Concours Centrale - Supélec 2008

Épreuve : INFORMATIQUE **Filière** MP

Les deux parties sont indépendantes. Le candidat indiquera en tête de sa copie le langage choisi : Caml ou Pascal.

Partie I - Mots de Lukasiewicz

Dans cette partie, les mots considérés sont pris sur l'alphabet $\{-1, +1\}$.

On appelle *mots de Lukasiewicz* les mots $u = (u_1, u_2, \ldots, u_n)$ sur cet alphabet qui vérifient les deux propriétés suivantes :

$$\sum_{i=1}^{n} u_i = -1 \quad \text{et} \quad \sum_{i=1}^{k} u_i \geqslant 0 \text{ pour } 1 \leqslant k \leqslant n-1.$$

On note avec un point $\cdot$ la concaténation de deux mots, par exemple $a \cdot b$.

- En Caml, un mot de Lukasiewicz sera représenté par une liste d'entiers (`int list`).
- En Pascal, un mot de Lukasiewicz sera représenté par une chaîne de caractères (`string`) composée de symboles `'+'` (pour $+1$) et `'-'` (pour -1). On rappelle les opérations suivantes :
  - accéder au caractère numéro i de la chaîne c : `c[i]` (la numérotation commence à 1) ;
  - obtenir la longueur d'une chaîne c : `length(c)` ;
  - extraire une chaîne d'une sous-chaîne : `copy(chaîne, position, longueur)`. Par exemple, `copy('abcdef', 2, 3)` retourne `'bcd'` ;
  - concaténer deux chaînes ou caractères c_1 et c_2 pour obtenir une nouvelle chaîne : `c1 + c2`.

On demande d'indiquer le type de toutes les fonctions écrites, que ce soit en Caml ou en Pascal.

I.A - Quelques propriétés

I.A.1) Donner tous les mots de Lukasiewicz de longueur 1, 2 et 3, puis tous ceux de longueur paire.

I.A.2) Écrire une fonction qui indique si un mot est de Lukasiewicz. Cette fonction renverra une valeur booléenne. La fonction proposée devra impérativement avoir une complexité (en termes d'opérations élémentaires) en $O(n)$, avec n la longueur du mot d'entrée.

I.A.3) Montrer que si u et v sont des mots de Lukasiewicz, alors $(+1) \cdot u \cdot v$ est un mot de Lukasiewicz.

I.A.4) Réciproquement, montrer que tout mot de Lukasiewicz de longueur supérieure ou égale à 3 admet une décomposition unique de la forme $(+1) \cdot u \cdot v$, où u et v sont des mots de Lukasiewicz.

I.A.5) Écrire une fonction `décompose` qui associe ce couple (u, v) à un mot de Lukasiewicz de longueur supérieure ou égale à 3. *En Pascal, on pourra écrire une procédure qui modifie deux de ses paramètres u et v passés par référence. On ne demande pas de traiter les cas où le mot fourni en entrée ne serait pas de Lukasiewicz.*

I.A.6) On souhaite calculer tous les mots de Lukasiewicz d'une longueur donnée. Comparer les avantages d'une solution récursive appliquant le principe de la décomposition suggéré par la question A.4), et celle d'une solution appliquant le même principe, mais pour laquelle on tabulerait les résultats intermédiaires.

I.A.7) Écrire une fonction `obtenirLukasiewicz` qui calcule la liste des mots de Lukasiewicz de taille inférieure ou égale à un entier donné.

En Pascal, on dispose des types et fonction :

```
type
  ptliste=^liste;
  liste=record
          valeur : string;
          suivant:ptliste
        end;
function adjonction(x:string; pt:ptliste):ptliste;
```

Cette dernière fonction `adjonction` réalise l'adjonction d'une nouvelle cellule en tête de liste (et retournant un pointeur sur ladite cellule). On ne demande pas de l'écrire.

I.B - Dénombrement

I.B.1) Soit $u = (u_1, \ldots, u_n)$ un mot tel que $\sum_{i=1}^{n} u_i = -1$. Démontrer qu'il existe un unique entier i, $1 \leqslant i \leqslant n$, tel que $(u_i, u_{i+1}, \ldots, u_n, u_1, \ldots, u_{i-1})$ soit un mot de Lukasiewicz. Ce mot est appelé *conjugué* de u.

