Concours Centrale - Supélec 1997

Épreuve : INFORMATIQUE — Filière MP

Chaque candidat doit indiquer en début de sa copie le langage (CAML ou PASCAL) qu'il choisit pour rédiger les programmes demandés. Tous les programmes doivent être rédigés dans ce langage.

Partie I

L'objet de cette partie est l'étude de différentes méthodes de programmation qui répondent à la question : "Étant données deux chaînes de caractères `source` et `motif`, est-ce que `motif` est une sous-chaîne de `source` ?"

Dans toute cette partie, les tableaux sont indexés à partir de 0, aussi bien en PASCAL qu'en CAML.

I.A - Présentation

Les définitions et notations introduites ici sont valables dans toute la suite de la partie I.

Soit un alphabet A, ensemble fini de symboles aussi appelés lettres.

- A^* désigne l'ensemble des séquences finies, éventuellement vides, d'éléments de A, c'est-à-dire l'ensemble de tous les mots, éventuellement vides, sur A ;
- pour $w \in A^*$, $|w|$ désigne la longueur de w, c'est-à-dire le nombre de symboles composant w ;
- pour $w \in A^*$ et i entier $(0 \leq i < |w|)$, $w(i)$ désigne la $i+1^{ème}$ lettre de w ;
- pour $x, y \in A^*$, xy désigne le mot issu de la concaténation de x avec y ;
- pour un mot $z \in A^*$, l'ensemble sousmot(z) est défini par :

  $\text{sousmot}(z) = \{w \in A^* | (\exists z_1, z_2 \in A^*).\ z = z_1 w z_2\}$;

  si $w \in \text{sousmot}(z)$, on dira que w est un sous-mot de z ;
- pour deux mots $w, z \in A^*$, w est un préfixe de z si et seulement s'il existe $y \in A^*$ tel que $z = wy$; on dit alors que y est un suffixe de z.

Avec ce formalisme, la question étudiée s'écrit :

> Soient deux mots `source` et `motif` dans A^* avec $|\text{motif}| \leq |\text{source}|$,
> est-ce que `motif` $\in$ sousmot(source) ?

I.B - Méthode itérative

Dans la partie I.B, les mots sont représentés par des tableaux de caractères. L'algorithme itératif suivant permet de répondre à la question.

```
 1   RECHERCHE ITÉRATIVE BRUTE(motif, source).
 2   i <- 0
 3   j <- 0
 4   tant que i < |motif| et j < |source|
 5   faire si motif(i) = source(j)
 6      alors
 7         i <- i + 1
 8         j <- j + 1
 9      sinon
10         j <- j - i + 1
11         i <- 0
12   si i = |motif|
13      alors retourner vrai
14      sinon retourner faux
```

I.B.1) Donner un invariant de boucle. Indication : on pourra introduire l'indice `k` initialisé à 0 et correspondant à la mise à jour de `j` à la ligne 10.

I.B.2) Montrer la terminaison de l'algorithme.

I.B.3) Traduire l'algorithme en PASCAL ou en CAML en utilisant les définitions suivantes.

En PASCAL (il est nécessaire de borner la longueur des mots `source` et `motif` pour définir les types) :

```
Const
   N = <valeur maximale de |source| - 1> ;
   P = <valeur maximale de |motif| - 1> ;
Type
   T_SOURCE = ARRAY [0..N] OF CHAR ;
   T_MOTIF = ARRAY [0..P] OF CHAR ;
FUNCTION RECHERCHE_ITER_BRUTE (source:T_SOURCE;
      motif:T_MOTIF) : BOOLEAN ;
```

En CAML :

```
Recherche_iter_brute : 'a vect -> 'a vect -> bool = <fun>
```

I.B.4) Dans cette question, l'alphabet A possède uniquement deux lettres : $A = \{a, b\}$. Les mots `source` et `motif` ont pour longueurs respectives n et p. Déterminer la complexité de l'algorithme en nombre de comparaisons de caractères en fonction de n et p dans le pire cas. Donner un exemple où `source` et `motif` sont de longueurs respectives n et p et pour lesquels la complexité de l'algorithme est la plus défavorable.

