OPTION INFORMATIQUE
Devoir surveillé 2

Toutes les fonctions sont à écrire en OCaml. On attachera le plus grand soin à leur lisibilité. On pourra toujours librement utiliser une fonction demandée à une question précédente, même si cette question n'a pas été traitée.

1 Polynômes

Dans cet exercice, on identifie un polynôme $P = a_n X^n + \ldots + a_0$ et sa représentation sous forme de liste de flottants `[`$a_0; \ldots; a_n$`]`.

1. Écrire une fonction `somme` calculant la somme de deux polynômes.
2. Écrire une fonction `produit_externe` calculant le produit d'un polynôme et d'un flottant.
3. Écrire une fonction `produit` calculant le produit de deux polynômes (on ne demande pas une version diviser pour régner).
4. Écrire une fonction `composee` prenant en argument deux polynômes p et q et renvoyant le polynôme $p \circ q$.
5. Écrire une fonction `primitive` prenant en argument un polynôme p et un flottant b et renvoyant le polynôme dont la dérivée est p et valant b en 0.

2 Minimum local dans un tableau

On considère un type `'a` et un tableau `t` de type `'a array` non vide. On note n la longueur de `t` (et donc $n \geq 1$). On dit qu'une valeur `t.(i)` est un **minimum local** de `t` si elle est supérieure à sa ou ses voisines. Formellement :

- `t.(0)` est un minimum local si `t.(0)` $\leq$ `t.(1)` (ou si $n = 1$),
- pour $i \in [\![1, n-2]\!]$, `t.(i)` est un minimum local si `t.(i)` $\leq$ `t.(i-1)` et `t.(i)` $\leq$ `t.(i+1)`,
- Si $n > 1$, `t.(n-1)` est un minimum local si `t.(n-1)` $\leq$ `t.(n-2)`.

On remarque que, un minimum global étant un minimum local, tout tableau non vide admet au moins un minimum local.

1. Écrire une fonction `indice_ming : 'a array -> int` prenant en argument un tableau `t` supposé non vide et renvoyant un indice du minimum **global** de `t`.
2. Écrire une fonction `premier_indice_minl : 'a array -> int` prenant en argument un tableau `t` supposé non vide et renvoyant le premier indice d'un minimum local de `t`.
3. Soit $k \in [\![1, n-2]\!]$ tel que `t.(k)` n'est pas un minimum local de `t`. Démontrer les propriétés suivantes, où `t[i:j]` désigne le sous-tableau de `t` de l'indice i inclus à l'indice j exclus :
   - Si `t.(k)` $\geq$ `t.(k-1)`, alors un minimum local de `t[0:k]` est un minimum local de `t`.
   - Si `t.(k)` $\geq$ `t.(k+1)`, alors un minimum local de `t[k+1:n]` est un minimum local de `t`.
4. En déduire une fonction `indice_minl : 'a array -> int` prenant en argument un tableau `t` supposé non vide, et renvoyant un indice réalisant un minimum local de `t`, en utilisant une stratégie *diviser pour régner* aussi efficace que possible.
5. Déterminer la complexité de la fonction précédente, en supposant que chaque comparaison d'élements de `t` est en $O(1)$.

3 Parcours d'arbre binaire

On considère le type d'arbres binaires :

```
type 'a arbre = V | N of 'a arbre * 'a * 'a arbre;;
```

1. Définir en Caml deux arbres ayant `[1; 2; 3]` comme noeuds visités dans l'ordre du parcours préfixe, et des parcours infixes différents. Préciser le parcours infixe de chaque arbre.
2. Écrire une fonction `parcours_postfixe : 'a arbre -> ('a -> unit) -> unit` prenant en argument un arbre et une procédure et appliquant cette procédure sur les noeuds dans l'ordre d'un parcours postfixe.
3. On souhaite écrire une version efficace d'un parcours en largeur, en exploitant une implémentation de la structure de file mutable qu'on suppose donnée, constituée des fonctions :
   - `creer_file : unit -> 'a file`
   - `enfiler : 'a file -> 'a -> unit`
   - `defiler : 'a file -> 'a`
   - `est_vide : 'a file -> bool`

   Écrire une fonction `parcours_largeur : 'a arbre -> ('a -> unit) -> unit` réalisant un parcours en largeur. La fonction créera et utilisera une `'a arbre file` à l'aide des fonctions listées.

4 Sélection de la valeur de rang k

Étant donné un tableau de longueur n, et $k \in [\![0, n-1]\!]$, on définit la **valeur de rang k** de ce tableau comme la valeur qui serait à l'indice k s'il était trié. Ainsi, la valeur de rang 0 est le minimum, la valeur de rang $n-1$ est le maximum, et la valeur de rang $\lfloor n/2 \rfloor$ est une médiane. On définit de la même façon la valeur de rang k d'une liste. L'objectif de cet exercice est d'implémenter plusieurs méthodes de calcul de la valeur de rang k, et de comparer leur complexité.

4.1 Par recherches successives de minimum

Le principe de la première méthode est de chercher k fois de suite un minimum.

