Éléments de correction du devoir surveillé n° 2.

I Exercices d'échauffement

I.1

1. Analyse : comme on travaille avec des listes une programmation récursive est à privilégier. Le cas de base naturel est celui où l'on reçoit une liste vide qui correspond évidemment à la compression de la liste vide.

   Il est facile de voir que si l'on reçoit une liste dont le premier élément est le couple (n, e), il suffit de fabriquer une liste dont le premier élément est e, et dont la queue est la décompression de la même liste que celle reçue en paramètre, mais dont le premier terme est le couple $(n-1, e)$. En revanche il y a un piège... Car si on ne fait rien on ne passera jamais sur le cas de base naturel ! Il faut donc rajouter un cas de base supplémentaire correspondant à un premier terme de la forme $(0, e)$. Dans ce cas on appellera la fonction récursivement sur la queue de la liste reçue en paramètre.

   D'où le code suivant :

```
let rec decompression (lc : (int * 'a) list) : ('a list) =
  match lc with
  | [] -> []
  | (0, e) :: q -> decompression q
  | (n, e) :: q -> e :: decompression ((n - 1, e) :: q);;
```

2. Analyse : on va bien sûr travailler de manière récursive comme dans la question précédente.

   Les cas de bases sont évidement : liste vide, codée par la liste vide, et liste à un seul élément e, codée par la liste [(1,e)].

   Le cas récursive est un peu plus délicat. Il est clair que pour une liste e la forme $e :: q$ il faudra appeler la fonction sur la queue. La difficulté est qu'il faut analyser le résultat de cette compression de la queue q, pour savoir si e est identique au premier élément apparaissant dans la tête de la liste compressée (auquel cas il est de la forme (n, e) et il faut le remplacer par $(n+1, e)$) ou s'il est différent auquel cas il suffit de rajouter $(1, e)$ en tête de la liste compression de la queue q !

   Ce qui donne par exemple le code suivant :

```

let rec compression (liste : 'a list) : (int * 'a) list =
  match liste with
  | [] -> []
  | [e] -> [(1,e)]
  | e :: q -> let qc = compression q in
              let (n, f) = List.hd qc in
              if e = f then (n + 1, f) :: List.tl qc
                       else (1, e) :: qc ;;
```

I.2 Mapping itératif sur un tableau

Analyse : Si on note r_i le résultat de $op(op(op(op(...a), v_0), v_1), \cdots, v_i)$ alors on a clairement $r_{i+1} = op(r_i, v_{i+1})$. Il suffit donc de parcourir le tableau en maintenant à jour une référence initialisée avec a, et de rendre en fin de parcours le contenu de cette référence.

Une difficultée est liée au fait que l'énoncé parle d'opérateur, alors qu'on ne peut passer en paramètre qu'une fonction. Il faudra être vigilant sur ce point dans les deux fonctions **sumv** et **prodv**...

On peut alors proposer le code suivant :

```
let foldopv (op : 'a -> 'b -> 'a) (a : 'a) (tab : 'b array) : 'a =
  let res = ref a in
  let n = Array.length tab in
  for i = 0 to n - 1 do
    res := op !res tab.(i)
  done ;
  !res ;;
```

Fonctions sumv et prodv Analyse : il s'agit juste de trouver le bon "opérateur", i.e. la bonne fonction de deux variables, ainsi la bonne valeur initiale. Pour la somme, comme on veut en fait $r_{i+1} = r_i + v_{i+1}$, il faut bien sûr passer un "opérateur" de sommation. Pour que r_0 ait la bonne valeur il faut nécessairement prendre $a = 0$. On obtient donc l'appel suivant, avec par exemple utilisation d'une fonction anonyme de sommation de deux entiers :

```
let sumv (tab : int array) : int = foldopv (fun x y -> x + y) 0 tab
    ;;
```

Pour le produit il vient facilement :

```
let prodv (tab : int array) : int = foldopv (fun x y -> x * y) 1
   tab ;;
```

On peut également en utilisant les fonctions partielles écrire (notez la différence de typage...) :

```
let sumv : int array -> int = foldopv (fun x y -> x + y) 0 ;;
let prodv : int array -> int = foldopv (fun x y -> x * y) 1 ;;
```

Pour les geeks.... Il se trouve qu'on peut transformer un opérateur infixe, i.e. qu'on utilise sous la forme **a op b** en opérateur préfixe, i.e. qu'on utilise sous la forme **op a b**, en l'entourant entre parenthèses ! On peut donc de manière encore plus synthétique écrire :

```
let sumv : int array -> int = foldopv (+) 0 ;;
```

Mais cela ne convient pas pour la multiplication, car **(*)** est interprété comme le début d'un commentaire OCaml !

