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

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 $P(X)$.
- **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 $P(X)$.
- **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`.
- **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.*
- **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)
- **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)

Exercice II

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`.

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

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

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`.

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)**.

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

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.

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

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.

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.

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.

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

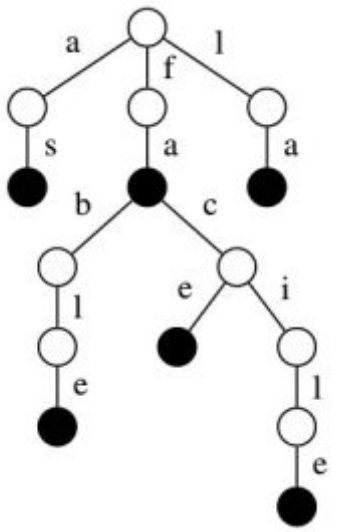
4) Écrire une fonction `postulant : int array -> int * int` telle que, si `tab` est un tableau d'entiers naturels de longueur N, `postulant tab` est un couple `(a,n)` tel que
- lorsque `postulant tab` renvoie le couple `(-1,0)`, le tableau `tab` n'a pas d'élu,
- lorsque `postulant tab` renvoie le couple `(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.

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.

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.

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.

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

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

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.

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.

7) Écrire également une fonction calculant le nombre de feuilles d'un arbre lexicographique.

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`.

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.

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.

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`.

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

Annexe

Pour tous les exercices

Aucune fonction sur les listes prédéfinie n'est autorisée à par, si besoin, `List.hd`, `List.tl`, le constructeur `::`.

Exercice 2 :

Pour les tableaux :
- `Array.make n a` : créer un tableau de taille n dont tous les éléments sont égaux à `a`
- `Array.length` : donne la longueur d'un tableau
- `p.(i)` : donne l'élément du tableau d'indice `i` (l'indice commence à 0)
- `p.(i)<- a` : pour mettre la valeur de `a` dans le tableau à l'indice `i`