1. On cherche à appliquer cette méthode sur une liste.
   (a) Écrire une fonction `extraire_min : 'a list -> 'a * 'a list` prenant une liste non vide et renvoyant son minimum ainsi que la liste formée du reste de ses éléments.
   (b) En déduire une fonction `rang_liste1 : 'a list -> int -> 'a` prenant en argument une liste non vide et un entier k et renvoyant sa valeur de rang k.
2. On applique maintenant la méthode sur un tableau, en place.
   (a) Écrire une fonction `echange : 'a array -> int -> int -> unit` échangeant deux éléments dans un tableau.
   (b) En déduire une fonction `rang_tableau1 : 'a array -> int -> 'a` en place.
3. Déterminer la complexité de cette méthode, en fonction de k et de n.

4.2 Par tri fusion

Le principe de la seconde méthode est de passer par un tri fusion

1. On applique la méthode dans le cas d'une liste.
   (a) Écrire une fonction `fusion_liste : 'a list -> 'a list -> 'a list` prenant en argument deux listes triées et renvoyant leur fusion triée.
   (b) Écrire une fonction `split : 'a list -> 'a list * 'a list` prenant en argument une liste l et renvoyant deux listes de longueurs proches se répartissant les éléments de l.
   (c) Écrire une fonction `tri_fusion_liste : 'a list -> 'a list` implémentant le tri fusion sur une liste.
   (d) En déduire la fonction `rang_liste2 : 'a list -> int -> 'a` (oui Armand, vous pouvez utiliser votre fonction préférée du module `List`).

2. On implémente à présent cette méthode sur un tableau (on demande une implémentation spécifique, qui ne convertit pas de tableau en liste)
   (a) Écrire une fonction `tranche : 'a array -> int -> int -> 'a array` prenant en argument un tableau t et deux indices a et b, et renvoyant le tableau formé des éléments de t entre les indices a inclus et b exclus.
   (b) Écrire une fonction `fusion_tableau : 'a array -> 'a array -> 'a array` fusionnant deux tableaux triés.
   (c) Écrire une fonction `tri_fusion_tableau : 'a array -> 'a array` implémentant le tri fusion sur un tableau.
   (d) En déduire la fonction `rang_tableau2 : 'a array -> int -> 'a`.
3. Déterminer la complexité de cette méthode.

4.3 Sélection rapide (Quick select)

Le principe de cette méthode suit celui du tri rapide : les éléments sont répartis selon leur comparaison à un élément pivot. À la différence du tri rapide, on ne fait ensuite qu'un seul appel récursif, sur celle de ces deux parties qui contient l'élément de rang k.

1. On implémente cette méthode sur les listes.
   (a) Écrire une fonction `repartition_liste : 'a list -> 'a -> 'a list * 'a list * int` prenant en argument une liste l et un pivot p et renvoyant le triplet l1, l2, n1, où l1 contient les éléments de l strictement inférieurs à p, l2 contient les éléments de l supérieurs ou égaux à p, et n1 est la longueur de l1.
   (b) En déduire une fonction `rang_liste3 : 'a list -> int -> 'a` implémentant la sélection rapide sur une liste.
2. On cherche à présent à implémenter cette méthode **en place** dans le cas d'un tableau.
   (a) Écrire une fonction `repartition_tableau : 'a array -> int -> int -> int` prenant en argument un tableau t, deux indices a et b, prenant comme pivot $p =$ `t.(b)`, et renvoyant un indice ip tout en permutant les éléments du tableau de façon à ce que p soit à l'indice ip, tous les éléments d'indice de a à $ip-1$ soient strictement inférieurs à p, et tous les éléments d'indice $ip+1$ à b soient supérieurs ou égaux à p.
   La fonction devra être en place, donc en particulier ne créera pas de nouveau tableau ou de nouvelle liste, et de complexité linéaire en $b-a$. On pourra commencer par initialiser ip à a, puis parcourir les indices i de a à $b-1$ en préservant l'invariant :
   $$p = \texttt{t.(b)} \text{ et } ip \leq i \text{ et } \forall k \in [\![a, ip-1]\!], \texttt{t.(k)} < p \text{ et } \forall k \in [\![ip, i-1]\!], \texttt{t.(k)} \geq p$$
   (b) En déduire une fonction `rang_tableau3 : 'a array -> int -> 'a` implémentant la sélection rapide en place sur un tableau.
3. Déterminer la complexité de cette méthode.

4.4 Médiane des médianes

Le principe de cette méthode est d'utiliser une sélection rapide dans laquelle on prend comme pivot la **médiane des médianes**. Cette médiane des médianes s'obtient en répartissant les éléments en $\lceil n/5 \rceil$ groupes de 5 éléments (éventuellement moins pour le dernier groupe), en calculant la médiane de chacun de ces groupes et en obtenant la médiane de ces médianes à l'aide d'un appel récursif.

1. Écrire une fonction `rang_liste4 : 'a list -> int -> 'a` implémentant cette méthode pour une liste.
2. Écrire une fonction `rang_tableau4 : 'a array -> int -> 'a` implémentant cette méthode pour un tableau. On ne demande pas une implémentation en place.

3. On se place dans le cas où n est une puissance de 10. Justifier qu'il y a au plus $7n/10$ éléments strictement supérieurs, et au plus $7n/10$ éléments strictement inférieurs, à la médiane des médiane.
4. On en déduit que dans le cas où tous les éléments sont distincts, la complexité de cette méthode vérifie la relation de récurrence

$$C(n) = C(n/5) + C(7n/10) + O(n)$$

   Montrer que cette relation entraine $C(n) = O(n)$.
5. Justifier que cette version de la sélection de la valeur de rang k permet d'améliorer la complexité asymptotique du tri rapide.