Correction de l'épreuve d'informatique Polytechnique 2008

Ce sujet sera largement commenté par des extraits du rapport du jury, très instructif sur ce que l'on attend des candidats.

Tout d'abord, « *Presque toutes les fonctions demandées pouvaient être écrites en moins de 10 lignes. Les candidats doivent être conscients du fait qu'une réponse longue doit être expliquée en détail et que très souvent un programme très long contient un grand nombre d'erreurs.* »

De plus, « *Pour obtenir la note maximale, il était nécessaire de traiter le problème en entier.* »

I. Préliminaires sur les mots

Question 1 La longueur d'un mot consiste bien sûr à compter les éléments de la liste. On peut l'écrire ainsi :

```
let rec longueurMot = function
  [] -> 0
| _ :: l -> 1 + (longueurMot l)
;;
```

On peut, bien sûr, faire une version récursive terminale :

```
let longueurMot m =
  let rec aux mot longueur =
    match liste with
      [] -> longueur
    | _ :: mot' -> aux mot' (longueur + 1)
  in
  aux l 0
;;
```

Cependant, si cette façon de programmer est une bonne idée en pratique, elle complique en général l'écriture des fonctions. Le rapport évoque d'ailleurs dans ce cas des solutions « *plus compliquées que nécessaire.* »

Question 2 Pas de difficulté pour cette fonction :

```
let rec iemeCar i mot =
  match mot with
    [] -> failwith "iemeCar : mot trop court"
  | lettre :: mot' ->
      if i = 0 then lettre else iemeCar (i - 1) mot'
;;
```

Question 3 Il faut bien faire attention à avoir, pour cette fonction, une complexité linéaire et non quadratique.

```
let rec prefixe k mot =
  if k = 0 then []
  else match mot with
    [] -> failwith "prefixe : mot trop court"
  | lettre :: mot' -> lettre :: (prefixe (k - 1) mot')
;;
```

Question 4 À nouveau, il convient d'avoir la bonne complexité.

```
let rec suffixe k mot =
  if k = 0 then mot
  else match mot with
    [] -> failwith "suffixe : mot trop court"
  | _ :: mot' -> suffixe (k - 1) mot'
;;
```

II. Opérations sur les cordes

Question 5 Attention, lisez bien l'énoncé et la définition d'une corde. Cette fonction peut être réalisée en temps constant :

```
let longueur = function
  Vide -> 0
| Feuille (l, _) -> l
| Noeud (l, _, _) -> l
;;
```

Une solution de complexité linéaire utilisant la fonction `longueurMot` est pénalisée.

Question 6 À nouveau, il faut avoir l'invariant en tête pour traiter le cas du mot vide :

```
let nouvelleCorde = function
```

```
  [] -> Vide
| mot -> Feuille (longueurMot mot, mot)
;;
```

Question 7 Le mot vide ne pouvant être fils d'un noeud, il faut à nouveau penser à traiter les bons cas.

```
let concat c1 c2 =
  match (c1, c2) with
    Vide, _ -> c2
  | _, Vide -> c1
  | _ -> Noeud ((longueur c1) + (longueur c2), c1, c2)
;;
```

Question 8 La condition sur i par rapport à la longueur de la corde permet de ne pas faire de tests inutiles.

```
let rec caractere i c =
  match c with
    Vide -> failwith "caractere : corde vide"
  | Feuille (_, m) -> iemeCar i m
  | Noeud (_, c1, c2) ->
      let l1 = longueur c1 in
      if i < l1 then caractere i c1 else caractere (i - l1) c2
;;
```

