X 2008 — Option informatique

Partie I — Préliminaires sur les mots

Question I.1
Il s'agit simplement du calcul de la longueur d'une liste.

```
1  let rec longueurMot = function
2    | [] -> 0
3    | _ :: q -> longueurMot q + 1 ;;
```

Programme 1 La fonction `longueurMot`

Question I.2
On peut ici supprimer la ligne 5 si on ne souhaite pas vérifier les arguments de la fonction.

```
4  let rec iemeCar i = function
5      | [] -> failwith "erreur d'argument pour iemeCar"
6      | t :: q -> if i = 0 then t else iemeCar (i-1) q ;;
```

Programme 2 La fonction `iemeCar`

Question I.3
Si on veut se passer de vérification, la ligne 7 peut être réécrite simplement `| [] -> []`.

```
7  let rec prefixe k = function
8    | [] -> if k = 0 then [] else failwith "erreur d'argument pour prefixe"
9    | t :: q -> if k >= 1 then t :: (prefixe (k-1) q) else [] ;;
```

Programme 3 La fonction `prefixe`

Question I.4
Même remarque !

```
10 let rec suffixe k = function
11   | [] -> if k = 0 then [] else failwith "erreur d'argument pour suffixe"
12   | t :: q -> if k>=1 then suffixe (k-1) q else t :: q ;;
```

Programme 4 La fonction `suffixe`

Partie II — Opérations sur les cordes

Question II.5
Les nœuds de l'arbre contiennent l'information utile : pas d'appel récursif nécessaire.

```
13  let longueur = function
14    | Vide -> 0
15    | Feuille(n,_) -> n
16    | Noeud(n,_,_) -> n ;;
```

Programme 5 La fonction `longueur`

Question II.6
Il faut simplement distinguer le cas où $n = 0$.

```
17  let nouvelleCorde x = let n = longueurMot x in
18    if n = 0 then Vide else Feuille (n,x) ;;
```

Programme 6 La fonction `nouvelleCorde`

Question II.7
Un nœud ne peut avoir de fils vide, il faut donc traiter à part le cas où l'un des arguments est la corde vide.

```
19  let concat c1 c2 = match (c1,c2) with
20      | (Vide,c2) -> c2
21      | (c1,Vide) -> c1
22      | (c1,c2) -> Noeud((longueur c1) + (longueur c2),c1,c2)  ;;
```

Programme 7 La fonction `concat`

Question II.8
Pas de difficulté particulière ici.

```
23  let rec caractere i = function
24      | Vide -> failwith "erreur d'argument pour caractere"
25      | Feuille(_,x) -> iemeCar i x
26      | Noeud(_,c1,c2) -> let n1 = longueur c1 in
27         if i < n1 then caractere i c1
28         else caractere (i-n1) c2 ;;
```

Programme 8 La fonction `caractere`

Question II.9
Dans le cas d'un nœud, on distingue trois cas : ou bien le facteur cherché est facteur de la partie gauche `c1`, ou bien de la partie droite `c2`, ou bien c'est le résultat de la concaténation d'un suffixe de `c1` et d'un préfixe de `c2`. On obtient le programme 9, page 3.

```
29  let rec sousCorde i m = function
30      | Vide -> failwith "erreur d'argument pour sousCorde"
31      | Feuille(n,x) -> Feuille(m,prefixe m (suffixe i x))
32      | Noeud(n,c1,c2) -> let n1 = longueur c1 in
33         if m+i <= n1 then sousCorde i m c1
34         else if i >= n1 then sousCorde (i-n1) m c2
35         else concat (sousCorde i (n1-i) c1) (sousCorde 0 (m-n1+i) c2) ;;
```

