Correction du devoir surveillé n°2

MPSI Lycée Clemenceau : option informatique

Mardi 7 mai 2024

Vous avez 3 heures.

Il sera tenu compte de la présentation et de la rédaction. Les programmes (ou fonctions) devront être commentés et le plus souvent justifiés (terminaison et correction).

Exercice I

1) À une liste `a=[`a_0`; ...;` a_n`]` d'entiers entre 0 et 9 on associe l'entier $\sum_{i=0}^{n} a_i 10^i$ et réciproquement à un entier positif on associe la liste de ses chiffres en base 10 à l'envers sans 0 à la fin, en convenant que la liste associée à 0 est `[]`, et que réciproquement à `[]` on associe 0.

Ainsi à `[1; 5; 7; 2]` est associé l'entier 2751, et réciproquement à 2751 est associée la liste `[1; 5; 7; 2]`.

Dans la suite toutes les listes sont formées d'entiers entre 0 et 9, et éventuellement vides.

a) Écrire une fonction récursive `valeur : int list -> int` telle que `valeur a` renvoie l'entier associé à `a`.

Correction : on utilise le fait que $a_n a_{n-1}..a_2 a_1 a_0 = a_0 + 10(a_1 + 10(a_2 + 10(...)))$.

```
let rec valeur lst = match lst with
        | [] -> 0
        | a::q -> a+10*(valeur q);;
```

b) Écrire une fonction récursive `liste : int -> int list` telle que `liste n` renvoie la liste associée à `n`, `n` étant supposé positif.

Correction : on utilise le reste et le quotient de la division euclidienne par 10 pour écrire la récursivité.

```
let rec liste n = match n with
        | 0 -> []
        | _ -> (n mod 10)::(liste (n/10));;
```

2) Ici, à une liste non vide d'entiers `[`a_0`; ...;` a_n`]` on associe le polynôme $a_0+a_1X+...+a_nX^n$, et à un polynôme $a_0 + a_1X + ... + a_nX^n$ avec $a_n \neq 0$, on associe la liste `[`a_0`; ...;` a_n`]`. On associe au polynôme nul la liste `[]` et inversement à `[]` on associe le polynôme nul.

a) Écrire une fonction `val0 : int list -> int` de sorte que `val0 a` renvoie la valeur en 0 du polynôme associé à `a`. C'est-à-dire $P(0)$ si `a` est associée à $P(X)$.

Correction : la valeur en 0 correspond à a_0 donc :

```
let val0 lst = match lst with
          | [] -> 0
          | a::q -> a;;
```

b) Écrire une fonction `val1 : int list -> int` de sorte que `val1 a` renvoie la valeur en 1 du polynôme associé à `a`. C'est-à-dire $P(1)$ si `a` est associé à $P(X)$.

Correction : on a $P(1) = \sum_{k=0}^{n} a_k$ d'où

```
let rec val1 lst = match lst with
        | [] -> 0
        | a::q -> a + val1 q;;
```

c) Écrire une fonction `derivation : int list -> int list` de sorte que `derivation a` renvoie la liste associée à la dérivée du polynôme associé à `a`.

Correction : on utilise une fonction auxiliaire qui permet de gérer le degré. On fait de plus attention au fait que la dérivée d'un polynôme constant est nulle.

```
let derivation lst = match lst with
        | [] -> []
        | t::q -> let rec aux b k = match b with
              | [] -> []
              | t::q -> (k * t)::(aux q (k+1))
            in aux q 1;;
```

d) Écrire une fonction `integration : int list -> int list` de sorte que `integration a` renvoie la liste associée à la primitive nulle en 0 du polynôme associé à `a`. *On supposera que la primitive est encore à coefficients entiers.*

Correction : il faut gérer le fait que la primitive de la fonction nulle cherchée est la fonction nulle. De plus il y a une erreur d'énoncé (dans le sujet de concours initial) car on ne peut pas garder des coefficients entiers du fait de la division.

```
let integration lst = match lst with
        | [] -> []
        | a -> let rec aux b k =match b with
              | [] -> []
              | t::q -> (t/(k+1))::(aux q (k+1))
            in 0::(aux a 0);;
```

e) Écrire une fonction récursive `somme : int list -> int list -> int list` de sorte que `somme a b` renvoie la liste associée à la somme des polynômes associés à `a` et `b`.