I.B.2) Écrire une fonction `conjugue` qui calcule le conjugué d'un mot $u = (u_1, \ldots u_n)$ vérifiant $\sum_{i=1}^{n} u_i = -1$.

I.B.3) En utilisant les résultats précédents, déterminer le nombre de mots de Lukasiewicz de longueur $2n+1$.

On pourra utiliser ce résultat admis : « Si u et v sont deux mots non vides, les deux propositions suivantes sont équivalentes :

- $uv = vu$
- il existe un mot w non vide et deux entiers $k, \ell \geqslant 1$ tels que $u = w^k$ et $v = w^\ell$ »

I.C - Régularité

I.C.1) Montrer que le langage $L = \left\{(+1)^n(-1)^{n+1}, n \in \mathbb{N}\right\}$ n'est pas reconnaissable (implicitement dans la suite : « par un automate fini »).

I.C.2) Montrer que l'intersection de deux langages reconnaissables est un langage reconnaissable.

I.C.3) Prouver le caractère reconnaissable ou non du langage constitué des mots de Lukasiewicz.

I.D - Capsules

On appelle *capsule* d'un mot u tout facteur de u de la forme $(+1,-1,-1)$.

On définit sur $\{-1,1\}^*$ une fonction ρ dite de *décapsulage* :

$$\begin{cases} \rho(u) = u, \text{ si } u \text{ ne contient pas de capsule} \\ \rho(u) = (u_1, \ldots, u_{i-1}, u_{i+2}, \ldots u_n), \\ \qquad\qquad \text{si } (u_i, u_{i+1}, u_{i+2}) = (+1,-1,-1) \text{ est la première capsule de } u \end{cases}$$

I.D.1) Justifier le fait que la suite $\left(\rho^n(u)\right)_{n\in\mathbb{N}}$ est constante au delà d'un certain rang. La valeur limite de la suite $\left(\rho^n(u)\right)_{n\in\mathbb{N}}$ est notée $\rho^*(u)$.

I.D.2) Écrire une fonction `rho` qui calcule $\rho(u)$.

I.D.3) Écrire une fonction `rhoLim` qui calcule $\rho^*(u)$.

I.D.4) Démontrer que u est un mot de Lukasiewicz si et seulement si $\rho^*(u) = (-1)$.

Partie II - Recherche de motif

Dans ce problème, nous allons étudier deux algorithmes de recherche de motif (en général noté p) dans un mot (en général noté m). Par exemple, le motif `p0=bra` apparaît deux fois dans le mot `m0=abracadabra`. Les programmes de recherche de motif devront retourner la liste (éventuellement vide) des positions (au sens de Caml/Pascal) du motif dans le mot. Dans l'exemple précédent, les programmes devront retourner une liste contenant les positions 1 et 8 en Caml ; 2 et 9 en Pascal. C'est la position de la première lettre qui est prise en compte. Plus formellement, si $m = m_1...m_n$, on dit que p apparaît en position i dans m lorsque $p = m_i...m_{i+|p|-1}$, avec $|p|$ la longueur de p.

II.A - Algorithme naïf

II.A.1) Écrire une fonction `coincide` prenant en entrée deux chaînes de caractères `p` et `m`, une position `pos`, et retournant `true` si `p` apparaît en position `pos` dans `m` (et `false` sinon). *Cette fonction devra uniquement utiliser des comparaisons de caractères, sans utiliser* `sub_string` *(Caml) ou* `copy` *(Pascal). De plus, en cas de réponse* `false`*, elle devra arrêter les tests dès que possible.*

II.A.2) Écrire une fonction `recherche` prenant en entrée deux chaînes de caractères `p` et `m`, et retournant la liste (dans n'importe quel ordre, et éventuellement vide) des positions de `p` dans `m`.

En Pascal, on dispose comme dans la première partie des types `ptliste,liste` *et de la fonction* `adjonction`*, avec cette fois le champ* `valeur` *de type* `integer`*.*

II.A.3) Évaluer la complexité (en termes de comparaisons de caractères, et en fonction de $|p|$ et $|m|$) de la fonction précédente dans le pire des cas. On exhibera un cas défavorable (en terme de complexité), avec p et m arbitrairement grands.