Question 9 Voyons ce que dit le rapport : « *Cette question est apparue plus diffcile que les précédentes, la première diffculté étant de bien considérer les trois cas de figure (la sous-corde est entièrement contenue dans le sous-arbre gauche, dans le sous-arbre droit, ou chevauche les deux) et la seconde de traiter avec soin les nombreux paramètres des appels récursifs.* »

```
let rec sousCorde i m c =
  (* il est possible d'avoir m = 0, on connait alors le résultat *)
  if m = 0 then Vide else
  match c with
    Vide -> failwith "sousCorde : corde vide"
  | Feuille (_, mot) -> Feuille (m, prefixe m (suffixe i mot))
  | Noeud (_, c1, c2) ->
      let l1 = longueur c1 in
      if i >= l1
      then sousCorde (i - l1) m c2
      else if i + m <= l1
        then sousCorde i m c1
        else concat
          (sousCorde i (l1 - i) c1) (sousCorde 0 (m - l1 + i) c2)
;;
```

III. Équilibrage

Question 10 On rappelle les premières valeurs de la suite de Fibonacci :

$$F_0 = 0,\ F_1 = F_2 = 1,\ F_3 = 2,\ F_4 = 3,\ F_5 = 5,\ F_6 = 8,\ F_7 = 13,\ F_8 = 21, \ldots$$

Ainsi, par exemple, la case 5 ne peut contenir qu'une corde dont la longueur est comprise entre 5 et 7.

Simulons donc l'algorithme. On commence par insérer la feuille `F("Ec")` dans la case 2. Or, on obtient une corde de longueur 2, on est donc obligé de la décaler à la case 3.

Nous devons maintenant insérer la feuille `F("oleP")`, en partant de la case 2. La feuille est trop grand, on concatène avec la corde de la case 3, pour obtenir la corde `N(F("Ec"), F("oleP"))` de longueur 6. On doit atteindre la case 5 pour la ranger.

Ensuite, la feuille `F("o")` peut être ranger dans la case 2.

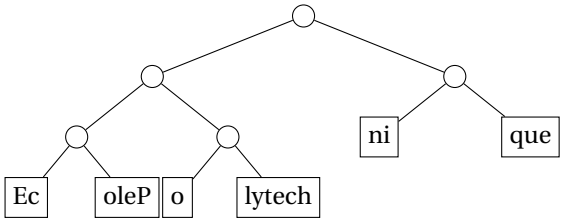
En insérant la feuille `F("lytech")` dans la case 2, on obtient la corde `N(F("o"), F("lytech"))` de longueur 7. On est obligé de la décaler vers la droite jusqu'au moins la case 5. Mais on y trouve la corde précédente de longueur 6. Concaténées, on obtient la corde :

```
N(N(F("Ec"), F("oleP")), N(F("o"), F("lytech")))
```

Cette corde, de longueur 13, va dans la case 7. À ce moment, toutes les autres cases contiennent une corde vide.

L'insertion successive des feuilles `F("ni")` et `F("que")` amène à avoir la corde `N(F("ni"), F("que"))` de longueur 5 dans la case 5.

Finalement, toutes les feuilles ayant été insérées, on construit la corde finale en concaténant les cordes obtenus dans le sens des indices décroissants :



Il faut insister sur l'importance d'une telle question. Faire un exemple à la main est un excellent moyen d'appréhender un algorithme et de se l'approprier. Les personnes ayant pris le temps de répondre à cette question serait très avantagés plus tard lorsqu'il faudra en donner un implémentation.

Question 11 Bien sûr, on utilise une méthode de complexité linéaire et non exponentielle…

```
let initialiserFib () =
  fib.(0) <- 0 ;
  fib.(1) <- 1 ;
  for i = 2 to tailleMax do
    fib.(i) <- fib.(i - 1) + fib.(i - 2)
  done
;;
```

Question 12 Une traduction directe de l'algorithme de remplissage donne :

```
let rec inserer c i =
  let c' = concat file.(i) c in
  if longueur c' < fib.(i+1)
  then file.(i) <- c'
  else begin
    file.(i) <- Vide ;
    inserer c' (i + 1)
  end
;;
```

Le rapport du jury met en garde sur l'oublie fréquent du `file.(i) <- Vide`, ainsi que sur l'absence de traitement du dépassement de la capacité de la file. Ce dernier point est plus étonnant, vu le manque d'information sur ce qu'il convient de faire en cas de dépassement. La levée d'exception de Caml semble suffisant dans une première approche.