(remarque : les listes `a` et `b` ne sont pas supposées de mêmes longueurs)

Cor il suffit de sommer les termes des deux listes tant qu'il y a des éléments au même niveau. Il faut faire attention aussi au fait que la somme peut être nulle.

```
let rec somme l1 l2 = match (l1,l2) with
        | lst,[] -> lst
        | [],lst -> lst
        | [a],[b] when a+b = 0 -> []
        | [a],[b] -> [a+b]
        | a::p,b::q -> let r = (somme p q) in
              if a + b <> 0 then (a+b)::r else if r = [] then [] else 0::r ;;
```

f) Écrire une fonction récursive `produit : int list -> int list -> int list` de sorte que `produit a b` renvoie la liste associée au produit des polynômes associés à `a` et `b`.

(même remarque)

Correction : on va utiliser une fonction auxiliaire qui multiplie un polynôme par un coefficient et on gère l'augmentation du degré.

```
let rec produit l1 l2 = match (l1,l2) with
          | lst,[]-> []
          | [],lst -> []
          | a::q,lst -> let rec aux b l = match b,l with
                      | 0,_ -> []
                      | _,[] -> []
                      | b,t::r -> (b*t)::(aux b r) in
                  somme (aux a lst) (0::(produit q lst));;
```

Exercice II : E3A 2015

On souhaite analyser les résultats de sondages concernant une élection. Les candidats à l'élection sont numérotés de 0 à $k-1$. Les résultats de chaque sondage sont stockés dans un tableau : si le sondage a recueilli N réponses, le tableau comporte N cases, une pour chaque réponse : la i-ème case du tableau contient le numéro du candidat proposé par la i-ème personne sondée. On a ainsi un tableau T de longueur N qui contient des entiers naturels entre 0 et $k-1$.

Le sondage donne le candidat numéroté i élu si le nombre i est dans strictement plus de $N/2$ cases de T. Par exemple, un sondage correspondant au tableau $[|2;4;5;0;4;4;4|]$ donne le candidat 4 élu. Mais un tableau ne donne pas toujours un élu, par exemple le tableau $[|1;2;3;4;6;2;3;3|]$.

On suppose qu'un entier k est défini.

1) a) Écrire une fonction `nb` de type `int array -> int -> int` telle que si `a` est un entier naturel et `tab` un tableau de taille N issu d'un tel sondage, `nb tab a` est le nombre de cases du tableau `tab` qui contiennent `a`.

Correction : Simple parcours de tableau avec gestion d'une référence pour compter le nombre de `a` rencontrés.

```
let nb tab a =
   let n=ref 0 in
   for i=0 to (Array.length tab -1) do
      if tab.(i)=a then incr n
      done;
   !n;;
```

b) Évaluer la complexité de l'appel de `nb` en fonction de N.

Correction : Il y a N itérations et chacune se fait en temps constant. Hors de la boucle, il y a un nombre constant d'opérations et la complexité est finalement $O(N)$.

c) En déduire une fonction `elu1` de type `int array -> int` telle que `elu1 tab` est l'entier donné élu par le tableau `tab` si celui-ci existe et -1 sinon.

Correction : L'idée est de chercher successivement si $0, 1, 2, \ldots$ est élu en s'arrêtant quand on trouve un élu ou quand on arrive à la personne k. La fonction prend en argument le tableau. Noter qu'en Caml $N/2$ est une division entière. Cependant, pour x entier, les conditions $x \leq N/2$ et $x \leq \lfloor N/2 \rfloor$ sont équivalentes.

```
let elu1 tab k =
    let N=Array.length tab in
    let a=ref 0 in
    while !a<k && ((nb tab !a)<=N/2) do incr a done ;
    if !a=k then (-1) else !a ;;
```

REMARQUE : on pourrait aussi créer la liste des éléments présents dans le tableau, cela évite de tester tous les candidats, et tester chaque élément de la liste. Voici la fonction associée. `inserer` est une fonction d'insertion dans une liste triée. `creer_liste` renvoie la liste ordonnée des candidats. `parcours` teste les candidats de la liste.

```
let elu1 tab =
    let n=Array.length tab in
    let rec inserer kk lst =
              match l with
             | [] -> [kk]
             |x::q -> if x=kk then lst
          else if x<kk then x::(inserer kk q)
          else kk::l;;
    let rec creer_liste i =
       if i=n then []
       else inserer tab.(i) (creer_liste (i+1))
    in
    let rec parcours lst =
        match lst with
        | [] -> -1
        |a::q -> if nb tab a <= n/2 then parcours q
                 else a
    in
    parcours (creer_liste 0);;
```

d) Évaluer la complexité de votre algorithme en fonction de N et de k.