Programme 9 La fonction `sousCorde`

Partie III — Équilibrage

Question III.10
Rappel : $F_0 = 0$, $F_1 = 1$, $F_2 = 1$, $F_3 = 2$, $F_4 = 3$, $F_5 = 5$, $F_6 = 8$, $F_7 = 13\ldots$ Ainsi, dans la case 2, on ne peut ranger qu'une corde de longueur 1 ; dans la case 3, de longueur 2 ; dans la case 4, de longueur 3 ou 4; dans la case 5, de longueur 5, 6 ou 7 ; dans la case 6, de longueur entre 8 et 12 ; dans la case 7, de longueur entre 13 et 20.
On part avec $i = 2$ et $j = 0$: $x_0 =$ `Ec` ne peut être rangé dans la case 2. Donc on range `Vide` dans la case 2. On continue avec $i = 3$ et on range `Ec` dans la case 3.
On range maintenant $x_1 =$ `oleP` en partant de $i = 2$. Il n'y a bien sûr pas la place dans la case 2. On appelle donc c' le concaténé de x_0 et de x_1, on affecte `Vide` à la case 3, et on ne peut ranger c' (de longueur 6) que dans la case 5.
On range maintenant $x_2 =$ `o` en partant de $i = 2$: on range simplement x_2 dans la case 2.
On range maintenant $x_3 =$ `lytech` en partant de $i = 2$: on affecte `Vide` à la case 2 et on range c', concaténé de x_2 et de x_3, de longueur 7, à partir de $i = 3$. On efface la case 3, et on range finalement `Noeud(13,Noeud(6,Feuille(2,Ec),Feuille(4,oleP)),Noeud(7,Feuille(1,o),Feuille(6,lytech)))` dans la case 7.
On range maintenant $x_4 =$ `ni` en partant de $i = 2$ et on le range finalement dans la case 3.
Reste $x_5 =$ `que` : on range le concaténé c' de x_4 et de x_5, qui est de longueur 5, dans la case 5.
À la fin : il n'y a que deux cases non vides, la case 5 qui contient `Noeud(5,Feuille(2,ni),Feuille(3,que)` et la case 7 qui contient `Noeud(13,Noeud(6,Feuille(2,Ec),Feuille(4,oleP)),Noeud(7,Feuille(1,o),Feuille(6,lytech)))`.
La corde obtenue est représentée par l'arbre de la figure 1. La corde initiale avait un coût égal à 58, la corde finale a un coût égal à 49.

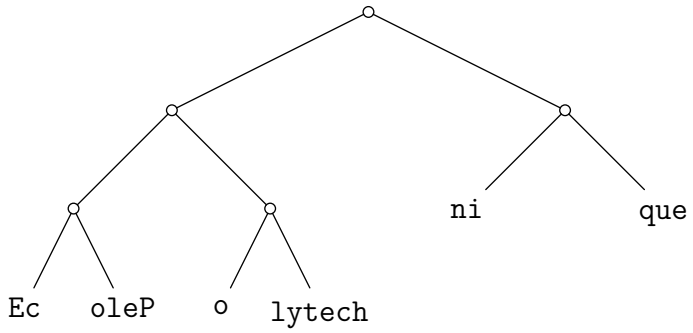

Figure 1 La corde obtenue par l'algorithme

Question III.11
Aucune difficulté ici. On obtient le programme 10, page 4.

Question III.12
On écrit le programme 11 de la page 4 en appliquant l'algorithme décrit par l'énoncé.

Question III.13
Au début de l'algorithme, toutes les cases du tableau `file` sont vides, et l'invariant est vérifié.

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

Programme 10 La fonction `initialiserFib`

```
40  let rec inserer c i =
41    let p = longueur c in
42      if file.(i) = Vide then
43        if p < fib.(i+1) then file.(i) <- c
44        else inserer c (i+1)
45      else
46        let c' = concat file.(i) c in let p' = longueur c' in
47          if p' < fib.(i+1) then file.(i) <- c'
48          else (file.(i) <- Vide ; inserer c' (i+1)) ;;
```

Programme 11 La fonction `inserer`

Vérifions que l'invariant proposé est préservé à chaque appel récursif de la fonction `inserer`, sachant que l'on commence toujours par un appel `inserer c 2` où la corde `c` est réduite à une feuille contenant un des mots x_j.
De deux choses l'une :

