Université de Mons
FS/1PREPAA.P.MAS.IN/6584 – Algorithmique
Examen de seconde session – Partie pratique

Le 23 août 2010

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

Donnez les complexités (simplifiées, et en terme de $\mathcal{O}$) des deux fonctions données à l'Algorithme 1. Justifiez.

Remarquez la différence dans l'initialisation de la variable k (seule différence entre les deux fonctions).

Correction

La première fonction est en $\mathcal{O}(n \times m)$. En effet, le contenu de la boucle la plus intérieure est en $\mathcal{O}(1)$. Cette boucle s'exécute toujours m fois. Elle est contenue dans la boucle qui utilise l'indice j, qui s'exécute toujours 7 fois. La boucle médiane a donc une complexité de $\mathcal{O}(7 \times m)$, soit $\mathcal{O}(m)$. Finalement, la boucle extérieure s'exécute toujours n fois, et on obtient donc $\mathcal{O}(n \times m)$.

La seconde fonction est en $\mathcal{O}(n+m)$. Dans ce cas, il importe d'observer que la boucle la plus intérieure va effectuer m itérations la première fois qu'on l'atteint (c'est-à-dire pour $i = 1$ et $j = 1$), mais ne s'exécutera plus après, étant donné que la variable k n'est plus jamais réinitialisée à 1. On obtient la complexité en tenant le raisonnement suivant. On atteint la boucle la plus intérieure $7 \times n$ fois (en raison des deux boucles les plus extérieures qui la contiennent), mais la boucle la plus intérieure n'effectue m opérations qu'une seule fois, et 0 itérations les autres fois. La première des $7 \times n$ « exécutions » de la boucle la plus intérieure « coûte » donc $\mathcal{O}(m)$, alors que les $7 \times n - 1$ autres « coûtent » $\mathcal{O}(1)$. On a donc : $\mathcal{O}(m) + (7 \times n - 1) \times \mathcal{O}(1) = \mathcal{O}(m + (7 \times n - 1)) = \mathcal{O}(m + n)$.

Question 2 – 6 points

Considérons la structure dont deux exemples sont donnés à la Fig. 1, où le vecteur V est de type **Nœud** *$V[n]$, et contient donc, dans chacune de ses cases, la tête d'une liste d'entiers. Les éléments de ces listes sont triés de la manière suivante :

1. Chaque liste est triée de manière *croissante* ;

```
F1(Entier n, Entier m)  début
    Entier i, j, k ;
    i := 1 ;
    tant que i ≤ n faire
        j := 1 ;
        tant que j ≤ 7 faire
            k := 1 ;
            tant que k ≤ m faire
                Afficher i, j et k ;
                k := k + 1 ;
            j := j + 1 ;
        i := i + 1 ;
fin
```

```
F2(Entier n, Entier m)  début
    Entier i, j, k ;
    i := 1 ;
    k := 1 ;
    tant que i ≤ n faire
        j := 1 ;
        tant que j ≤ 7 faire
            tant que k ≤ m faire
                Afficher i, j et k ;
                k := k + 1 ;
            j := j + 1 ;
        i := i + 1 ;
fin
```

Algorithme 1 : Les deux fonctions dont il faut calculer la complexité

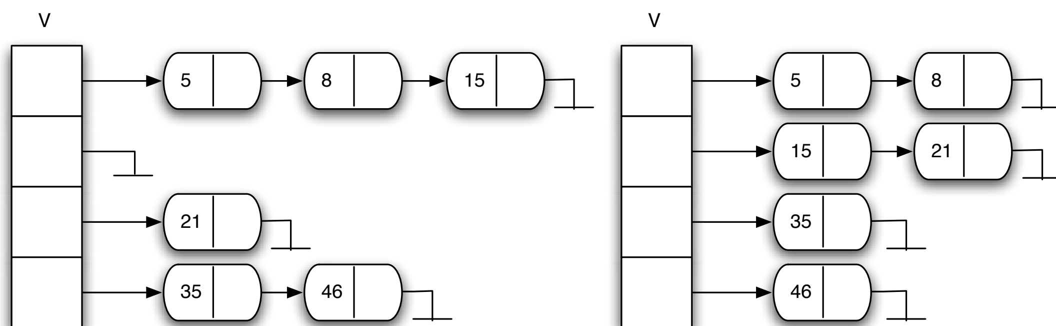

FIG. 1 – Exemples de listes dont les têtes sont dans des vecteurs : avant et après l'appel à l'algorithme demandé.

2. le dernier élément de la liste dont la tête est en $V[i]$ (pour tout $1 \leq i \leq n-1$) est inférieur au égal au premier élément de la première liste non-vide qui suit (dont la tête est en $V[j]$, avec $j > i$), si elle existe.