Correction : Il y a au plus k itération et chacune a un coût $O(N)$. Hors de la boucle, il y a un nombre constant d'opérations. Finalement, la complexité de la fonction est $O(kn)$.

2) On se propose d'utiliser la stratégie diviser pour régner pour déterminer l'éventuel élu donné par un tableau. On dispose de deux fonctions, qu'on ne cherchera pas à décrire :

- `miGauche : int array -> int array` qui prend en argument un tableau `tab` de longueur $N \geqslant 2$ et retourne le tableau de longueur $\lfloor N/2 \rfloor$ formé par les $\lfloor N/2 \rfloor$ premières cases de `tab`,
- `miDroite : int array -> int array` qui prend en argument un tableau `tab` de longueur $N \geqslant 2$ et retourne le tableau de longueur $N - \lfloor N/2 \rfloor$ formé par les $N - \lfloor N/2 \rfloor$ dernières cases de `tab`.

a) Soit `tab` un tableau de longueur $N \geqslant 2$. Démontrer que si `tab` donne a comme élu alors celui-ci est aussi donné élu par le tableau `miGauche tab` ou par le tableau `miDroite tab`.

Correction : Scindons `tab` (de taille n) en deux sous-tableaux de tailles `ta` et `tb` (de tailles n_a et n_b avec $n_a + n_b = n$). Si x n'est élu ni pour `ta` ni pour `tb` alors son nombre d'apparitions dans `tab` est inférieur ou égal à $\frac{(n_a+n_b)}{2} = \frac{n}{2}$. x n'est donc pas élu pour `tab`. On obtient le résultat demandé en contraposant et en prenant le cas particuliers des demi-tableaux à droite et gauche.

b) Proposer une fonction `elu2`, utilisant la stratégie diviser pour régner, de type `int array -> int * int` ; si `tab` est un tableau, `elu2 tab` est le couple `(a,n)` si l'entier `a` est donné élu par `tab` et apparait dans `n` cases exactement de `tab`, et `(-1,0)` si `tab` ne donne pas d'élu. On pourra utiliser la fonction `nb` définie en **1)a)**.

Correction : Un appel récursif sur chaque demi-tableau nous donne les seuls éligibles. Il suffit donc de vétrifier si ces éligibles atteignent effectivement la majorité. D'après la

question précédente s'il n'y a pas d'éligibles dans les deux demi tableaux alors il n'y a pas d'élus. On choisit comme cas de base celui où il n'y a qu'une seule case (qui est évident : le seul candidat est élu et il a une voix).

```
let rec elu2 tab =
   let n=Array.length tab in
   if n=1 then (tab.(0),1)
   else begin
       let tabg=(miGauche tab) and tabd=(miDroite tab) in
       let (xg,ng)=elu2 tabg and (xd,nd)=elu2 tabd in
       if xg = -1 && xd = -1 then (-1,0) else begin
             let mg=nb tabd xg and md=nb tabg xd in
              if mg+ng>n/2 then (xg,mg+ng)
              else if md+nd>n/2 then (xd,md+nd)
                   else (-1,0)
             end
       end;;
```

c) Évaluer la complexité de cette fonction en fonction de N.

Correction : Notons C_p le nombre maximal d'opérations effectué par la fonction quand on l'appelle avec un tableau de longueur $\leq 2^p$. On a alors

$$C_0 = O(1) \quad \text{et} \quad \forall p \geq 1,\ C_p = 2C_{p-1} + O(2^p)$$

le $O(2^p)$ couvrant les appels aux fonctions `miGauche`, `miDroite` et `nb`. On en déduit que

$$\frac{C_p}{2^p} = \frac{C_{p-1}}{2^{p-1}} + O(1)$$

puis que $\frac{C_p}{2^p} = O(p)$ ou encore $C_p = O(p2^n)$. En revenant à la variable N, on en déduit que la complexité de la fonction est

$$O(N \log_2(N))$$

c'est à dire quasi-linéaire en fonction de la taille du tableau.