II.B - Algorithme de Rabin-Karp

La présentation de l'algorithme de Rabin-Karp est faite dans le cas de l'alphabet $A_0 = \{0,1,...,9\}$.

Un mot $m \in A_0^*$ peut être vu naturellement comme un entier ($m = 366$ est dans A_0^*... mais aussi dans $\mathbb{N}$).

II.B.1) La première idée de l'algorithme de Rabin-Karp est que si on cherche $p = 366$ dans $m = 97463667305$, on va regarder les lettres de m par groupes de 3, en initialisant un compteur à $c = 974$, et en « avançant » dans m en ajoutant à chaque fois une nouvelle lettre, et en effaçant la première de c. Dans notre exemple, c passe d'abord à 746 puis 463. Plus formellement, lire la lettre $\ell = m_{i+|p|}$ dans m en effaçant m_i change c en $10c + \ell - 10^{|p|}m_i$. Si $c = p$, cela signifie que le motif p est présent en position i dans m.

On suppose dans cette première version de l'algorithme de Rabin-Karp que p est de très petite longueur, de sorte que le compteur c ne dépassera jamais la valeur du plus grand entier autorisé par Caml ou Pascal.

On considère définie une fonction appelée `numeral` prenant comme entrée un caractère de l'alphabet A_0 et retournant la valeur entière correspondante (par exemple on associe l'entier 0 au caractère `'0'`) :

- En Caml, `numeral : char -> int = <fun>`
- En Pascal, `function numeral(a : char) : integer ;`

a) Écrire une fonction prenant en entrée un mot m, une longueur ℓ, et retournant la valeur initiale du compteur, calculée en lisant les $\ell = |p|$ premières lettres de m. Dans l'exemple donné plus haut, sur l'entrée $(m, 3)$, la fonction doit retourner 974.

b) Écrire enfin une fonction prenant en entrée un motif, un mot (supposé de taille supérieure au motif), et calculant la liste des positions dans m où le motif p est présent.

II.B.2) L'hypothèse quant à la longueur « faible » de p étant très restrictive, on modifie l'algorithme précédent en choisissant un entier q modulo lequel on calculera c. En lisant la lettre $\ell = m_{i+|p|}$ et en effaçant m_i, le nouveau compteur devient donc $c' \leftarrow \left(10c' + \ell - 10^{|p|} m_i\right) \ [q]$.

Lorsque $c = p$, on a $c' \equiv p \ [q]$. Mais réciproquement, lorsque $c' \equiv p \ [q]$, on n'est pas assuré d'avoir $c = p$. On regarde alors (avec l'algorithme naïf) si le facteur de m correspondant au compteur c calculé est égal à p. Si $c' \equiv p \ [q]$ mais $c \neq p$, on parle de « fausse-position ».

a) Donner les valeurs successives de c' lors de la recherche de $p = 366$ dans $m = 97463667305$, avec $q = 9$.

b) Écrire une fonction recherchant les positions d'un motif dans un mot, en appliquant l'algorithme de Rabin-Karp, avec un entier q donné en paramètre.

c) Lors de la recherche de $p = 0001000$ dans $m = 000000000$ avec $q = 1000$, combien de cas de fausse-position va-t-on rencontrer ?

d) Majorer le temps de calcul de la liste des positions de p dans m, en fonction de $|p|$ et $|m|$ avec l'algorithme de Rabin-Karp.

q est supposé tel que les calculs arithmétiques modulo q sont d'un coût constant. Le temps de calcul pendra donc en compte ces opérations arithmétiques, et les comparaisons de caractères, du type $m_i = p_j$.

e) Comparer les complexités dans le pire des cas de l'algorithme de Rabin-Karp et de l'algorithme naïf.

f) En pratique, aura-t-on intérêt à prendre q plutôt petit ou plutôt grand ? Que peut-on alors espérer pour le temps de calcul de la recherche des occurrences de p dans m ?

On demande une justification informelle, le choix de l'entier q et l'évaluation de temps moyen de calcul étant deux choses très délicates ...

• • • FIN • • •