- ou bien on cherche à ranger un mot x_j de longueur n dans la case i : c'est que les cases de numéro 2 à $i-1$ étaient vides, car sinon ce ne serait plus x_j lui-même que nous serions en train de ranger. Cela entraîne en particulier que $n \geqslant F_i$.
  - Si la case i est vide, ou bien on recommence avec la case $i+1$, et l'invariant est toujours vrai, ou bien $F_i \leqslant |x_j| < F_{i+1}$ et on range x_j dans la case i : la feuille contenant x_j est de hauteur nulle et $i \geqslant 2$, donc l'invariant est bien vérifié.
  - Si la case i est non vide, elle contient une corde c de hauteur $h \leqslant i-2$ et de longueur p telle que $F_i \leqslant p < F_{i+1}$, d'après ce que nous garantit notre invariant. Mais alors $n+p \geqslant 2F_i = F_i + F_i > F_{i+1}$. Or, pour insérer x, on concatène c et x obtenant une corde c' de hauteur $h+1$ et de longueur $n+p > F_{i+1}$. L'algorithme demande qu'on vide la case i, donc l'invariant est encore vrai, avant le prochain appel récursif.
- ou bien on est en train de ranger une corde c de longueur n et de hauteur h dans la case i : cette corde a été créée dans une étape précédente par concaténation de la corde qui figurait dans une case c_k où $k \leqslant i-1$ avec une autre corde, comme on l'a vu dans le cas précédent, donc il est sûr que $h-1 \leqslant k-2 \leqslant (i-1)-2$ et que $n \geqslant F_i$. Finalement on a bien $n \geqslant F_i$ et $h \leqslant i-2$. Le raisonnement se poursuit comme ci-dessus :
  - Si la case i est vide, ou bien on recommence avec la case $i+1$ et l'invariant reste vrai, ou bien $F_i \leqslant n < F_{i+1}$ et on range la corde c dans la case i et l'invariant est vrai puisque $h \leqslant i-2$.
  - Si la case i est non vide, elle contient une corde c' de hauteur $h' \leqslant i-2$ et de longueur p' telle que $F_i \leqslant p' < F_{i+1}$, d'après ce que nous garantit notre invariant. Mais alors $n+p > F_{i+1}$. Or, pour insérer c, on concatène c' et c, obtenant une corde c'' de hauteur $h'' = 1+\max(h',h) \leqslant i-1$ et de longueur $n+p > F_{i+1}$ L'algorithme demande qu'on vide la case i, donc l'invariant est encore vrai, avant le prochain appel récursif.

Question III.14
On insère successivement les mots qui figurent dans la corde de départ, lors d'un parcours récursif de l'arbre. Il suffit alors de récupérer les éléments de la file qu'on concatène, en la parcourant à rebours. Comme `concat` gère convenablement et efficacement les cordes vides, il n'est pas utile d'ajouter un test dans la fonction récursive auxiliaire `recupereResultat`. On obtient le programme 12, page 5.

```
let equilibrer c =
  let rec insereTous = function
      | Vide -> ()
      | Feuille(n,x) as f -> inserer f 2
      | Noeud(_,c1,c2) -> (insereTous c1 ; insereTous c2 )
  in
  let rec recupereResultat c = function
      | 1 -> c
      | i -> recupereResultat (concat c file.(i)) (i-1)
  in
    insereTous c ;
    recupereResultat Vide (tailleMax-1) ;;
```

Programme 12 La fonction `equilibrer`

Question III.15
La corde c est constituée de la concaténation (à rebours) des cordes non vides c_i du tableau `file` après la suite d'insertions.
Donc $n = \sum_i \texttt{longueur}(c_i)$.
Quand on concatène des cordes non vides, dont la hauteur maximale est h : ou bien cette hauteur h est atteinte au moins deux fois consécutivement par des cordes de départ et alors le résultat est de hauteur $h+2$, ou bien le résultat est de hauteur $h+1$.
Notons $p = \max\{i, c_i \neq \texttt{Vide}\}$. D'après ce qui précède, $\texttt{hauteur}(c_i) \leqslant i-2$ pour tout i tel que $c_i \neq$ Vide, donc hauteur$(c_p) \leqslant p-2$ et hauteur$(c_i) \leqslant p-3$ pour $i < p$.
Donc soit le maximum de hauteur vaut $p-2$ mais n'est atteint qu'une fois, soit il est au plus égal à $p-3$.
On en déduit que, dans tous les cas, $h =$ hauteur$(c) \leqslant p-1$, donc $F_{h+1} \leqslant F_p \leqslant \texttt{longueur}(c_p) \leqslant n$.
On en déduit que $Coût(c) = \sum_j \texttt{longueurMot}(x_j) \times \texttt{prof}(x_j) \leqslant h \sum_j \texttt{longueurMot}(x_j) = hn$.
Comme $\dfrac{\phi^h}{\sqrt{5}} \leqslant F_{h+1} \leqslant n$, on en déduit que $h \leqslant \log_\phi(n\sqrt{5}) = K + \log_\phi n$ où on a posé $K = \dfrac{1}{2}\log_\phi 5$.
Alors on a bien $Coût(c) \leqslant n(K + \log_\phi n)$.

Partie IV — Équilibrage optimal

Question IV.16
Au lieu de commencer par dresser la liste des feuilles de la corde argument pour les ranger ensuite dans le tableau, on se propose de les ranger au fur et à mesure de leur découverte dans un parcours de l'arbre.
On obtient le programme 13.

```
let initialiserQ c =
  let rec parcours i = function
    | Vide -> failwith "argument interdit"
    | Feuille _ as f -> (q.(i) <- f ; i+1)
    | Noeud(_,c1,c2) -> let i' = parcours i c1 in parcours i' c2
  in
    parcours 0 c - 1 ;;
```

Programme 13 La fonction `initialiserQ`

Bien que ce ne soit pas demandé, on trouvera dans la figure 2 la corde obtenue dans la phase 1 à partir de la corde qui nous a déjà servi d'exemple dans la question III.10. Le lecteur curieux trouvera à la fin de ce document le programme 17 qui fournit une programmation possible (plus fastidieuse que difficile) de la fonction `phase1`.

