Université de Mons–Hainaut
FS/1/5684 – Algorithmique
Examen de première session – Partie pratique

Le 12 février 2008

Consignes

- Pour cette partie, vous avez le droit de consulter vos notes et tout ouvrage qui vous semble utile.
- Cette partie de l'examen dure 1 heure 45 minutes.
- Veillez à bien justifier vos réponses. Une réponse mal justifiée, même correcte, ne permet pas d'obtenir le maximum des points.
- Quand vous indiquez une complexité, veillez à bien expliquer ce que sont les paramètres qui apparaissent dans le $\mathcal{O}$. Par exemple, $\mathcal{O}(n^2)$ n'a aucun sens si n n'apparaît pas dans l'algorithme ou dans la définition de la structure qui est traitée…

Question 1 – 4 points

Considérons un tableau M de nombres entiers à ℓ lignes et c colonnes. On appelle $M[i][j]$ l'élément à la ligne i et à la colonne j. Ce tableau possède la propriété suivante : si on parcourt le tableau M ligne par ligne, la séquence des éléments rencontrés est strictement croissante. Par exemple, le tableau M pourrait être ($\ell = 3$ et $c = 4$) :

1	3	5	7
10	15	27	28
30	39	52	100

On propose l'Algorithme 1, qui reçoit un tableau M tel que décrit ci-dessus ainsi qu'une valeur k. Il renvoie vrai si et seulement si k apparaît dans une des cases de M.

Expliquez comment fonctionne cet algorithme, et donnez-en la complexité en fonction des dimensions du tableau (veillez à bien détailler tout le raisonnement qui amène à la réponse finale).

```
début
    Entier i := 1 ;
    tant que M[i][1] ≤ k et i ≤ ℓ faire
        i := i + 1 ;
    si i = 1 alors
        retourner faux ;
    sinon
        i := i − 1 ;
        Entier bi := 1, bs := c, m := bi + ⌊(bs − bi)/2⌋ ;
        tant que M[i][m] ≠ k et bi ≤ bs faire
            si M[i][m] < k alors
                bi := m + 1 ;
            sinon
                bs := m − 1 ;
            m := bi + ⌊(bs − bi)/2⌋ ;
        si bs < bi alors retourner faux ;
        sinon retourner vrai ;
fin
```

Algorithme 1 : Un algorithme pour recherche une valeur k dans une matrice.

Correction

L'algorithme consiste à recherche une ligne i de la matrice qui est susceptible de contenir la valeur k, puis à effectuer une recherche dichotomique dans cette ligne.

La ligne i est repérée en utilisant le fait que la matrice est triée ligne par ligne. Si k est présente dans la matrice, elle doit l'être dans la dernière ligne dont le premier élément est $\leq k$. Une première boucle commence donc par faire évoluer i de manière à passer toutes les lignes qui commencent par une valeur $> k$.

À la fin de la boucle, il y a trois possibilités :

1. Soit $i = 1$. Dans ce cas, on n'est pas entré dans la première boucle, donc $M[1][1] > k$, et on est donc sûr que k n'est pas dans la matrice.
2. Soit $2 \leq i \leq \ell$. Dans ce cas, i est le numéro de la première ligne qui commence par une valeur $> k$. La valeur k ne peut donc se trouver que dans la ligne $i-1$.
3. Soit $i = \ell + 1$. Dans ce cas, toutes les lignes commencent par une valeur plus grande que k. Néanmoins, k peut tout de même être présente dans la dernière ligne de la matrice. Ici aussi, la valeur k ne peut donc se trouver que dans la ligne $i-1$.

La recherche dichotomique s'effectue de façon classique (voir Syllabus, section 7.2.3).

Les deux boucles s'exécutent l'une après l'autre. Au pire cas, la première boucle parcourt toutes les lignes. Elle est donc en $\mathcal{O}(\ell)$. La complexité de la seconde boucle est celle de la recherche dichotomique, soit $\mathcal{O}(\log(\text{nombre d'éléments}))$. Dans ce cas, il s'agit de $\mathcal{O}(\log(c))$. La complexité totale est donc en $\mathcal{O}(\ell + \log(c))$. On en peut pas simplifier cette expression, car on ne connaît pas la relation entre ℓ et c, et l'on ne sait donc pas lequel de ces deux termes est le plus significatif.

Question 2 – 3 points

Donnez un algorithme *itératif* qui fusionne deux listes triées. Cet algorithme doit recevoir deux listes L_1 et L_2, qui contiennent des entiers et sont triées par ordre croissant. L'algorithme doit renvoyer une seule liste L qui contient tous les éléments de L_1 et de L_2, et qui est triée elle aussi par ordre croissant.

Remarque : il ne faut pas *créer* de nouveaux éléments dans L, mais bien *déplacer* les éléments de L_1 et L_2 dans L. Cet algorithme détruira donc L_1 et L_2 (ce n'est pas un problème).

