OPTION INFORMATIQUE
Devoir surveillé 2
Corrigé

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.
   Corrigé :

```
let rec somme p q =
    match p,q with
        [],_ -> q
        |_,[] -> p
        |a::p',b::q' -> (a+.b)::(somme p' q');;
```

2. Écrire une fonction `produit_externe` calculant le produit d'un polynôme et d'un flottant.
   Corrigé :

```
let rec produit_ext a p =
    match p with
        [] -> []
        |b::q -> a*.b ::(produit_ext a q);;
```

3. Écrire une fonction `produit` calculant le produit de deux polynômes (on ne demande pas une version diviser pour régner).
   Corrigé :

```
let rec produit p q =
    match p with
        [] -> []
        |a::r -> somme (produit_ext a q) (0.::(produit r q));;
```

4. Écrire une fonction `composee` prenant en argument deux polynômes p et q et renvoyant le polynôme $p \circ q$.
   Corrigé :

```
let rec composee p q =
    match p with
        [] -> []
        |a::r -> somme [a] (produit (composee r q) 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.
   Corrigé :

```
let primitive p b =
    let rec aux l k =
        match l with
            [] -> []
            |a::q -> (a /. k)::(aux q (k+.1.))
    in
    b::(aux p 1.);;
```

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`.
   Corrigé :

```
let indice_ming t =
    let n = Array.length t in
    let m = ref 0 in
    for i = 1 to (n-1) do
        if t.(i) < t.(!m) then m:= i
    done;
    !m;;
```

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`.
   Corrigé :

```
let premier_indice_minl t =
    let n = Array.length t in
    let m = ref 0 in
    while !m<(n-1) && t.(!m) > t.(!m+1) do
        incr m
    done;
    !m;;
```

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`.
     Corrigé :
     Soit m un minimum local de `t[0:k]` atteint sur l'indice i. Si $i < k-1$, m est clairement minimum local de t. Si $i = k-1$, on a donc `t.(k-1)` $\leq$ `t.(k-2)`. De plus par hypothèse, `t.(k-1)` $\leq$ `t.(k)`, et m est donc bien encore 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`.
     Corrigé : Soit m un minimum local de `t[k+1:n]` atteint sur l'indice i. Si $i > k+1$, m est clairement un mnimal local de t. Si $i = k+1$, on a donc `t.(k+1)` $\leq$ `t.(k+2)`. De plus par hypothèse, `t.(k+1)` $\leq$ `t.(k)`, et m est donc bien encore 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.
   Corrigé :

```
let indice_minl t =
    let n = Array.length t in
    let rec aux i j =
        (* renvoie un minimum local de t[i:j] *)
        if j-i = 1 then i else
        if j-i = 2 then (if t.(i+1) < t.(i) then i+1 else i) else
        let k = (i+j)/2 in
```

```
            match t.(k) <= t.(k-1), t.(k) <= t.(k+1) with
                true,true -> k
                |false,_ -> aux i (k-1)
                |_,false -> aux (k+1) j
        in
        aux 0 n;;
```

5. Déterminer la complexité de la fonction précédente, en supposant que chaque comparaison d'élements de `t` est en $O(1)$.

   Corrigé : En notant $n = j - i$, la complexité de la fonction aux vérifie une relation de récurrence de la forme $C(n) = C(n/2) + O(1)$. On a donc $C(n) = O(\log n)$, et la complexité de la fonction `indice_minl` est donc logarithmique en la taille du tableau.

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.

   Corrigé :

2. ```
   let l1 = N(N(V,2,V), 1, N(V,3,V));;
   let l2 = N(V, 1, N(V, 2, N(V, 3, V)));;
   ```

   `l1` a pour parcours infixe `[2; 1; 3]`, et `l2` a pour parcours postfixe `[1; 2; 3]`.

3. 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.

   Corrigé :

4. ```
   let rec parcours_postfixe a p =
       match a with
           V -> ()
           |N(g, x, d) -> parcours_postfixe g p; parcours_postfixe d p; p x
   ;;
   ```

5. 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.

   Corrigé :

   ```
   let rec parcours_largeur a p =
       let f = creer_file () in
       enfiler f a;
       while not (est_vide f) do
           match defiler f with
               V -> ()
               |N(g, x, d) -> p x; enfiler f g; enfiler f d
       done
   ;;
   ```

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.

   Corrigé :

```
let rec extraire_min l =
    match l with
        [] -> failwith "min d'une liste vide"
        |[a] -> a, []
        |a::q -> let b, r = extraire_min q in
                    if a < b then a, q
                    else b, a::r
;;
```

   (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.

   Corrigé :

```
let rec rang_liste1 l k =
    let a, q = extraire_min l in
    if k = 0 then a else rang_liste1 q (k-1)
;;
```

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.

```
let echange t i j =
    let a = t.(i) in
    t.(i) <- t.(j);
    t.(j) <- a
;;
```

   (b) En déduire une fonction `rang_tableau1 : 'a array -> int -> 'a` en place.