Question 13 Supposons donc que les invariants $\texttt{hauteur}(c_i) \leq i-2$ et $\texttt{longueur}(c_i) \geq F_i$ sont vérifiés pour toutes les cordes non vides du tableau. Si l'on appelle la fonction `inserer` pour une corde c et une position i, avec $2 \leq i < \texttt{tailleMax}$, $\texttt{hauteur}(c) \leq i-2$ et $\texttt{longueur}(c) \geq 2$, deux cas sont possibles :

- soit la case i est vide, auquel cas on peut bien mettre la corde c dans la case i, l'invariant étant bien respecté ;
- soit la case i est non vide. Elle contient la corde c_i qui, d'après l'invariant, vérifie $\texttt{hauteur}(c_i) \leq i-2$ et $\texttt{longueur}(c_i) \geq F_i$. La corde c' obtenue en concatenant c_i et c vérifie alors :

$$\texttt{hauteur}(c') = 1 + \max(\texttt{hauteur}(c), \texttt{hauteur}(c_i)) \leq i-1 = (i+1)-2$$
$$\texttt{longueur}(c') = \texttt{longueur}(c) + \texttt{longueur}(c_i) \geq F_i + F_i \geq F_{i+1}$$

  D'après l'algorithme utilisé, il faut donc insérer la corde c' obtenue à partir de la case d'indice $i+1$, et de mettre la corde `Vide` dans la case i. Comme on l'a vue, la corde c' vérifie les conditions pour être insérée à partir de la case d'indice $i+1$, donc l'invariant sera bien vérifié.

Au début de l'algorithme, le tableau ne contient que des cordes vides, l'invariant est vérifié. De plus, chaque insertion commence par insérer une feuille non vide dans la case d'indice 2. Les conditions de la fonction `inserer` sont donc bien vérifiée (la hauteur est majorée par $0 = 2-2$ et la longueur est minorée par $F_2 = 1$). Ainsi, tout au long de l'exécution de l'algorithme, l'invariant est vérifié par le tableau.

À nouveau, tout ceci est vrai dans que l'on n'a pas de dépassement de la taille du tableau.

Question 14 À priori, l'écriture de cette fonction ne pose pas de problème. Cependant, le rapport évoque « *plusieurs possibilités de ne pas respecter l'algorithme proposé : soit en construisant une corde qui ne représentait pas le bon mot (par inversion des sous-cordes), soit en construisant une corde n'ayant pas la bonne hauteur (en concaténant les éléments de la file dans l'ordre inverse).* »

```
let equilibrer corde =
  (* rien ne dit que le tableau file contient les bonnes valeurs *)
  for i = 2 to tailleMax - 1 do
    file.(i) <- Vide
  done ;
  let rec insere_corde = function
    Vide -> ()
  | (Feuille (_, m) as feuille) -> inserer feuille 2
  | Noeud (_, c1, c2) -> insere_corde c1 ; insere_corde c2
  in
  insere_corde corde ;
  let res = ref Vide in
  for i = 2 to tailleMax - 1 do
```

```
    res := concat file.(i) !res
  done ;
  !res
;;
```