Correction

Initialement, on possède un pointeur L_1 vers la tête de la première liste et un pointeur L_2 vers la tête de la seconde liste. Comme les listes sont triées, le premier élément à placer dans la liste résultat est le premier élément de L_1 si celui-ci contient une information plus petite que le premier élément de L_2. Si le premier élément de L_2 contient, au contraire, une information plus petite que le premier élément de L_1, c'est lui qu'il faut placer au début de L.
Une fois le bon élément placé, on peut le retirer de la liste qui le contenait originellement. On se retrouve alors dans la même situation qu'au début : on ajoutera *en fin* de L le plus petit des premiers éléments de L_1 et L_2, qu'on retirera ensuite de la liste d'origine, *etc.*
Cette opération suppose que les deux listes ne sont pas vides. On arrêtera donc la boucle de traitement si L_1 est vide ou si L_2 est vide. La condition de la boucle sera donc $L_1 \neq$ **NIL** et $L_2 \neq$ **NIL**.
À la fin de cette boucle, il se pourrait qu'une des deux listes contienne encore des éléments. Il convient donc de placer ces éléments à la fin de L.
Remarquons que l'insertion en fin, telle que définie dans le cours, est une opération coûteuse, puisqu'il faut parcourir toute la liste pour accéder au dernier élément. Dans ce cas, la procédure décrite ci-dessus est en $\mathcal{O}(n^2)$, où n est le nombre total d'éléments dans les deux listes L_1 et L_2. En effet, la première insertion demande de parcourir une liste de longueur 1, la seconde, une liste de longueur 2, etc. On doit donc effectuer de l'ordre de $1+2+\cdots+n-1 = \frac{(n-1)\times n}{2} = \mathcal{O}(n^2)$ opérations. Par contre, en maintenant à tout moment un pointeur p vers le dernier élément de L, on évite de parcourir cette liste, et l'insertion en fin peut se faire en temps constant. On effectue dès lors n insertions en temps constant, soit $\mathcal{O}(n)$. L'Algorithme 2 présente la solution.

```
Elem * FusionListes(Elem * L1, Elem * L2)
début
    Elem * p ;
    si L1 = NIL alors retourner L2 ;
    sinon si L2 = NIL alors retourner L1 ;
    sinon
        si L1.info < L2.info alors
            L := L1 ;
            L1 := L1.next ;
        sinon
            L := L2 ;
            L2 := L2.next ;
        p := L ;
        tant que L1 ≠ NIL et L2 ≠ NIL faire
            si L1.info < L2.info alors
                p.next := L1 ;
                L1 := L1.next ;
            sinon
                p.next := L2 ;
                L2 := L2.next ;
            p := p.next ;
        si L1 ≠ NIL alors p.next := L1 ;
        sinon p.next := L2 ;
        retourner L ;
fin
```

Algorithme 2 : La fusion de deux listes triées.

Question 3 – 3 points

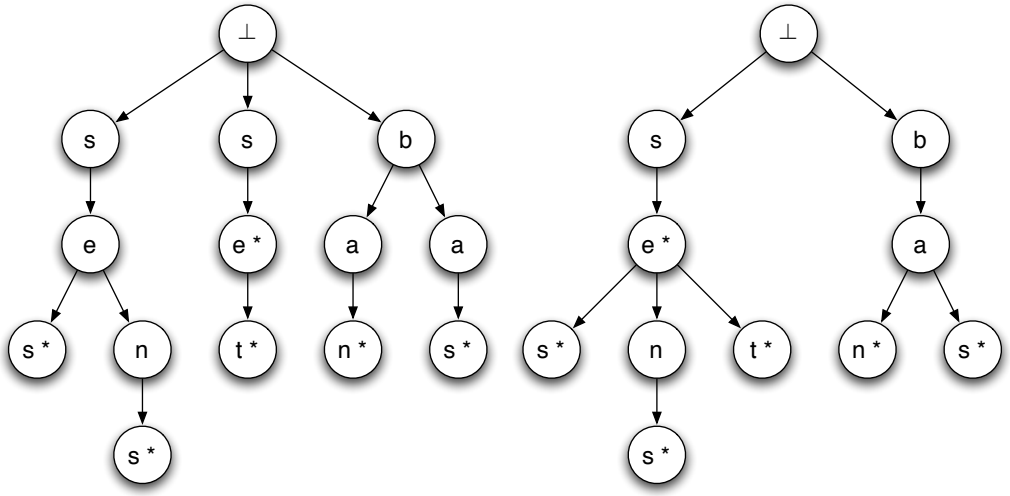
Dans cette question, nous allons considérer des arbres n-aires (où n est plus grand que 2), dont chaque nœud contient un tableau `T` de n pointeurs vers ses n fils ; une étiquette `etiq`, qui est un caractère ; et un booléen `marque`.