I.C - Méthode récursive

Dans la partie I.C, les mot sont représentés par une structure de liste munie des opérations pré-définies suivantes :

ListeVide : liste -> booléen
Construit : lettre * liste -> liste
Tete : liste -> lettre
Queue : liste -> liste

Le raisonnement suivant permet de construire une méthode récursive :

- Le mot `motif` est identique à un sous-mot de `source` quand `motif` est un préfixe de `source` ou quand `motif` est identique à un sous-mot de `source` privé de son premier caractère.
- Le mot `motif` est un préfixe de `source` quand `motif` est vide ou quand il satisfait les deux conditions suivantes :
  - le premier caractère de `motif` est identique au premier caractère de `source` ;
  - `motif` privé de son premier caractère est un préfixe de `source` privé de son premier caractère.

I.C.1) Écrire la fonction de recherche récursive en PASCAL ou en CAML, en utilisant la fonction `Est_Prefixe` et les prototypes donnés ci-dessous. Prouver la terminaison de l'algorithme.

En PASCAL :

```
Type
   Liste = <type supposé défini>

FUNCTION ListeVide(L:Liste) : BOOLEAN ;
FUNCTION Tete(L:Liste) : CHAR ;
FUNCTION Queue(L:Liste) : Liste ;
FUNCTION Construit(c:CHAR; L:Liste) : Liste ;
FUNCTION Est_Prefixe(source:Liste; motif:Liste) : BOOLEAN ;
FUNCTION Recherche_Recursive(source:Liste; motif:Liste) : BOOLEAN ;
```

En CAML :

```
ListeVide : 'a list -> bool = <fun>
Tete : 'a list -> 'a= <fun>
Queue : 'a list -> 'a list = <fun>
Construit : 'a -> 'a list -> 'a list = <fun>
Est_Prefixe : 'a list -> 'a list -> bool = <fun>
Recherche_Recursive : 'a list -> 'a list -> bool = <fun>
```

I.C.2) Écrire la fonction `Est_Prefixe` en PASCAL ou en CAML en utilisant les prototypes de la question précédente.

I.C.3) Déterminer la complexité, en nombre de comparaisons de caractères, de cet algorithme dans le cas le plus défavorable. Comparer avec le résultat de la version itérative.

I.D - Algorithme simplifié de Knuth-Morris-Pratt

Dans la partie I.D, les chaînes de caractères sont à nouveau représentées par des tableaux et les définitions de type de la partie I.B sont utilisées.

Cette partie présente une première amélioration, découverte par Knuth, Morris et Pratt, de la méthode itérative. L'idée de base est la suivante. Dans la méthode proposée partie I.B, dès que le premier caractère de `motif` est identique au caractère courant de `source`, la course à la comparaison commence et continue tant que les caractères coïncident, balayant ainsi les préfixes successifs de `motif`. Appelons faux-départ le plus grand de ces préfixes quand une différence est détectée. Par exemple, supposons que `motif` = "abbbbbbc". Si un échec de comparaison est rencontré en `motif(5)`, le faux-départ est constitué du sous-mot "abbbb" qui se retrouve également dans `source`. Il est inutile de reprendre la recherche dans `source` avant l'indice courant, puisque `source` ne comprend pas d'autre a entre la position de départ et la position courante.

Cette situation est particulière. Dans le cas général, un suffixe de faux-départ peut coïncider avec un préfixe de `motif`. La course à la comparaison peut alors reprendre à partir de l'indice suivant le préfixe dans `motif` et partir de l'indice courant dans `source`.

Par exemple :

```
motif            a b c a a b c b b
source       ... a b c a a b c a ...
                                 ^
```

La caractère ^ indique l'indice courant dans `source`. Le faux départ est détecté et vaut "abcaabc". Le suffixe "abc" du faux-départ en est également un préfixe, donc est un préfixe de `motif`. La recherche peut continuer par :

```
motif                    a b c a a b c b b
source       ... a b c a a b c a ...
                                 ^
```

La position de reprise dans `motif` ne dépend que du mot `motif` et de l'indice dans `motif` lors de l'échec. Dans l'exemple précédent, l'échec a eu lieu à l'indice 7 dans `motif`, et l'indice de reprise vaut 3. Il est donc possible de prédéterminer les indices de reprise dans `motif` pour chaque position possible d'échec. Les indices de reprise seront stockés dans un tableau auxiliaire. Ce tableau auxiliaire, `aux`, de même longueur que `motif` est tel que `aux(i)` est l'indice de reprise dans `motif` en cas d'échec à l'indice `i`. Plus précisément, en cas d'échec lors de la comparaison de la lettre d'indice `i` dans `motif` et de la lettre d'indice `j` dans `source`, la comparaison reprendra à l'indice `aux(i)` dans `motif` et à l'indice `j` dans `source`. Néanmoins, si l'échec a eu lieu sur la première lettre de motif (`i=0`), il convient de reprendre la comparaison à l'indice 0 dans `motif` et à l'indice `j+1` dans `source`. Pour faciliter le traitement de ce cas particulier, on prendra `aux(0)=-1`. Le tableau auxiliaire associé à l'exemple précédent vaut donc (-1,0,0,0,1,1,2,3,0).