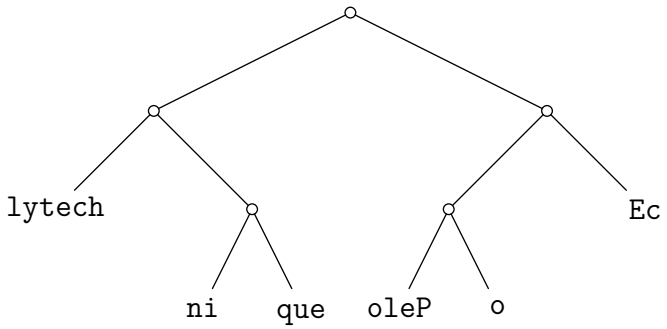

Figure 2 Après la phase 1

La corde obtenue est de coût égal à 46 : c'est effectivement meilleur que ce que nous avions déjà trouvé. Mais on observe que les feuilles ne sont plus dans l'ordre.

Question IV.17
On propose le programme 14 : la fonction auxiliaire `index : mot -> int -> int` calcule, lors de l'appel `index x 0`, l'index i tel que $x = x_i$. Il suffit alors de parcourir la corde c_1 en gardant trace de la profondeur courante `p` et de la ranger au bon endroit quand on rencontre une feuille.

```
68  let initialiserProf c c1 =
69    let k = initialiserQ c in
70    let rec index x i = if q.(i) = x then i else index x (i+1) in
71    let rec parcours p = function
72      | Vide -> failwith "corde vide"
73      | Feuille _ as f -> prof.(index f 0) <- p
74      | Noeud(_,c1,c2) -> ( parcours (p+1) c1 ; parcours (p+1) c2 )
75    in
76      parcours 0 c1 ;;
```

Programme 14 La fonction `initialiserProf`

Question IV.18
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 15, page 7, où la fonction auxiliaire `agglutine` se charge des modifications des deux tableaux `prof` et `q`.
Poursuivant sur notre exemple, on obtient la corde finale de coût optimal de la figure 3, page 7.

Question IV.19
Il n'y a plus qu'à recoller les morceaux...

```
77  let reconstruire () =
78    let agglutine i =
79      prof.(i) <- prof.(i) - 1 ;
80      q.(i) <- concat q.(i) q.(i+1) ;
81      let j = ref (i+1) in while prof.(!j) > 0 do
82         prof.(!j) <- prof.(!j+1) ; q.(!j) <- q.(!j+1) ; incr j
83        done
84    in
85    let i = ref 0 in
86      while prof.(0) > 0 do
87        i := 0 ;
88        while prof.(!i) <> prof.(!i+1) do incr i done ;
89        agglutine !i
90      done ;
91      q.(0) ;;
```

Programme 15 La fonction `reconstruire`

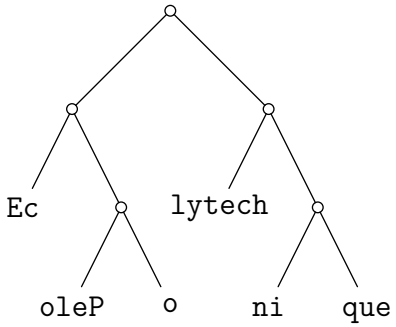

Figure 3 La corde c_2

```
92  let equilibrerOpt c =
93    let c1 = phase1 c in
94      initialiserProf c c1 ;
95      reconstruire () ;;
```

Programme 16 La fonction `equilibrerOpt`

```
96  let phase1 c =
97    let k = initialiserQ c in
98    let m = ref k in
99      while !m > 0 do
100       let i = ref 1 and j = ref 0 in
101        while !i < !m && longueur q.(!i-1) > longueur q.(!i+1) do incr i done ;
102        let c' = concat q.(!i-1) q.(!i) in
103          j := !i-1 ;
104          while !j>0 && longueur q.(!j-1) < longueur c' do decr j done ;
105          for k = !i-1 downto !j+1 do q.(k) <- q.(k-1) done ;
106          q.(!j) <- c' ;
107          decr m ;
108          for k = !i to !m do q.(k) <- q.(k+1) done
109     done ;
110     q.(0) ;;
```

Programme 17 La fonction `phase1`