Pour la dernière partie de l'équilibrage, à savoir la reconstruction de la corde à partir du tableau, l'énoncé est assez peu clair. En effet, il parle de concaténer les éléments dans l'ordre inverse. Mais il faut malgré tout aller de gauche à droite pour parcourir le tableau et non l'inverse. En effet, les cordes ont des hauteurs et des coûts croissants en allant de gauche à droite, et il faut mettre les cordes les plus hautes le plus près possible de la racine. Ainsi, si l'on doit concatener des cordes $[c_1;\ldots;c_n]$ (on voit le tableau comme une liste où l'on a omit les cordes vides), il faut retourner la corde :

$$\texttt{Noeud(}c_n\texttt{ ; Noeud(...(Noeud(}c_2\texttt{ ; }c_1\texttt{))...))}$$

et non la corde :

$$\texttt{Noeud(Noeud(...(Noeud(}c_n\texttt{ ; }c_{n-1}\texttt{) ; ...) ; }c_2\texttt{) ; }c_1\texttt{)}$$

Question 15 D'après les questions précédentes, on a :

$$n = \sum_i \texttt{longueur}(c_i)$$

Intéressons-nous maintenant à la hauteur h_k de l'arbre obtenu en concaténant les cordes contenues dans les k premières cases non « `vides` » du tableau. Si l'on note i_k l'indice de la k-ième case non vide, on montre par récurrence que l'on a $h_k \leq i_k - 1$.

Pour $k = 1$, on ne fait aucune concaténation, on a donc, d'après l'invariant précédent :

$$h_1 = \texttt{hauteur}(c_{i_1}) \leq i_1 - 2 \leq i_1 - 1$$

Si notre relation est vraie au rang k, on a alors :

$$h_{k+1} = 1 + \max(h_k, \texttt{hauteur}(c_{i_{k+1}})) \leq 1 + \max(i_k - 1, i_{k+1} - 2)$$

Mais $i_k < i_{k+1}$, donc on a bien $h_{k+1} \leq i_{k+1} - 1$.

Ainsi, la relation est vrai pour tout k, elle est donc vraie pour p le nombre total de cordes à concaténer. On a donc finalement $h \leq i_p - 1$, autrement dit $i_p \geq h + 1$. On a alors :

$$n = \sum_k c_{i_k} \geq c_{i_p} \geq F_{i_p} \geq F_{h+1}$$

Pour la seconde inégalité, on a :

$$\text{Coût}(c) = \sum_i \texttt{longueur}(c_i)\texttt{hauteur}(c_i) \leq hn$$

Or, puisque d'après l'indication, comme $F_{h+1} \leq n$, on en déduit que $h \leq \log_\phi n + \log_\phi \sqrt{5}$. On a donc bien, en posant $K = \frac{1}{2}\log_\phi 5$:

$$\text{Coût}(c) \leq n \times (\log_\phi n + K)$$

IV. Équilibrage optimal

Question 16 L'initialisation ne pose pas de problème. Pourtant, le rapport indique que « *Une partie non négligeable des candidats a choisi d'utiliser une fonction auxiliaire renvoyant la liste des feuilles d'une corde. Cette solution était acceptée si elle était réalisée avec une complexité linéaire, ce qui était rarement le cas (la réalisation la plus courante consistant à utiliser la concaténation de listes, résultant en une complexité quadratique). Il existait cependant une solution directe de complexité linéaire.* » La fonction suivante illustre une telle solution.

```
let initialiserQ c =
  let rec aux c k =
    match c with
      Vide -> k
    | Feuille _ -> (q.(k) <- c ; k + 1)
    | Noeud (_, c1, c2) -> aux c2 (aux c1 k)
  in
  (aux c 0) - 1
;;
```

Question 17 On propose la fonction suivante, où `trouve_feuille` recherche dans le tableau `q` la position d'une feuille donnée. Il doit exister une telle position, donc il n'y a pas de test en cas de problème.

```
let initialiserProf c c1 =
  let k = initialiserQ c in
  let rec trouve_feuille f i =
    if f = q.(i) then i else trouve_feuille f (i+1)
  in
  let rec parcours c p =
    match c with
      Vide -> ()
    | Feuille _ -> prof.(trouve_feuille c 0) <- p
    | Noeud (_, c1, c2) -> (parcours c1 (p + 1); parcours c2 (p + 1))
  in
  parcours c1 0
;;
```

Il faut remarquer que cette façon de procéder n'est *a priori* valide que lorsque toutes les feuilles sont distinctes.