I.3 Conversion entre tableaux et listes

1. Comme la fonction reçoit une liste on va utiliser une programmation récursive. La difficulté vient du fait qu'il va falloir créer le tableau que l'on va rendre dès le premier appel (il faudra alors récupérer la taille de la liste, créer le tableau de la bonne taille, initialisé avec le premier élément de la liste (s'il y en a un)) et ensuite remplir le tableau par appel d'une fonction auxiliaire récursive (sur les queues successives de la liste initiale). Cela nécessite de passer en plus l'indice de la case dans laquelle placer les têtes successives des listes reçus en paramètre !

   On propose le code suivant :

```
let list_to_array (l : 'a list) : 'a array =
  let n = List.length l in
  if n = 0 then [||]
  else if n = 1 then [|List.hd l|]
  else let rep = Array.make n (List.hd l) in
  let rec list_to_array_aux l i =
    match l with
    | [] -> rep
    | t :: q -> rep.(i) <- t; list_to_array_aux q (i + 1)
    in list_to_array_aux l 0;;
```

2. Compte tenu de la contrainte qui interdit d'utiliser une référence pour manipuler une liste on est obligé de travailler récursivement.

   On utilise donc une fonction auxiliaire récursive qui prend en argument un entier donnant l'indice à partir duquel il faut convertir le tableau (et qui par ailleurs indique l'indice de l'élément que l'on doit insérer) et la liste dans laquelle l'insérer (en tête).

   Il suffit de faire croître cet indice à partir de 0. Quand on arrive à n c'est qu'on a fini et on rend la liste vide. Sinon il suffit de mettre l'élement t.(i) en tête du résultat de la conversion de la fin du tableau en liste.

   On peut donc proposer le code suivant :

```
let array_to_list (t : 'a array) : 'a list =
  let n = Array.length t in
  let rec array_to_list_aux i =
    if i >= n then []
    else t.(i) :: array_to_list_aux (i + 1)
    in array_to_list_aux 0 ;;
```

   Il faut au minimum manipuler chaque valeur pour la transférer du tableau vers la liste. La meilleure complexité possible semble donc être linéaire. Cette fonction est bien de meilleure complexité possible. En effet dans la fonction récursive il n'y a que des opérations élémentaires en temps constant. Et il y a n appels à cette fonction (pour les valeurs de l'indice de 0 à n). La fonction est donc bien de complexité linéaire.

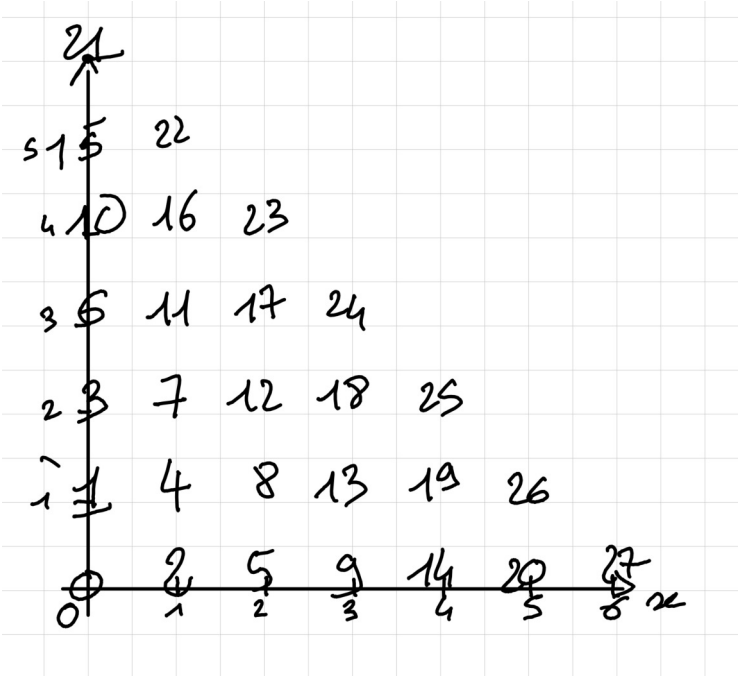
II Fonction triangle

1. Aucune difficulté...

```
let triangle (x : int) (y : int) : int =
  (x + y ) * (x + y + 1) / 2 + x;;
```

2. Pas de difficulté non plus.

   On constate que l'on range les entiers successifs dans le plan selon des droites de pente -1, ce qui fait apparaître un triangle à l'intérieur duquel on trouve ces entiers, d'où le nom.