```
let rang_tableau1 t k =
    let t' = Array.copy t in
    let n = Array.length t in
    for i = 0 to k do
        let jmin = ref i in
        for j = (i+1) to n-1 do
            if t'.(j) < t'.(!jmin) then jmin := j
        done;
        echange t' i !jmin
    done;
    t'.(k)
;;
```

3. Déterminer la complexité de cette méthode, en fonction de k et de n.
   Corrigé : La méthode effectue k recherches de minimum, chacune en $O(n)$, d'où une complexité en $O(kn)$.

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.
   Corrigé :

```
let rec fusion_liste l1 l2 =
    match l1, l2 with
        [], l | l, [] -> l
        |a1::q1, a2::q2 -> if a1 < a2 then a1::(fusion_liste q1 l2) else a2::(fusion_li
;;
```

   (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.
   Corrigé :

```
let rec split l =
    match l with
        [] | [_] -> [], l
        |a::b::q -> let l1, l2 = split q in a::l1, b::l2
;;
```

   (c) Écrire une fonction `tri_fusion_liste : 'a list -> 'a list` implémentant le tri fusion sur une liste.
   Corrigé :

```
let rec tri_fusion_liste l =
    match l with
        [] | [_] -> l
        |_ ->
            let l1, l2 = split l in
            fusion_liste (tri_fusion_liste l1) (tri_fusion_liste l2)
;;
```

   (d) En déduire la fonction `rang_liste2 : 'a list -> int -> 'a` (oui Armand, vous pouvez utiliser votre fonction préférée du module `List`).
   Corrigé :

```
let rang_liste2 l k = List.nth (tri_fusion_liste l) k;;
```

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.
   Corrigé :

```
let tranche t a b =
    let t' = Array.make (b-a) t.(0) in
    for i = a to b-1 do
        t'.(i-a) <- t.(i)
    done;
    t'
;;
```

(b) Écrire une fonction `fusion_tableau : 'a array -> 'a array -> 'a array` fusionnant deux tableaux triés.

Corrigé :

```
let fusion_tableau t1 t2 =
    let n1 = Array.length t1 in
    let n2 = Array.length t2 in
    let t = Array.make (n1 + n2) t1.(0) in
    let i1 = ref 0 in
    let i2 = ref 0 in
    while !i1 < n1 && !i2 < n2 do
        if t1.(!i1) < t2.(!i2) then begin
            t.(!i1 + !i2) <- t1.(!i1);
            incr i1
        end
        else begin
            t.(!i1 + !i2) <- t2.(!i2);
            incr i2
        end
    done;
    for i = !i1 to n1 - 1 do
        t.(i + !i2) <- t1.(i)
    done;
    for i = !i2 to n2 - 1 do
        t.(i + !i1) <- t2.(i)
    done;
    t
;;
```

(c) Écrire une fonction `tri_fusion_tableau : 'a array -> 'a array` implémentant le tri fusion sur un tableau.

Corrigé :

```
let rec tri_fusion_tableau t =
    let n = Array.length t in
    if n <= 1 then t
    else begin
        let t1 = tranche t 0 (n/2) in
        let t2 = tranche t (n/2) n in
        fusion_tableau (tri_fusion_tableau t1) (tri_fusion_tableau t2)
    end
;;
```

(d) En déduire la fonction `rang_tableau2 : 'a array -> int -> 'a`.

Corrigé :

```
let rang_tableau2 t k = (tri_fusion t).(k);;
```

3. Déterminer la complexité de cette méthode.

Corrigé :

Le tri fusion est en $O(n \ln n)$. Dans le cas d'une liste, il y a ensuite un appel à `List.nth` qui est en $O(k)$, puisque $k \leq n$, la complexité totale reste un $O(n \ln n)$.

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.

Corrigé :

```
let rec repartition_liste l p =
    match l with
        [] -> [], [], 0
        |a::q -> let l1, l2, n = repartition_liste q p in
                    if a < p then a::l1, l2, n + 1
                    else l1, a::l2, n
;;
```

(b) En déduire une fonction `rang_liste3 : 'a list -> int -> 'a` implémentant la sélection rapide sur une liste.

Corrigé :

```
let rec rang_liste3 l k =
    match l with
        [] -> failwith "min d'une liste vide"
        |[a] -> a
        |a::q -> let l1, l2, n = repartition_liste q a in
                    if k<n then rang_liste3 l1 k
                    else if k=n then a
                    else rang_liste3 l2 (k - n - 1)
;;
```

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 :

$$p = \texttt{t.(b)} \text{ et } ip \leq i \text{ et } \forall k \in [\![a, ip - 1]\!], \texttt{t.(k)} < p \text{ et } \forall k \in [\![ip, i - 1]\!], \texttt{t.(k)} \geq p$$

Corrigé :

Corrigé :

```
let repartition_tableau t a b =
    let p = t.(b) in
    let ip = ref a in
    for i = a to (b-1) do
        if t.(i) < p then begin
            echange t i !ip;
            incr ip
        end
    done;
    echange t b !ip;
    !ip;;
;;
```

(b) En déduire une fonction `rang_tableau3 : 'a array -> int -> 'a` implémentant la sélection rapide en place sur un tableau.

Corrigé :