I.D.1) Écrire, en PASCAL ou en CAML, l'algorithme simplifié de Knuth, Morris et Pratt en supposant donnée la fonction `calcule_tab_aux` qui construit le tableau auxiliaire et en utilisant les prototypes ci-dessous.

En PASCAL :

```
Type
   T_AUX = ARRAY [0..P] OF INTEGRER ;
FUNCTION Recherche_KMP(source:T_SOURCE; motif:T_MOTIF;
      aux:T_AUX) : BOOLEAN ;
FUNCTION Calcule_Tab_Aux(motif:T_MOTIF) : T_AUX ;
```

En CAML :

```
recherche_KMP = 'a vect -> 'a vect -> int vect -> bool = <fun>
calcule_tab_aux = 'a vect -> int vect = <fun>
```

I.D.2) Dans cette question, A est un alphabet de trois lettres : $A = \{a, b, c\}$. Construire à la main le tableau auxiliaire pour le motif "abaababaabaab".

I.D.3) Écrire, en PASCAL ou en CAML, un algorithme qui construit le tableau auxiliaire en utilisant les prototypes de la question I.D.1. (Indication : il s'agit de rechercher à l'intérieur de `motif` les préfixes qui en sont également des sous-mots).

Partie II

On note S un dictionnaire constitué d'un ensemble de n mots, appelés clés, de même longueur l, sur un alphabet v constitué des symboles $0, 1, 2, \ldots k-1$. Ces clés permettent d'accéder à une information. Les opérations sur le dictionnaire sont :

recherche : clé * dictionnaire -> booléen
insertion : clé * dictionnaire -> dictionnaire
suppression : clé * dictionnaire -> dictionnaire

On représente S par un arbre dont chaque noeud possède au maximum k fils. Chaque chemin entre la racine et un noeud de l'arbre est un préfixe d'une clé de S. Cette structure particulière d'arbre est appelée k-arbre. La figure 1 donne un exemple de cette représentation.

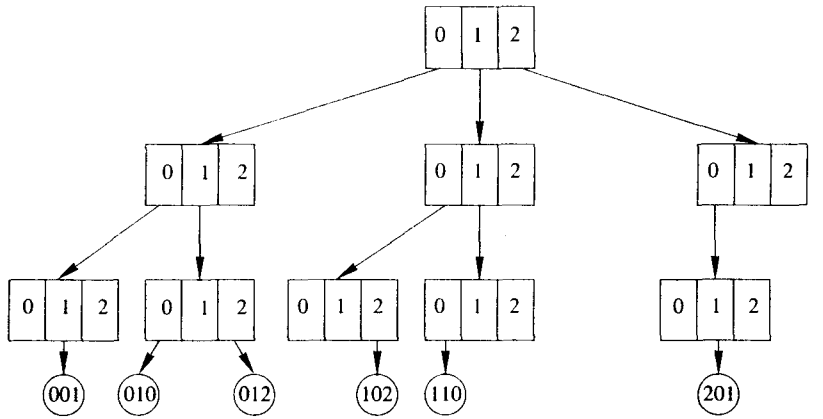

Figure 1 : $S = \{001, 010, 012, 102, 110, 201\}$

La structure de données utilisée pour représenter un k-arbre est la suivante :

En PASCAL :

```
Const
   LONGCLE = <longueur d'une clé> ;
   K = <nombre de lettres de l'alphabet> ;
Type
   TypeNoeud = ( T_FEUILLE, T_INTERNE ) ;
   Feuille = RECORD
      cle : ARRAY [0..LONGCLE-1] OF INTEGER ;
      valeur : INTEGER ;
      END ;
   pNoeud = ^Noeud ;
   NoeudInterne = ARRAY [0..K-1] OF pNoeud ;
   Noeud = RECORD
      CASE t : TypeNoeud OF
      T_FEUILLE : ( f : Feuille ) ;
      T_INTERNE : ( i : NoeudInterne ) ;
      END;
   KArbre = NoeudInterne ;
```

En CAML :

```
type info = {cle:int vect; valeur:int}
and noeudinterne = Nil | Feuille of info |
      Noeud of (noeudinterne array)
and karbre = noeudinterne
```

II.A -

II.A.1) Déterminer la complexité en temps (nombre de noeuds parcourus) pour l'opération recherche.

