Devoir en temps libre n° 1 : corrigé

I Exercices CAML à faire sur papier

EXERCICE 2 *Existence d'un élément satisfaisant à un prédicat*

Il suffit de parcourir tous les éléments de la liste, de manière récursive, et de s'arrêter si un élément satisfait au prédicat. Le cas de la liste vide est atteint si aucun élément ne convient (ou si la liste est vide). Le caractère paresseux du `||` CAML nous assure que l'appel récursif n'est effectué qu'en cas de besoin.

```
let rec existe predicat liste =
  match liste with
  | [] -> false
  | tete :: queue ->
      predicat tete || existe predicat queue
;;
```

Cette fonction est de type `('a -> bool) -> 'a list -> bool` et de complexité linéaire en la longueur de la liste. Le pire des cas est atteint lorsqu'aucun élément ne satisfait au prédicat.

EXERCICE 3 *Minimum et maximum d'une liste*

Si la liste est vide, on lève l'exception comme demandé. Si elle a un seul élément, cet élément est à la fois le maximum et le minimum. Sinon, le minimum de la liste est aussi le minimum entre la tête et le minimum de la queue ; et de même pour le maximum. Les appels récursifs s'achèvent bien sur le cas de base à un élément et non sur celui de la liste vide (qui renvoie une erreur).

```
let rec min_et_max liste =
  match liste with
  | [] -> failwith "Liste vide"
  | tete :: [] -> (tete, tete)
  | tete :: queue ->
     let (mini, maxi) = min_et_max queue in
     (min tete mini, max tete maxi)
;;
```

Cette fonction est de type `'a list -> ('a * 'a)` et de complexité linéaire en la longueur de la liste, puisqu'il y a autant d'appels récursifs que d'éléments dans la liste.

EXERCICE 4 *Supprimer l'élément d'indice n d'une liste*

Si la liste est vide il n'y a rien à faire. Sinon, si l'élément à supprimer est celui d'indice 0, on renvoie la queue. Sinon enfin, on supprime l'élément d'indice $(n-1)$ de la queue (s'il existe), récursivement, et la liste recherchée est obtenue en y ajoutant la tête.

```
let rec supprime n liste =
  match liste with
  | [] -> []
  | tete :: queue when n = 0 -> queue
  | tete :: queue -> tete :: (supprime (n - 1) queue)
;;
```

La fonction est de type `int -> 'a list -> 'a list`. Si la liste est de longueur ℓ, il y a au plus $\min(|n|, \ell)$ appels récursifs, avec ℓ appels si $n >= \ell$ ou si n est négatif, ce qui est le cas si on considère le pire des cas. La complexité est donc en $\Theta(\ell)$, linéaire en la longueur de la liste (et pour la complexité dans le pire des cas, indépendante de n).

Remarque 1. *On ne peut pas modifier ni supprimer les éléments d'une liste en* CAML *car les listes ne sont pas mutables (modifiables). Mais on peut renvoyer une « nouvelle » liste obtenue à partir de la première. Au premier abord, ceci peut sembler inefficace, mais il n'y a pas nécessairement de copie créée, car il peut y avoir un partage en mémoire, du fait justement que les listes ne peuvent jamais être modifiées.*

EXERCICE 5 *X 2010*

On cherche l'indice du bit de poids fort de n que l'on obtient en divisant n par 2 autant de fois que possible.

```
let rec log2 = function
  | 1 -> 0
  | n -> 1 + log2 (n / 2)
;;
```

On obtient bien sûr une fonction de type `int -> int`. On peut supposer ici que la division par deux (qui est d'ailleurs un simple décalage de bits) est une opération élémentaire. Il a k appels récursifs. Si comme taille de l'entrée on considère l'entier n lui-même, alors la complexité est logarithmique en $\Theta(\log n)$. Si on prend comme taille de l'entrée le nombre de chiffres k de n en base 2, ce qui correspond à la place occupée par n en mémoire, alors la complexité est linéaire en $\Theta(k)$.

Voici un extrait du rapport de Polytechnique 2010 où cet exercice était posé : *il s'agissait d'une question très facile à laquelle les candidats ont répondu de manière décevante, en oubliant notamment de vérifier que* $\log_2(1) = 0$, *ce qui était sévèrement sanctionné.*

II Exercices CAML à faire sur machine

EXERCICE 6 *Réduction d'éléments consécutifs égaux*

L'idée est la suivante : on commence par réduire récursivement la queue, puis on compare son premier élément à la tête de la liste. Si ils sont égaux on ne conserve que la queue réduite, sinon on concatène la tête à la queue réduite. Le programme suivant implémente cette idée, en n'oubliant pas le cas de base (une liste à un élément) et le cas de la liste vide.

```
let rec reduction liste =
  match liste with
  | [] -> []
  | tete :: [] -> [tete]
  | tete1 :: tete2 :: queue when tete1 = tete2 ->
     reduction (tete2 :: queue)
  | tete :: queue ->
     tete :: (reduction queue)
;;
```

Le deuxième cas n'est pas nécessaire car il est couvert par le premier et le dernier, mais la fonction me semble plus explicite ainsi. La complexité est linéaire en la taille de la liste car les appels récursifs se font toujours sur la queue de la liste en argument.

