche dès qu'un tel `i` est trouvé :

```
exception Trouve of int
```

**Exercice 3 (typage) :** Pour chacune des fonctions suivantes :

- si elle est acceptée par OCaml, indiquer son type ;
- sinon, expliquer pourquoi elle est refusée.

1. 

```
let fonction1 x y =  
  if y mod 2 = 0 then x.(y) else 0.
```

2. 

```
let fonction2 x y = match y with  
  | 0 -> 0.  
  | _ -> x y
```

3. 

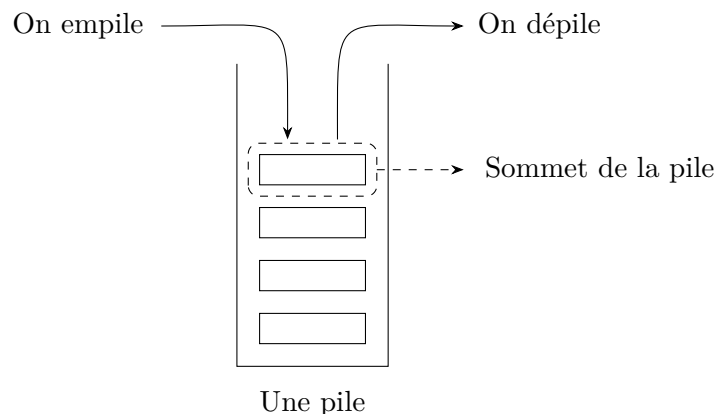
```
let fonction3 x y z = (x y) (y z)
```

**Exercice 4 (d'après Baccalauréat NSI – sujet 0) :** Dans ce sujet, l'utilisation de listes ou de tableaux est **interdit**.

Une pile est une structure de données abstraite fondée sur le principe “dernier arrivé, premier sorti”, ou LIFO (“last in, first out”). Autrement dit, lorsqu'on souhaite sortir un élément de la pile, on sort l'élément le plus récent (cet élément est au **sommet de la pile**).

On suppose que cette structure de données est déjà implémentée par un type `'a pile`, et munie des fonctions suivantes :

```
— Fonctions du type 'a pile —  
creer_pile : unit -> 'a pile  
pile_vide : 'a pile -> bool  
empiler : 'a pile -> 'a -> unit  
sommet : 'a pile -> 'a  
depiler : 'a pile -> 'a
```



Ainsi :

- l'instruction `let p = creer_pile () in ...` crée une nouvelle pile ne contenant aucun élément ;
  - l'instruction `pile_vide p` renvoie `true` si la pile `p` est vide, et `false` sinon ;
  - l'instruction `empiler p x` ajoute la valeur `x` au sommet de la pile `p` ;
  - l'instruction `sommet p` renvoie la valeur se trouvant au sommet de la pile `p` sans modifier le contenu de la pile `p` ;
  - l'instruction `depiler p` renvoie la valeur se trouvant au sommet de la pile `p` et supprime cette valeur la pile `p`.
1. On suppose dans cette question que la variable `p` contient une pile dont le contenu est le suivant (les éléments étant empilés par le haut) :

4
2
5
8

Que contiennent les variables `p` et `q` après l'exécution du code ci-dessous ?

```
let q = creer_pile () in  
while not (pile_vide p) do  
  empiler q (depiler p)  
done
```

On appelle **hauteur** d'une pile le nombre d'éléments qu'elle contient.

2. Écrire une fonction `hauteur_pile` : 'a pile -> int prenant en argument une pile et renvoyant sa hauteur.

**Important** : à la fin de votre fonction, la pile prise en entrée doit avoir retrouvé son état d'origine.

3. Écrire une fonction `max_pile` : 'a pile -> int -> int telle que `max_pile p i` renvoie la position `j` du plus grand élément parmi les `i` derniers éléments de la pile `p`. Par convention, la position du sommet de la pile est 1.

**Important** : à la fin de votre fonction, la pile `p` doit avoir retrouvé son état d'origine.

Exemple

```
(* p contient la pile de la question 1 *)
# max pile p 2 ;;
- : int = 1
```

4. Écrire une fonction `retourner` : 'a pile -> int -> unit telle que `retourner p j` modifie `p` en inversant l'ordre des `j` derniers éléments empilés dans `p` et ne renvoie rien.

**Exemple** : si `p` contient la pile de la question 1, alors après l'instruction `retourner p 3`, l'état de la pile `p` sera :

5
2
4
8

L'objectif de cette question est de trier une pile de crêpes.

On modélise une pile de crêpes par une pile d'entiers représentant le diamètre de chaque crêpe. On souhaite réordonner les crêpes de la plus grande (placée en bas de pile) à la plus petite (placée au sommet de la pile).

On dispose uniquement d'une spatule que l'on peut insérer dans le pile de crêpes de façon à retourner l'ensemble des crêpes qui lui sont au dessus.

Le principe est le suivant :

- on recherche la plus grande crêpe ;
  - on retourne la pile à partir de cette crêpe de façon à mettre cette plus grande crêpe tout en haut de la pile ;
  - on retourne l'ensemble de la pile de façon à ce que cette plus grande crêpe se retrouve tout en bas ;
  - la plus grande crêpe étant à sa place, on recommence le principe avec le reste de la pile.
5. Écrire une fonction `tri_crepes` : int pile -> unit qui trie la pile prise en argument et ne renvoie rien.

**Exercice 5 (Vendredi 13)** : On dispose des informations suivantes :

- le 1<sup>er</sup> janvier 2024 était un lundi ;
- les années entre 2024 et 2099 sont bissextiles si et seulement si l'année est divisible par 4.

Écrire un programme OCaml calculant le nombre de vendredis 13 entre le 1<sup>er</sup> janvier 2024 et le 31 décembre 2099.

On pourra organiser ce programme en plusieurs fonctions, et on commentera le code pour clarifier ce qu'il fait si nécessaire.