II.A.2) Évaluer l'espace mémoire maximal utilisé pour représenter un dictionnaire S, de taille n, par un k-arbre. Commenter.

II.B - Une méthode simple pour réduire l'espace occupé par l'arbre consiste à supprimer les noeuds internes qui n'ont qu'un seul fils. Le passage d'un noeud père à un noeud fils nécessitera d'indiquer le nombre de lettres de la clé omises entre le père et le fils. Cette nouvelle structure s'appelle un k-arbre comprimé. La figure 2 présente le k-arbre de la figure 1 une fois comprimé.

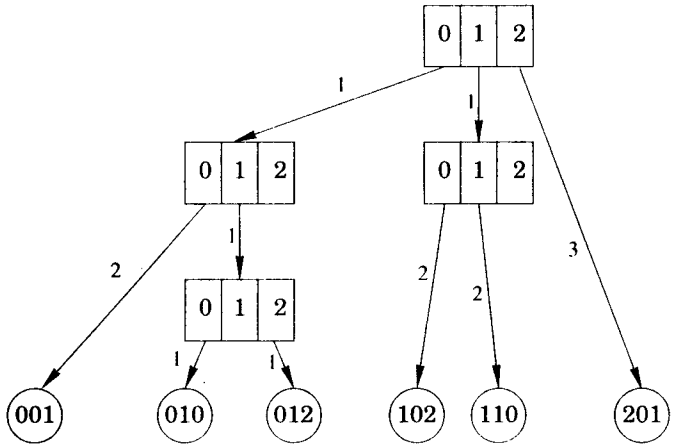

Figure 2 : $S = \{001, 010, 012, 102, 110, 201\}$

II.B.1) Modifier la structure de données précédente pour représenter ce nouvel arbre. Donner un algorithme pour la fonction recherche dans un k-arbre comprimé. La démonstration de l'algorithme n'est pas demandée.

II.B.2) Construire à la main le k-arbre et le k-arbre comprimé associés au dictionnaire $S = \{001, 013, 310, 023, 230, 333, 121, 232, 321, 010\}$ sur l'alphabet $v = \{0, 1, 2, 3\}$.

II.B.3) Que devient l'espace mémoire requis pour représenter un dictionnaire S de taille n ? Que devient la complexité en temps de la fonction recherche dans le pire cas ?

II.C - Soit à déterminer le temps moyen de recherche d'un élément d'un dictionnaire S quelconque, de taille n. On suppose que tous les dictionnaires de taille n sont équiprobables.

II.C.1) Calculer le nombre de k-arbres qui représentent des dictionnaires S de taille n. Montrer que parmi ces k-arbres, le nombre N_d de ceux dont le k-arbre comprimé a une profondeur strictement supérieure à d satisfait à l'inégalité :

$$N_d \leq \binom{k^l}{n}\left[1 - \prod_{i=0}^{n-1}\left(1 - \frac{i}{k^{d-1}}\right)\right].$$

Soit $f_{d-1} : v^l \to v^{d-1}$ la fonction qui à une clé associe son préfixe de longueur $d-1$. On pourra raisonner sur le nombre de sous-ensembles S de v^l de taille n pour lesquels la restriction de la fonction f_{d-1} à S est injective.

II.C.2) En notant q_d le rapport

$$N_d / \binom{k^l}{n},$$

montrer que la quantité $\bar{d}_n$, définie par

$$\bar{d}_n = \sum_{d \geq 1} d(q_{d-1} - q_d)$$

et correspondant à la profondeur moyenne du k-arbre comprimé pour un ensemble de taille n, satisfait à l'inégalité

$$\bar{d}_n \leq 2\lceil \log_k n \rceil + O(1)$$

La notation $\lceil x \rceil$ désigne le plus petit entier plus grand que x. L'inégalité suivante sera admise :

$$\prod_{i=0}^{n-1}\left(1 - \frac{i}{k^{d-1}}\right) \geq e^{-\frac{n^2}{k^{d-1}}},$$

et on pourra montrer que

$$q_d \leq \min\left(1, \frac{n^2}{k^{d-1}}\right).$$

••• FIN •••