Remarque : on peut aussi directement utiliser le théorème maitre avec la relation de récurrence $C_N = C_{\lfloor N/2 \rfloor} + C_{\lceil N/2 \rceil} + \Theta(N)$, mais c'est maintenant hors programme.

3) Soit T un tableau de longueur N. On dit que le nombre entier a est un *postulant* pour la valeur n du tableau T si : n est un entier strictement supérieur à $N/2$ tel que a apparait au plus (au sens large) n fois dans T et tout entier b distinct de a, apparait au plus (au sens large) $N - n$ fois dans T. Par exemple, 3 est un postulant pour $n = 5$ du tableau $[|1; 2; 3; 4; 3; 2; 3; 3|]$.

On dit que le nombre entier a est un postulant du tableau T s'il existe un nombre entier $n > N/2$ tel que a est un postulant pour la valeur n du tableau T.

a) Démontrer que si le tableau T donne a élu alors a est un postulant de T.

Correction : Supposons que le tableau T (de taille N) donne l'élu a et notons n le nombre des apparitions de a dans T. Par définition, on a $n > N/2$, a apparaît au plus (et même exactement) n fois dans T et tout autre élément apparaît au plus $N - n$ fois (puisque $N - n$ est le nombre de positions où a n'est pas). a est donc postulant de T (pour cette valeur n).

b) Démontrer que si a est un postulant de T, alors aucun autre élément de T ne pourrait être donné comme élu.

Correction : Supposons que a soit un postulant du tableau T (de taille N) associé à la valeur n. Si $b \neq a$ alors b apparaît au plus $N - n$ fois et comme $n > N/2$, b apparait au plus $N/2$ fois et n'est donc pas élu. a est donc le seul élément qui a une chance d'être élu.

c) Donner un exemple de tableau qui contient un postulant mais ne donne aucun élu et un exemple de tableau n'ayant aucun postulant.

Correction : Dans le tableau $[|1; 1; 2; 3|]$, on a $N = 4$ et donc $N/2 = 2$. 1 apparaît au plus 3 fois et tout autre élément apparaît au plus 1 fois. 1 est donc postulant pour la valeur 3 mais n'est cependant pas élu.
Dans le tableau $[|1; 1; 2; 2|]$, on a encore $N/4$. Un postulant a doit être associé à $n = 3$ ou $n = 4$. Mais dans ce cas, l'autre élément doit apparaître au plus 0 ou 1 fois ce qui n'est pas le cas. Il n'y a donc pas de postulant.

d) Soit T un tableau de longueur un entier pair N. On note TG le tableau de longueur $N/2$ formé par les $N/2$ premières cases de T et TD le tableau de longueur $N/2$ formé par les $N/2$ dernières cases de T.

i) On suppose que le tableau TD ne donne pas d'élu. Soit a un postulant pour la valeur l du tableau TG. Démontrer que a est un postulant de T. On exprimera la valeur d'un entier n tel que $n > N/2$ et a est un postulant pour la valeur n du tableau T en fonction de l et N.

Correction : Posons $n = l + \lfloor (N+2)/4 \rfloor$ et montrons que a est postulant pour T associé à la valeur n. Pour cela, on note qu'un élément apparaît dans TD au plus $\lfloor N/4 \rfloor$ fois (sinon il serait élu) et que $l > N/4$ (définition d'un élu).
* a apparaît donc au plus $l + \lfloor N/4 \rfloor$ fois T (on ajoute les apparitions dans TG et TD) et donc a fortiori au plus n fois
* un élément $b \neq a$ apparaît au plus $(N/2 - l) + \lfloor N/4 \rfloor$ fois dans T et il suffit de montrer que ceci est inférieur à $N - n$ c'est à dire que $N/2 \geq \lfloor N/4 \rfloor + \lfloor (N+2)/4 \rfloor$ ce que l'on justifie en traitant les cas $N = 4k$ ou $N = 4k + 2$
* $n = l + \lfloor (N + 2)/4 \rfloor$. Si $N = 4k$ alors $l > k$ et $n > 2k = N/2$. Si $N = 4k + 2$ alors $l \geq k + 1$ et $n \geq 2k + 2 > N/2$.

ii) Soient a un postulant pour la valeur l du tableau TG et b un postulant pour la valeur m du tableau TD.