3. 3-a On vérifie facilement par le calcul que $triangle(m,n) = triangle(m-1,n+1)+1$ (il suffit de substituer).

   3-b On a directement $triangle(0,n) = \dfrac{n(n+1)}{2}$.

   3-c Dès lors la programmation se fait facilement selon :

```
let rec triangle_rec (x : int) (y : int) : int =
  match x with
  | 0 -> y * (y + 1) / 2
  | _ -> triangle (x - 1) (y + 1) + 1;;
```

4. La relation de récurrence se fait sur deux paramètres dont le premier décroît strictement mais l'autre croît strictement à chaque appel... Il est donc raisonnable de penser à travailler sur l'ensemble $E = \mathbb{N}^2$ muni de l'ordre lexicographique $\preccurlyeq_L$. On sait que E est un ensemble ordonné bien fondé.

   L'ensemble des éléments minimaux est alors réduit au singleton $\{(0,0)\}$.

   L'ensemble des cas de base est $\mathcal{B} = \{(0,n), n \in \mathbb{N}\}$. Il inclut bien l'ensemble des éléments minimaux.

   Cas de base : pour $(m,n) \in \mathcal{B}$, la fonction termine car il y a juste un calcul et pas d'appels à d'autres fonctions.

   Induction : soit $(m,n) \in E$ quelconque, avec $(m,n) \notin \mathcal{B}$. Supposons que $\forall (p,q) \in E, (p,q) \prec_L (m,n)$ la fonction termine. L'appel `triangle_rec(m,n)` déclenche l'appel de `triangle_rec (p,q)` avec $p = m-1$ et $q = n+1$. Or $(p,q) \prec_L (m,n)$ et par hypothèse la fonction termine avec les paramètres $(p = m-1, q = n+1)$. La fonction termine donc pour (m,n).

   Conclusion : par induction bien fondée on en conclut que la fonction `triangle_rec` termine pour tout $(m,n) \in E$.

5. On voit facilement que les appels successifs de la fonction se font avec les paramètres (m,n), $(m-1,n+1)$, $(m-2,n+2)$, jusqu'à $(0,m+n)$. La complexité est donc linéaire en $m+n$, soit $\mathcal{O}(m+n)$.

III Algorithme de sélection utilisant la médiane des médianes

III.1 Généralités

1. D'après la définition de l'énoncé la recherche du plus petit élément se fait par `selection tab 1` et celle du plus grand élément par `selection tab n` où n est la taille du tableau
2. Dans le cas où n est impair il n'y a pas d'ambiguïté sur ce que l'on va appeler la médiane. En écrivant $n = 2p+1$, on voit que la médiane est plus grande que p éléments exactement, c'est donc le $p+1$ -ième plus petit élément. Et on a bien $p+1 = \lfloor \frac{n+1}{2} \rfloor$.

   Si n est pair en revanche, on pourrait choisir le p -ième plus petit élément (médiane inférieure), ou le $p+1$ -ième plus petit élément (médiane supérieure). Or ici, si $n = 2p$, $\lfloor \frac{n+1}{2} \rfloor = p$. On fait donc bien le choix de la médiane inférieure.
3. Une idée possible est de proposer de trier, par exemple avec le tri fusion le tableau en complexité $\mathcal{O}(n \ln n)$, puis d'aller chercher l'élément d'incide $i-1$ dans le tableau, ce qui se fait en temps constant. La complexité serait alors bien quasi-linéaire.

   Pour la preuve que le tri fusion est de complexité quasi linéaire (ce qu'il faut établir ici), cf. cours dans le chapitre sur la méthode diviser pour régner.

III.2 Algorithme optimal utilisant la médiane des médianes

1. Analyse : la seule adaptation consiste à créer d'abord une copie du tableau à trier, ce qui peut se faire soit à la main, soit en utilisant la fonction **sub** rappelé dans l'énoncé (on extrait en fait la totalité du tableau !)