Question 18 Citons une fois de plus le rapport : « *C'était peut-être la question la plus difficile du sujet, mais il existait néanmoins de nombreuses solutions : une solution courte*

et élégante de complexité linéaire et utilisant la récursivité ; une autre consistant à rechercher les deux premières feuilles de profondeur maximale et à les concaténer, avant de réitérer le processus ; une solution similaire mais consistant à rechercher les deux premières feuilles consécutives de même profondeur ; ou encore une solution consistant à insérer successivement chaque feuille le plus à gauche dans un arbre initialement vide. Cette dernière solution a été tentée par de nombreux candidats, mais s'est révélée difficile à mettre en œuvre et n'a été traitée correctement qu'une seule fois. »

La réponse que je vous propose semble correspondre à la solution « *courte et élégante de complexité linéaire et utilisant la récursivité.* » On construit l'arbre par récursivité en indiquant à chaque moment à quelle profondeur on se trouve, et le compteur `indice` indique la prochaine feuille à utiliser. La complexité est linéaire, puisque le nombre d'appel à la sous-fonction aux est égale à au nombre de feuille n_f ajouté au nombre de noeuds, autrement dit à $2n_f - 1$.

```
let reconstruire () =
  let indice = ref 0 in
  let rec aux profondeur =
    if profondeur = prof.(!indice)
    then begin (* on est à la bonne profondeur *)
      let feuille = q.(!indice) in
      indice := !indice + 1 ;
      feuille
    end else begin
      let c1 = aux (profondeur + 1) in
      let c2 = aux (profondeur + 1) in
      Noeud ((longueur c1 + longueur c2), c1, c2)
    end
  in aux 0
;;
```

J'indique aussi la solution proposée par Laurent Cheno, dont la justification est la suivante : « *Pour une fois, surtout parce que l'énoncé nous demande d'utiliser des tableaux, nous proposons un programme en style impératif… On cherche dans le tableau de profondeurs le premier indice i tel que $p_i = p_{i+1}$: on remplace alors q_i par la corde concaténée de q_i et q_{i+1} et on supprime q_{i+1}, et, de même, on remplace p_i par $p_i - 1$ et on supprime p_{i+1}. Ainsi on commencera par trouver les deux premières feuilles de même profondeur, et on les remplace par un nœud père de ces deux feuilles. Au fur et à mesure, les q_i deviennent des cordes de plus en plus complexes, sous-arbres de la corde finale, rangés dans le bon ordre, et p_i indique à quelle profondeur doit se trouver la racine de la corde q_i dans la corde finale (C'est là l'invariant de boucle.) On obtient le programme suivant, où la fonction auxiliaire agglutine se charge des modifications des deux tableaux `prof` et `q`.* »

```
let reconstruire_bis () =
  let agglutine i =
  prof.(i) <- prof.(i) - 1 ;
  q.(i) <- concat q.(i) q.(i+1) ;
  let j = ref (i+1) in
  while prof.(!j) > 0 do
    prof.(!j) <- prof.(!j+1) ;
    q.(!j) <- q.(!j+1) ; incr j
  done
in
let i = ref 0 in
while prof.(0) > 0 do
  i := 0 ;
  while prof.(!i) <> prof.(!i+1) do
    incr i
  done ;
  agglutine !i
done ;
q.(0)
;;
```

Question 19 Pour finir, une question pour mettre en forme les fonctions précédentes. Il faut cependant faire attention car la méthode utilisée (en particulier la reconstruction) ne s'applique pas à la corde vide.

```
let equilibrerOpt = function
  Vide -> Vide
| corde -> begin
    let c1 = phase1 corde in
    initialiserProf corde c1 ;
    reconstruire ()
  end
;;
```