Ces arbres sont utilisés pour représenter des ensembles de mots de la façon suivante :

1. La racine est étiquetée par un caractère spécial $\perp$. Son champ `marque` est à faux.
2. Les champs `marque` des autres nœuds peuvent être à vrai ou à faux.
3. Un mot w appartient à l'ensemble représenté par l'arbre si et seulement s'il existe un chemin dans l'arbre tel que :
   (a) la suite des lettres qui étiquettent les nœuds du chemin (sauf la racine) forme le mot w
   (b) le dernier nœud du chemin a son champ `marque` à vrai.

Un tel arbre est donc *presque* un *Trie*. En effet, comme les *Tries*, ces arbres représentent des mots par des chemins dans l'arbre. Mais contrairement aux *Tries*, il est possible qu'un nœud n possède plusieurs fils différents étiquetés par la même lettre. On vous demande d'écrire un algorithme *récursif* qui reçoit un tel arbre et le transforme en *Trie* qui représente le même ensemble de mots : pour chaque nœud n de l'arbre retourné, et chaque lettre ℓ, il y a au plus un fils n' de n dont l'étiquette `etiq` vaut ℓ.

Par exemple, dans l'illustration ci-dessous, nous avons marqué d'une étoile (`*`) les nœuds dont le champ booléen est vrai. L'arbre de gauche représente donc les mots `ses`, `sens`, `se`, `set`, `ban` et `bas`. Si l'algorithme demandé reçoit l'arbre de gauche, il doit le transformer en le *Trie* de droite :



Correction

Comme on demandait un algorithme récursif, il y avait lieu d'analyser le problème « de façon récursive », c'est-à-dire en considérant d'abord le cas de base, puis des cas plus complexes :

- Le cas de base est l'arbre vide. Si la fonction reçoit un arbre vide, elle n'a pas de traitement à effectuer.
- Le cas inductif est le cas où l'arbre possède une racine et plusieurs sous-arbres (potentiellement vides). Dans ce cas, l'algorithme doit s'arranger pour que les fils de la racine (s'ils existent) respectent la condition donnée dans l'énoncé. L'algorithme doit donc considérer chaque fils l'un après l'autre, et le comparer à tous les autres. Si l'algorithme détecte deux fils (non-vides) f et f' qui ont la même étiquette, il doit les « fusionner », c'est-à-dire qu'il doit attribuer à f tous les fils de f', et marquer f si f' est marqué. Ensuite, le nœud f' peut être supprimé. Une fois que toutes les opérations de fusion ont été réalisées, il reste à effectuer un appel récursif sur chacun des fils restants.
  Cette opération est illustrée à la Fig. 1.

L'Algorithme 3 donne la solution. Il suppose l'existence d'une fonction **FusionNoeuds**, qui reçoit deux pointeurs f et f' vers des nœuds et attribue à f tous les fils de f', comme décrit ci-dessus.

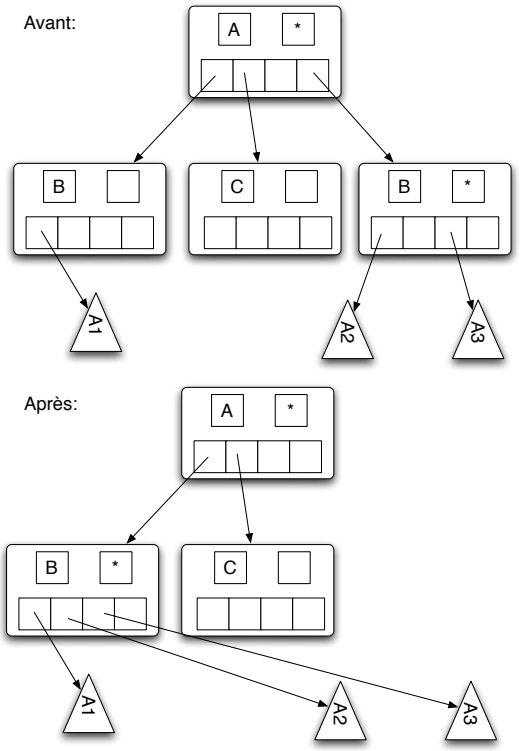


FIG. 1 – L'illustration de la fusion de deux nœuds.

Trie(**Nœud *** A)
début
  Entier i, j ;
  Nœud * f, f' ;
  si $A \neq$ **NIL alors**
    $i := 1$;
    tant que $i \leq n$ **faire**
      si $A.T[i] \neq$ **NIL alors**
        $f := A$.T[i] ;
        $j := i+1$;
        tant que $j \leq n$ **faire**
          si $A.T[j] \neq$ **NIL alors**
            $f' := A$.T[j] ;
            si $f.etiq = f'.etiq$ **alors**
              FusionNoeuds(f, f') ;
              si $f'.marque = vrai$ **alors** f.marque := $vrai$;
              delete A.T[j] ;
              A.T[j] := **NIL** ;
          $j := j+1$;
      $i := i+1$;
    $i := 1$;
    tant que $i \leq n$ **faire**
      Trie(A.T[i]) ;
      $i := i+1$;
fin

Algorithme 3 : L'algorithme pour transformer un arbre en *Trie*.