A) On suppose que $a = b$. Démontrer que a est un postulant de T. On exprimera la valeur d'un entier n tel que $n > N/2$ et a est un postulant pour la valeur n du tableau T en fonction de l et m.

Correction : Si $a = b$, on pose $n = l + m$. Comme $l, m > N/4$, $n > N/2$. De plus, a apparaît au plus $n = l + m$ fois. Enfin, si $x \neq a$ alors le nombre d'apparition de x dans T est majoré par $(N/2 - l) + (N/2 - m) = N - n$. a est donc postulant de T pour la valeur $l + m$.

B) On suppose que $a \neq b$ et $m > l$. Démontrer que b est un postulant pour la valeur $N/2 + m - l$ de T.

Correction : On suppose $a \neq b$ et $m > l$. Posons $n = \frac{N}{2} + m - l$ et montrons que b est postulant de T pour cette valeur.
* On a bien $n > m/2$ puisque $m > l$.
* b apparaît au plus $(N/2 - l) + m$ fois dans T (au plus $N/2 - l$ fois dans TG et m fois dans TD).
* Si $x \notin \{a, b\}$, x apparaît au plus $(N/2 - l) + (N/2 - m) = N - (l + m)$ fois

dans T et cette quantité est plus petite que $N-n$ car $N/4 \leq l$ (ce qui donne facilement $N-(l+m) \leq N-n$).
$*$ a apparaît au plus $l+(N/2-m) = N-n$ fois dans T.

C) On suppose que $a \neq b$ et $m = l$. Démontrer que T ne donne pas d'élu.

Correction : On suppose $a \neq b$ et $m = l$. S'il existait un élu pour T, il serait élu pour TG ou TD (question 2) et donc serait égal à a ou b (question 3.b). Par symétrie des rôles, on peut supposer qu'il s'agirait de a. Mais a apparaît au plus $l+(N/2-m) = N/2$ fois dans T. On a donc une contradiction.

4) Écrire une fonction `postulant : int array -> int * int` telle que, si `tab` est un tableau d'entiers naturels de longueur N, `postulant tab` est un couple `(a,n)` tel que
- lorsque `postulant tab` renvoie le couple `(-1,0)`, le tableau `tab` n'a pas d'élu,
- lorsque `postulant tab` renvoie le couple `(a,n)` avec `n` $> N/2$, `a` est un postulant pour la valeur `n` du tableau `tab`. On supposera pour simplifier que la taille du tableau est une puissance de 2, La procédure utilisera la stratégie diviser pour régner et aura une complexité linéaire, ce qu'on justifiera précisément.

Correction : Il s'agit d'utiliser les questions précédentes. On peut en effet déduire l'existence et la valeur d'un postulant pour le tableau à partir d'une connaissance similaire sur les tableaux gauche et droit. En effet :
- si il n'y a aucun postulant à droite et gauche alors il n'y a pas d'élu ni à droite ni à gauche et donc pas d'élu pour le tableau et donc pas de postulant
- s'il y a un postulant d'un seul côté, $d.i$ donne un postulant du tableau entier
- s'il y a un postulant des deux côtés, $d.ii$ donne un postulant global ou montre qu'il n'en existe pas (cas C où il n'y a pas d'élu)

```
let rec postulant tab =
   let n=Array.length tab in
   if n=1 then (tab.(0),1)
   else begin
      let tabg=(miGauche tab) and tabd=(miDroite tab) in
      let (xg,ng)=postulant tabg and (xd,nd)=postulant tabd in
      if xg=(-1) && xd=(-1) then (-1,0)
      else if xd=(-1) then (xg,ng+(n+2)/4)
      else if xg=(-1) then (xd,nd+(n+2)/4)
      else if xg=xd then (xg,ng+nd)
      else if ng>nd then (xg,n/2+ng-nd)
      else if nd>ng then (xd,n/2+nd-ng)
      else (-1,0)
      end;;
```