EXERCICE 7 *Aplatissement d'une liste de listes*

Pour aplatir une liste de listes, on extrait successivement tous les éléments de la liste qui se trouve en tête et on les ajoute au résultat, obtenu par un appel récursif sur ce qui reste. Attention, les deux « queues » n'ont pas le même statut : `queue1` est la queue de la liste en tête (donc une `'a list`), alors que `queue2` est la suite des autres listes (donc une `'a list list`. Dans le dernier cas, il s'agit donc bien de `queue1 :: queue2` (et non de quelque chose comme `queue1 @ queue2`). Dans cet appel récursif on traite toute la liste de listes sauf le tout premier élément de la toute première liste.

```
let rec aplatir liste =
  match liste with
  | [] -> []
  | [] :: queue -> aplatir queue
  | (tete1 :: queue1) :: queue2 ->
      tete1 :: aplatir (queue1 :: queue2)
;;
```

Si on note ℓ_i la longueur de la i-ème liste, il y a $\ell_i + 1$ appels récursifs pour chaque liste, et on trouve une complexité en $\Theta(\sum_{i=1}^{n} \ell_i)$ s'il y a n listes dans la liste de listes. La complexité est donc linéaire en la taille de la liste en entrée, si l'on prend comme taille de la liste le nombre total d'éléments (et non sa longueur).

EXERCICE 8 *Plus long plateau d'une liste (facultatif)*

L'idée est de généraliser la recherche d'un élément maximal dans une liste, où l'on recherche l'élément maximal de la queue, que l'on met ensuite éventuellement à jour après comparaison avec la tête. On procède de même en recherchant récursivement le plus long plateau. Pour pouvoir comparer le plus long plateau de la queue avec celui qui commencerait par l'élément en tête, il faut savoir quelle est la longueur du plateau en tête. On commence donc par écrire une fonction auxiliaire `plateau_aux : 'a list -> int * (int * int * 'a)` qui renvoie le couple formé par (a) la longueur du plateau commençant par le premier élément ; (b) le triplet (debut, longueur, element) du plus long plateau de la liste. Le cas de base avec un élément vient immédiatement (le cas d'une liste vide renvoie une erreur). Ensuite, on appelle récursivement la fonction sur la queue de la liste et on distingue le cas où il y a rupture de plateau de celui où le premier élément permet de continuer le plateau en tête. Dans ce dernier cas, si le plateau en tête est plus long que le plus long trouvé précédemment, on effectue cette mise à jour. Remarquons enfin que l'indice de début doit être incrémenté lors de chaque appel récursif.

Une fois cette fonction auxiliaire écrite, on en déduit immédiatement la fonction principale en renvoyant uniquement le deuxième élément du couple.

La complexité de `plateau` est celle de `plateau_aux` qui est linéaire en la taille de la liste, puisque l'on a encore et toujours un seul et simple appel récursif sur la queue de la liste et que le reste est en $\Theta(1)$.

```
let rec plateau_aux liste =
  match liste with
  | [] -> failwith "Arguments invalides"
  | [elem] -> (1, (0, 1, elem))
  | tete1 :: tete2 :: queue ->
     let long_tete, (deb_max, long_max, elem_max) =
       plateau_aux (tete2 :: queue)
     in
     if tete1 = tete2 then
     (* Le premier élément continue le plateau *)
       if long_tete >= long_max then
       (* Ce plateau va devenir le plus long *)
         (long_tete + 1, (0, long_tete + 1, tete1))
       else
       (* Simple mise à jour *)
         (long_tete + 1, (deb_max + 1, long_max, elem_max))
     else
     (* On commence un nouveau plateau *)
       (1, (deb_max + 1, long_max, elem_max))
;;
```

```
let plateau liste =
 let _, res = plateau_aux liste in res
;;
```

PROBLÈME 1 (*Représentation des polynômes*)

1. On traduit en CAML la formule récursive indiquée par l'énoncé : on évalue le polynôme `queue` en `x`, on multiplie par `x` et on ajoute le coefficient à cela.

   ```
   let rec evaluation polynome x =
     match polynome with
     | [] -> 0.
     | coeff :: queue -> coeff +. x *. evaluation queue x
   ;;
   ```

   La complexité est en $\Theta(n)$ où n est la longueur de la liste représentant le polynôme (*i.e.* son degré plus un, si le polynôme est non nul et si la représentation est propre).