   Le code n'est alors que la traduction directe de l'algorithme donné dans l'énoncé, en travaillant sur la copie plutôt que sur le tableau reçu. Il y a moyen de faire autrement d'ailleurs puisqu'on a aussi l'original...

```
let tri_insertion (tab : 'a array) : 'a array =
  let n = Array.length tab in
  let rep = Array.sub tab 0 n in
  for i = 1 to n - 1 do
    let x = rep.(i) in
    let j = ref(i) in
    while !j > 0 && rep.(!j - 1) > x do
      rep.(!j) <- rep.(!j - 1) ;
      decr j
    done ;
    rep.(!j) <- x
  done ;
  rep ;;
```

2. Dans notre algorithme la taille des tableaux qui seront passés en paramètre de cette fonction est majorée par 5. Dès lors le coût sera majorée par une certaine constante. Il sagit d'une complexité en $\mathcal{O}(1)$.

III.3 Récursion : première étape

III.3.1 Q6 Détails des calculs

Le nombre de sous-tableaux à 5 éléments est clairement $\lfloor \frac{n}{5} \rfloor$, qu'on calculera par la division entière $n/5$. Pour savoir s'il y aura un reliquat on calcule le reste de la disvion entière de n par 5, par **n mod 5**, que l'on note r. Si $r = 0$ il n'y a pas de reliquat, sinon il y a un sous-tableaux supplémentaire.

On peut donc ainsi calculer le nombre de sous-tableaux à créer, et on les créera vides grâce à l'indication de l'énoncé.

Ensuite on extrait grâce à la fonction **sub** des tableaux de taille 5, à partir des indices $5 * i$ pour i variant entre 0 et **n / 5 - 1**.

Enfin si nécessaire on extrait le reliquat qu'on mettra dans la dernière case du tableau.

III.3.2 Q7 Fonction decoupage

On peut alors proposer le code suivant :

```
let decoupage (tab : 'a array) : 'a array array =
ble  let n = Array.length tab in
  let m = n / 5 and r = n mod 5 in (* m est le nombre de tableaux à 5 éléments,
 on évalue avec r s'il y a un reliquat *)
  let nst = m + if r > 0 then 1 else 0 in (* nst est le nombre de sous-tableaux *)
  let res = Array.make nst [||] in
  begin
    for i = 0 to m - 1 do (* on traite ici les sous-tableaux à 5 éléments *)
      let tab_int = Array.sub tab (i * 5) 5 in
      res.(i) <- tab_int
    done ;
    if r > 0 then (* s'il y a du reliquat, on récupère les éléments correspondants *)
      let tab_int = Array.sub tab (m * 5) r in
      res.(nst - 1) <- tab_int
  end ;
  res ;;
```

III.3.3 Q8 Complexité

Finalement on ne fait que parcourir une seule fois tout le tableau pour répartir les éléments dans des sous-tableaux. La création des sous-tableaux est en $O(n)$ puisque la somme des tailles des sous-tableaux est égale à celle du tableau initial. Le parcours et le transfert des éléments se fait également en $O(n)$. On a donc une complexité linéaire.

III.4 Récursion : deuxième étape

III.4.1 Q9 Fonction tableau_médiane

Analyse : il suffit de parcourir le tableau des sous-tableaux, pour chaque sous-tableau d'appeler récursivement la fonction **selection** pour calculer la mediane en utilisant la formule de la question Q2, et de mettre le résultat dans un tableau qu'on aura créé de même taille que celle du tableau de tableaux.

Ce qui donne le code suivant :

```
let tableau_medianes (tab : 'a array array) : 'a array =
  let nst = Array.length tab in
  let rep = Array.make nst tab.(0).(0) in
  for i = 0 to nst - 1 do
    let n = Array.length tab.(i) in
    rep.(i) <- selection tab.(i) ((n + 1) / 2)
  done ;
  rep ;;
```

Le petit problème c'est que la fonction **selection** n'a pas encore été écrite... Il faudrait attendre donc l'écriture finale de cette fonction pour l'y définir par une liaison locale... Ou alors on définira deux fonctions mutuellement récursives.