Autre version :

```
let   postulant tab=(* version dont on contr\^ole la complexit\'e*)
             let rec aux t  d f=
            (*fonction qui cherche un postulant sur une portion de tableau*)
             let n= f- d in match n with
                       |0-> (-1,0)
                       |1-> (tab.(d),1)
                       | _-> let (dg,fg) =(d,d+n/2)  in
                             let (dd,fd) =(d,d+n/2) in
                             let (a,l)=aux t d (d+n/2) in
                             let (b,m)=aux t (d+n/2) f in
                             if a=b then (a, m+l)
          (*valable même si  TD et TG répondent "pas d'élu" cf 3diiA et2a*)
                             else if l*m=0 then
                             (max a b,  n/2 +  l+ m  - n/4)
                             (* TD ou bien TG répond "pas d'élu" cf 3di*)
                             else if  m>l then (b , n/2 + m - l) (*3diiB*)
                             else if  m<l then (a , n/2 - m + l) (*3diiB*)
                             else (-1,0) (*3diiC*);
              in aux tab 0  (Array.length tab);;
```

Soit C_p le nombre d'opérations pour un tableau de longueur 2^p. On a $C_0 = 2$ et pour tout p , $C_{p+1} = 2C_p + A$ où A est une constante donc $C_p \sim 2^p$ (suite arithmético géométrique strictement croissante)

donc | la complexité de `postulant` est bien linéaire |

5) En déduire une fonction `elu3`, de type `int array -> int` qui prend en argument un tableau `tab` et retourne l'entier élu de `tab` si celui-ci existe et -1 sinon, de complexité linéaire.

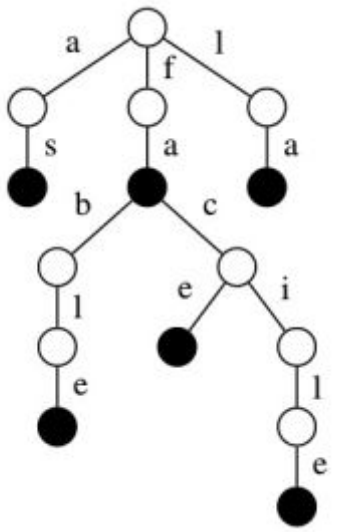
Correction :

```
let elu3 tab=let (a,n) = postulant tab in
              if 2 * (nb tab a) > (Array.length tab) then a
              else -1;;
```

Exercice III : Arbres lexicographiques

On représentera les mots sous Caml non pas par le type string mais une liste de caractères.
`type mot = char list;;`
Exemple : L'expression `[ 'f'; 'a'; 'c'; 'e' ]` est associée au mot « face », de longueur 4.

Les arbres lexicographiques sont des arbres utilisés pour représenter des ensembles de mots. Les **arêtes** sont étiquetés par un caractère. La séquence des caractères qui étiquettent les arêtes le long d'un chemin de la racine de l'arbre jusqu'à un noeud forme donc un mot. Certains noeuds sont appelés noeuds « terminaux ». Les mots représentés par l'arbre sont les mots obtenus en suivant un chemin de la racine jusqu'à un noeud terminal. Attention : un noeud terminal n'est pas forcément une feuille.



`Figure 1` Un exemple d'arbre lexicographique représentant l'ensemble {« as », « fa », « fable », « facile », « la »}. Les noeuds terminaux, respectivement non terminaux, sont de couleur noire, respectivement blanche.

Un arbre lexicographique est représenté par le type `arbre_lex` et le type auxiliaire `fils` ci-dessous :

```
type arbre_lex = Noeud of bool * fils
 and fils = (char * arbre_lex) list;;
```

Dans l'appel `Noeud(terminal, fils)`, `terminal` est un drapeau booléen qui indique si le noeud est terminal ou pas. Et `fils` contient la liste des fils ainsi que les étiquettes des arêtes qui y mènent. C'est une liste de couples de la forme `(c, f)` tel que `c` est l'étiquette de l'arête menant au fils `f`.