2. On parcourt la liste pour multiplier chaque coefficient. La complexité est linéaire (indépendante de λ).

   ```
   let rec produit_externe polynome lambda =
     match polynome with
     | [] -> []
     | coeff :: queue ->
        (lambda *. coeff) :: produit_externe queue lambda
    ;;
   ```

3. Il suffit de faire la somme des deux listes. Si l'une des deux est vide on renvoie l'autre, sinon on ajoute les deux têtes que l'on concatène à la liste somme renvoyée par l'appel récursif. Si les longueurs des listes sont ℓ_1 et ℓ_2, le nombre d'appels et donc ici la complexité est en $\Theta(\min(\ell_1, \ell_2)) = O(\ell_1 + \ell_2)$.

   ```
   let rec somme polynome1 polynome2 =
     match polynome1, polynome2 with
     | [], _ -> polynome2
     | _, [] -> polynome1
     | tete1 :: queue1, tete2 :: queue2 ->
        (tete1 +. tete2) :: somme queue1 queue2
   ;;
   ```

4. Multiplier par X revient à ajouter un 0 en tête de liste et diviser par X (si $a_0 = 0$) revient à prendre la queue de la liste. On traduit en CAML la relation récursive :

$$\left(\sum_{k=0}^{n} a_k X^k\right) Q = a_0 Q + X \left(\sum_{k=0}^{n-1} a_{k+1} X^k\right) Q$$

   ```
   let rec produit polynome1 polynome2 =
     match polynome1 with
     | [] -> []
     | coeff :: queue ->
        somme (produit_externe polynome2 coeff)
              (0. :: produit queue polynome2)
   ;;
   ```

   Si ℓ_P et ℓ_Q sont les longueurs des listes représentant P et Q, on note $C(\ell_P, \ell_Q)$ le coût dans le pire des cas. Il y a ℓ_P appels récursifs en tout *mais* on ne peut pas

conclure que la complexité est en $\Theta(\ell_P)$. En effet, dans chaque appel, on utilise la fonction `produit_externe` avec Q qui est de complexité $\Theta(\ell_Q)$ et la fonction `somme` avec une liste de longueur ℓ_Q et une autre de longueur $O(\ell_P + \ell_Q)$. On a donc $C(\ell_P, \ell_Q) = C(\ell_P - 1, \ell_Q) + O(\ell_Q)$ ce qui, après télescopage, donne une complexité en $O(\ell_Q \ell_P)$.

5. Il suffit de traduire la relation $P^k = PP^{k-1}$ et $P^0 = 1$ en CAML. On trouve une complexité en $O(k^2\ell_P^2)$.

```
let rec puissance polynome k =
  match k with
  | 0 -> [1.]
  | _ -> produit polynome (puissance polynome (k - 1))
 ;;
```

6. On traduit en CAML la relation récursive :

$$P \circ Q = \left(\sum_{k=0}^{n} a_k Q^k\right) = a_0 + Q\left(\sum_{k=0}^{n-1} a_{k+1} Q^k\right)$$

```
let rec compose polynome1 polynome2 =
  match polynome1 with
  | [] -> []
  | coeff :: queue ->
     somme [coeff]
       (produit polynome2 (compose queue polynome2))
;;
```

On reprend les notations précédentes. Il y a encore ℓ_P appels récursifs. La somme se fait en coût constant car la première liste est toujours de taille 1. On appelle la fonction `produit` sur une liste de taille ℓ_Q et une liste de taille $O(\ell_Q \ell_P)$. L'équation de récurrence est donc $C(\ell_P, \ell_Q) = C(\ell_P - 1, \ell_Q) + O(\ell_Q^2 \ell_P)$ ce qui, après télescopage, donne une complexité en $O(\ell_Q^2 \ell_P^2)$.

7. Il faut supprimer les 0 en fin de liste ce qui n'est pas aisé. En revanche supprimer les 0 en tête de liste est très simple, ce que réalise la fonction auxiliaire `nettoyage_tete : float list -> float list`. Il suffit de renverser la liste, d'enlever les 0 en tête et de renverser à nouveau. Pour que cela soit efficace, puisque le renversement a un coût linéaire, il faut faire cette manipulation une seule fois, au début et à la fin, et non pas pour chaque 0, évidemment.

```
let nettoyage polynome =
  let rec nettoyage_tete polynome =
    match polynome with
    | [] -> []
    | 0. :: queue -> nettoyage_tete queue
    | _ -> polynome
  in
  rev (nettoyage_tete (rev polynome))
;;
```

La complexité de `nettoyage_tete` est linéaire, dans le pire des cas, le pire des cas étant atteint quand la liste ne comporte que des 0. Comme la fonction `rev` est également de complexité linéaire, la fonction `nettoyage` est également linéaire.

8. Il suffit de renvoyer le dernier élément d'une liste, après avoir bien pris soin d'avoir une représentation propre du polynôme.

```
let coefficient_dominant polynome =
  let rec dernier_element polynome =
    match polynome with
    | [] -> failwith "Polynôme nul"
    | coeff :: [] -> coeff
    | coeff :: queue -> dernier_element queue
  in
  dernier_element (nettoyage polynome)
;;
```

La complexité est bien sûr linéaire ici.