Une autre solution pour tester la fonction par ailleurs est de faire le travail à la main comme dans le cas de base, selon le code suivant :

```
let tableau_medianes (tab : 'a array array) : 'a array =
  let nst = Array.length tab in
  let rep = Array.make nst tab.(0).(0) in
  for i = 0 to nst - 1 do
    let tab5 = tri_insertion tab.(i) in
    let n = Array.length tab5 in
      rep.(i) <- tab5.((n + 1) / 2 - 1)
  done ;
  rep ;;
```

III.4.2 Q10 Complexité

Il faut bien remarquer que les sous tableaux utilisés ont 5 éléments au plus, et donc seront considérés comme des cas de base sur lesquels la médiane sera calculé en temps constant. Il y $n/5$ tels tableaux, et donc la complexité sera linéaire !

III.5 Récursion : troisième étape

III.5.1 Q11

On fera bien sûr là aussi un appel récursif à la fonction **selection** pour calculer la médiane des médianes selon une ligne de code de la forme :

```
let m = select tab_medianes ((Array.length tab_medianes + 1) / 2) in
```

III.6 Récursion : quatrième étape

III.6.1 Q12 fonction partition

Analyse : la seule difficulté est de comprendre qu'il faut faire une première passe pour déterminer la taille de chaque tableau afin de pouvoir les créeer avant de faire le transfert des éléments dans le bon tableau.

Le code est alors très simple :

```
let partition (tab : 'a array) (e : 'a) : 'a array * 'a array =
  let n = Array.length tab in
  let nb_inf = ref 0 in
  for i = 0 to n - 1 do
    if tab.(i) <= e then incr nb_inf
  done ;
  let tab_inf = Array.make !nb_inf e and tab_sup = Array.make (n - !nb_inf) e in
  let i_inf = ref 0 and i_sup = ref 0 in
  for i = 0 to n - 1 do
    if tab.(i) <= e then
      begin
tab_inf.(!i_inf) <- tab.(i) ;
incr i_inf
      end
    else
      begin
tab_sup.(!i_sup) <- tab.(i) ;
incr i_sup
      end
```

```
done ;
tab_inf, tab_sup ;;
```

III.6.2 Q13 Cas où on peut rendre tout de suite la bonne valeur

La seule information que l'on ait à ce niveau, c'est que dans le tableau **tab_inf**, M est le plus grand élément. Et comme tous les éléments du tableau **tab_sup** sont strictement plus grand que M, alors, en notant **n_inf** la taille de **tab_inf**, c'est que le pivot M,i.e. la médiane des médianes est le **n_inf**-ième plus petit élément du tableau reçu en paramètre ! Dès lors si $i = tab\_inf$, alors on rend M.

III.6.3 Q14 Autres cas

Si on n'est pas dans le cas précédent, c'est soit que $i < tab\_inf$, soit que $i > tab\_inf$.

Dans le premier cas on cherche en fait un élément strictement plus petit que M. Il est nécessairement dans le tableau **tab_inf**, et c'est également le i-ième plus petit élément de ce tableau.

Dans l'autre cas on cherche nécessairement dans l'autre tableau **tab_sup**, en revanche il est cette fois le $i - n\_inf$ -ième plus petit élément de ce tableau puisqu'on a enlevé n_inf au tableau initial pour obtenir ce tableau !

III.6.4 Complexité

III.6.5 Q15

On a déjà vu que l'étape de découpage de la première étape était linéaire et va donc intervenir dans le $O(n)$.

De même la deuxième étape de fabrication du tableau des médianes est linéaire, et va grossier le terme en $O(n)$.

La troisième étape consiste en un appel récursif sur le tableau des médianes qui est de taille $\lceil \frac{n}{5} \rceil$. C'est lui est à l'origine du terme $C(\lceil \frac{n}{5} \rceil)$.

Dans la quatrième, la parition du tableau est clairement linéaire, et va encore grossier le terme en $O(n)$.

Dans le pire des cas de cette quatrième étape on va encore appeler récursivement la fonction sur un des deux tableaux. On peut montrer que globalement chacun de ces tableaux contient environ $3n/10$ éléments inférieurs ou supérieurs à la médiane (à compléter). Dès lors chacun ne contient au maximum que $7n/10$ éléments. C'est donc cette partie de cette étape qui est à l'origine du terme $C(7n/10 + 6)$.

III.6.6 Q16

La preuve est immédiate. On en déduit que la complexité asymptotique de notre fonction est linéaire !

III.7 Q17 Bonus : la fonction complète

Vous trouverez du code dans le fichier corrds02.ml accompagnant ce corrigé.