En outre, un arbre lexicographique sera dit *valide* si pour chaque noeud, les fils sont rangés dans l'ordre alphabétique des étiquettes des arêtes qui y mènent, et si toutes les feuilles sont des noeuds terminaux.
Sauf mention du contraire, on supposera que tous les arbres lexicographiques utilisés sont valides, et dans les questions demandant de créer un arbre, on veillera à créer un arbre valide.

Remarque : Ainsi les mots dans un arbre lexicographiques sont-ils rangé dans l'ordre alphabétique. Un tel arbre est donc semble à un arbre binaire de recherche, à la différence près qu'il n'est pas binaire, puisqu'un noeud peut avoir autant de fils qu'il y a de lettres dans l'alphabet choisi.

1) Écrire une fonction `mot_of_string` de type `string -> mot` prenant en entrée une chaîne de caractère, de type `string` et la convertissant en une liste de lettres. On pourra écrire une fonction récursive auxiliaire prenant un argument supplémentaire : la position actuelle dans la chaîne de caractères.

Correction :

```
let mot_of_string m = let n = String.length m in
    let rec aux i = if i >= n then []
                    else m.[i]::aux (i+1)
    in aux 0;;
```

2) L'expression suivante définit un arbre exemple :

```
let feuille= Noeud(true,[]);;
let exemple=
        Noeud(false,[
           'f',Noeud(false,[
                'a', Noeud(true,[
                     ('c', feuille)
                               ]
                     );
                ('i', feuille)
                           ])
               ]);;
```

Dessiner l'arbre lexicographique correspondant, et donner l'ensemble de mots qu'il représente.

Correction :

l'ensemble des mots est {fa,fac,fi}

3) Dans quelle situation un noeud terminal qui n'est pas une feuille est-il utile ?

Correction : Lorsqu'un mot est préfixe d'un autre

4) Dans quelle situation la racine est-elle un noeud terminal ?

Correction : Lorsque le mot vide est dans l'ensemble de mots à représenter.

5) Écrire une fonction `arbre_of_mot` de type `mot -> arbre_lex` telle que l'appel `(arbre_of_mot m)` sur un mot `m` renvoie un arbre lexicographique contenant uniquement le mot m.

Correction :

```
let rec arbre_of_mot lst = match lst with
      |[]-> feuille
      |t::q -> Noeud(false, [(t, arbre_of_mot q)]);;
```

6) Écrire une fonction `nb_mots` de type `arbre_lex -> int` telle que l'appel `(nb_mots a)` sur un arbre lexicographique `a` renvoie le nombre de mots contenus dans l'arbre a. L'algorithme utilisé ne devra parcourir qu'une seule fois l'arbre `a`. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.

Correction : Il s'agit tout simplement de compter le nombre de noeuds terminaux. On crée deux fonctions auxiliaires récursives pour traiter le cas d'un noeud et celui de la liste fils de ce noeud.

```
let nb_mots arb =
 let rec aux1 (Noeud(term, fils)) =
          if term then 1 + aux2 fils
```

```
          else aux2 fils
     and aux2 lst = match lst with
                    |[] -> 0
                    |(_,a)::q -> aux1 a + aux2 q
 in aux1 arb;;
```

7) Écrire également une fonction calculant le nombre de feuilles d'un arbre lexicographique.
Correction : on reprend le même principe que la question précédente en tenant compte de la définition d'une feuille.

```
let rec nb_feuilles arb =
         let rec aux1 arb = match arb with
               | Noeud(_,[]) -> 1
               | Noeud(_,fils)-> aux2 fils
            and aux2 lf =  match lf with
               |[] -> 0
               |(_,a)::q -> aux1 a + aux2 q
         in aux1 arb;;
```

8) Écrire une fonction `prefixed` prenant un caractère `x` et une liste de mots `lst` et renvoyant la liste obtenue en ajoutant `x` devant chaque mot de `lst`.
Correction : une solution immédiate est :

```
let prefixed x (lst:mot list)=
  (* Renvoie la liste contenant les mots de lst prefixée d'un x. *)
  List.map (fun m-> x::m) lst;;
```

Mais cela suppose connaitre la fonction `List.map`.
Voici une version qui reconstruit cette fonction :

```
let prefix x (lst : mot list) =
     let rec aux lst = match lst with
           | [] -> [] (*fin de la liste*)
           | m::q -> (x::m)::(aux q)
      in aux lst;;
```