```
let rang_tableau3 t k =
    let rec aux a b =
        if a = b then t.(a) else begin
            let ip = repartition_tableau t a b in
            if k < ip then aux a (ip - 1)
            else if k = ip then t.(ip)
            else aux (ip + 1) b
        end
    in
    aux 0 (Array.length t - 1)
;;
```

3. Déterminer la complexité de cette méthode.
   Corrigé :
   La complexité de `rang_tableau3` vérifie dans le pire cas (par exemple k = 0 et où la liste est triée) la relation de récurrence

$$C(n) = C(n-1) + O(n)$$

   d'où $C(n) = O(n^2)$.

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.
   Corrigé :

```
let rec supprimer l x =
        match l with
            [] -> failwith "element non present"
            |a::q -> if a = x then q else a::(supprimer q x)
;;

let rec rang_liste4 l k =
    match l with
        [] -> failwith "min d'une liste vide"
        |[a] -> a
        |_ ->
            let rec medianes l =
                match l with
                    [] -> []
                    |a::b::c::d::e::q -> (rang_liste1 [a;b;c;d;e] 2)::(medianes q)
                    |_ -> [rang_liste1 l (List.length l / 2)]
            in
            let m = medianes l in
            let n = List.length m in
            let p = rang_liste4 m (n/2) in
            let l1, l2, n = repartition_liste (supprimer l p) p in
            if k<n then rang_liste4 l1 k
            else if k=n then p
            else rang_liste4 l2 (k - n - 1)
;;
```

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

```
let mediane t a b =
    let n = Array.length t in
    let b' = min b (n-1) in
    for i = (a+1) to b' do
        let j = ref i in
        while !j > a && t.(!j - 1) > t.(!j) do
            echange t !j (!j - 1);
            decr j
        done
    done;
    let k = (a+b')/2 in
    t.(k);;


let rec rang_tableau4 t k =
    let rec aux a b =
        if a = b then t.(a) else begin
            let m = (4 + (b - a + 1)) / 5 in
            let medianes = Array.make m t.(0) in
            for i = 0 to (m-2) do
                medianes.(i) <- mediane t (a + 5*i) (a+ 5*(i+1) - 1)
            done;
            medianes.(m-1) <- mediane t (a + 5*(m-1)) b;
            let p = rang_tableau4 medianes (m/2) in
            let i = ref a in
            while t.(!i) <> p do incr i done;
            echange t !i b;
            let ip = repartition_tableau t a b in
            if k < ip then aux a (ip - 1)
            else if k = ip then t.(ip)
            else aux (ip + 1) b
        end
    in
    aux 0 (Array.length t - 1)
;;
```

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.
   Corrigé :
   Soit m la médiane des médianes. Il y a $n/5$ groupes de 5, par définition la médiane de la moitié d'entre eux, donc $n/10$, est inférieure ou égale à m. Pour chacun de ces $n/10$ groupes de 5, il y a trois éléments qui sont inférieurs ou égaux à la médiane de ce groupe. On a donc identifié au moins $3n/10$ éléments inférieurs ou égaux à m, donc il y a au plus $7n/10$ éléments strictement supérieurs à m.
   Par un raisonnement symétrique, il y a au plus $7n/10$ éléments strictement inférieurs à m.
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 $C(n) = O(n)$.

Corrigé :

La relation de récurrence indique qu'il existe $c \in \mathbb{R}$ tel qu'à partir d'un certain rang n_0, $C(n) \leq C(n/5) + C(7n/10) + cn$.

On pose $c' = \max(C(1), \ldots, C(n_0), c)$ de sorte que $\forall n \in \mathbb{N}^*, C(n) \leq C(n/5)+C(7n/10)+c'n$.

On montre alors par récurrence forte sur $n \in \mathbb{N}^*$: $C(n) \leq 10c'n$ (on trouve ce facteur 10 par analyse).

L'initialisation est immédiate (puisque $c' \geq C(1)$.

Si la propriété est vraie jusqu'au rang $n-1$, alors

$$C(n) \leq C(n/5) + C(7n/10) + c'n \leq 10c'n/5 + 10c'7n/10 + c'n = 10nc'$$

ce qui conclut la récurrence.

5. Justifier que cette version de la sélection de la valeur de rang k permet d'améliorer la complexité asymptotique du tri rapide.

Corrigé :

En utilisant cette version pour calculer une médiane en $O(n)$, on obtient une version du tri rapide dont la complexité dans le pire cas vérifie la relation de récurrence

$$C(n) = 2C(n/2) + O(n)$$

d'où $C(n) = O(n \ln n)$ (au lieu de $O(n^2)$).

Cette version n'est pas vraiment utilisée car le pire cas du tri rapide est rarement atteint (au moins si le pivot est tiré au hasard) et le coût du calcul de la médiane augmente la complexité moyenne.