9) Écrire une fonction `mots_of_arbre` de type `arbre_lex -> mot list` telle que l'appel `(mots_of_arbre a)` sur un arbre lexicographique `a` renvoie la liste des mots contenus dans `a`. L'algorithme utilisé ne devra parcourir qu'une seule fois l'arbre `a`. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.
Correction : on construit la liste des fils d'un noeud avant d'y ajouter la lettre de l'arête.

```
let rec mots_of_arbre arb=
    let rec aux1 arb = (* Renvoie la liste des mots contenus dans a. *)
           match arb with
             |Noeud(true, fils)-> [] :: aux2 fils
             (* le mot vide est contenu dans  arb*)
             |Noeud(false, fils)-> aux2 fils
        and aux2 f=  match f with
              |[]-> []
              |(lettre, a)::q -> (prefix lettre (aux1 a)) @ aux2 q
     in aux1 arb;;
```

Il est difficile de se passer de la concaténation des listes ici.

10) Écrire une fonction `appartient` de type `mot -> arbre_lex -> bool` telle que l'appel `(appartient m a)` sur un mot `m` et un arbre lexicographique valide `a` renvoie la valeur `true` si et seulement si l'arbre `a` contient le mot `m`. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives. La complexité de la recherche d'un mot `m` doit être linéaire en la longueur du mot. On justifiera cette complexité.

Correction :

```
let appartient (m:mot) a =
        let aux1 m a = (* Indique si le mot m est contenu dans l'arbre a. *)
                match m, a with
                  |[], Noeud(term,_) -> term
                  (* Le mot vide est reconnu ssi la racine est terminale *)
                  (* Une fois le mot trouvé il faut avoir un noeud terminal*)
                  |t::q, Noeud(_,fils)-> aux2 t q fils

            and aux2 lettre mot f = (*on cherche les lettres du mot dans l'arbre*)
                 match f with
                   |[] -> false
                   |(x,a)::q when x=lettre -> aux1 mot a
                   |(x,_)::q when x<lettre -> aux2 lettre mot q
                   (* on utilise le fait que les lettres sont dans l'ordre*)
                   | _ -> false
     in aux 1 m a ;;
```

Le nombre d'appels à la fonction auxiliaire `aux1` correspond au nombre de lettres du mot. Pour chaque lettre du mot la fonction `aux2` est appelée autant de fois qu'il y a de fils soit au maximum 26 (lettres). La complexité est donc bien en la longueur du mot.

11) Écrire une fonction `ajout` de type `mot -> arbre_lex -> arbre_lex` prenant en entrée un mot `m` et un arbre lexicographique `a` et renvoyant l'arbre obtenu en rajoutant `m` dans `a`.

Correction :

```
let  ajout m a=
   let rec aux1 m a = match m, a with
               |[], Noeud(_, fils)-> Noeud(true,fils)
               (*fin du mot : on met un noeud terminal*)
     |t::q, Noeud(term, fils) -> Noeud(term,aux2 t q fils)

       and aux2 lettre mot f =  match f with
               |[] -> [(lettre, arbre_of_mot mot)]
               (*ajout d'un fils*)
               |(x,a)::q when lettre <x -> (lettre, arbre_of_mot mot)::f
               (*ajout d'un fils avant la lettre existante*)
               |(x,a)::q when lettre=x -> (x, aux1 mot a)::q
               (*la lettre existe déjà*)
               |(x,a)::q -> (x,a):: (aux2 lettre mot q)
    in aux1 m a;;
```

12) En déduire une fonction `arbre_of_list` prenant une liste de mots et renvoyant un arbre lexicographique contenant les mêmes mots.

Correction : on utilise la fonction précédente en parcourant la liste. On initialise avec le premier mot à l'aide de la fonction `arbre_of_mot`.

```
let  arbre_of_list lst =
     let rec aux lst arb = match lst with
         | [] -> arb
         | m::q -> let arb1 = ajout m arb in aux q arb1
     in if lst = [] then feuille
              else aux (List.tl lst) (arbre_of_mot (List.hd lst));;
```