Autrement dit, les valeurs qu'on rencontre en parcourant les listes successivement, et dans l'ordre imposé par V, est une suite croissante de valeurs.

A priori, les longueurs (nombre d'éléments) des listes dont les têtes sont stockées dans V sont *arbitraires*. On vous demande d'écrire une fonction qui *répartit* ces éléments dans les différentes listes, de telle manière que *la différence de longueur (nombre d'éléments) entre deux listes quelconques soit de* 1 *au plus*, et ce, tout en respectant l'ordre donné. Cette répartition est illustrée à la Fig. 1, à droite (remarquez que les listes les plus longues ne doivent pas forcément être les premières dans le tableau V, il existe donc d'autres *outputs* corrects pour votre algorithme).

Pour ce faire, commencez par déterminer, étant donné la taille n du tableau V et le nombre total E d'éléments dans les listes de V :

1. Dans quel(s) cas toutes les listes auront la même longueur après la répartition ?
2. Si toutes les listes n'ont pas la même longueur, quelles seront les deux longueurs possibles des listes après répartition ?
3. Si toutes les listes n'ont pas la même longueur, combien y aura-t-il de listes de chaque longueur ?

Écrivez ensuite l'algorithme demandé, sur base de cette analyse. Pour ce faire, vous pouvez supposer que les valeurs n (la taille de V) et E (le nombre d'éléments dans les listes) vous sont données. Vous pouvez également supposer que vous avez accès à une fonction **Entier Longueur**(**Nœud** * L) qui renvoie la longueur de la liste dont la tête est L.

Conseil : vous pouvez vous aider des fonctions :

1. *modulo*, notée '%', qui renvoie le reste d'une division entière : par exemple, $6\%4 = 2$.
2. *division entière*, notée 'div', qui renvoie la partie entière du résultat de la division : par exemple $6 \operatorname{div} 4 = 1$.

Correction

Il y aura exactement le même nombre d'éléments dans chaque liste, après équilibrage, si et seulement si la taille n de V divise le nombre total d'éléments E. On peut aussi dire, ce qui est équivalent : ssi $E\%n = 0$ ou encore, ssi E est un multiple de n. Si toutes les listes n'ont pas la même longueur, comme la différence peut être de 1 au plus, les longueurs seront $E \operatorname{div} n$ et $E \operatorname{div} n + 1$. Dans notre exemple, nous avons $E = 6$ et $n = 4$ et donc $E \operatorname{div} n = 1$. Le nombre de listes « plus longues » sera exactement égal à $E\%n$ (2 dans ce cas), puis que $E\%n$ représente le reste de la division, et donc le nombre d'éléments qui sont « en trop ». Comme il faut insérer exactement 1 de ces éléments excédentaires dans chaque liste, $E\%n$ est aussi le nombre de listes « plus longues ».

Partant de ce constat, il est facile d'écrire l'algorithme demandé. On décide arbitrairement que les $E\%n$ premières listes contiendront $E \operatorname{div} n + 1$ éléments (ceci reste correct, même si n divise E, car dans ce cas $E\%n = 0$), et que les $n - E\%n$ dernières listes contiendront $E \operatorname{div} n$ éléments. Ensuite, on parcourt chaque liste l'une après l'autre, on calcule sa taille, et on l'ajuste :

1. Si la liste contient des éléments *en trop*, on la parcourt jusqu'au dernier élément qui doit rester dans liste, et on prélève les éléments excédentaires pour les insérer *en tête* de la liste suivante. Ceci peut donc alonger la liste suivantes, mais ce n'est pas un problème car on la traitera après, quitte à la racourcir si nécessaire. Une fois une liste traitée, on sait qu'elle contient les bons éléments, et on ne la modifie plus.
2. Si la liste contient *trop peu* d'éléments, on parcourt les listes suivantes dans lesquels on prélève des éléments *en tête* pour les insérer *à la fin* de la liste « trop courte ».

Correction

```
begin
    /* Le numéro de la liste en cours de traitement                        */
    Entier i := 1 ;
    /* La taille désirée pour la liste                                     */
    Entier t := (E div n) + 1 ;
    /* La taille de la liste avant traitement                              */
    Entier taille ;
    Nœud * p,q,tmp ;
    Entier cpt, j ;
    tant que i ≤ n faire
        taille := Longueur(V[i]) ;
        si taille > t alors
            /* On passe tous les éléments qui doivent rester dans V[i]     */
            p := V[i] ;
            cpt := 1 ;
            tant que cpt < t faire p := p.next; cpt := cpt + 1 ;
            /* Tous les éléments après p doivent maintenant être déplacés  */
            tmp := V[i+1] ;
            V[i+1] := p.next ;
            p.next := NIL ;
            p := V[i+1] ;
            tant que p.next ≠ NIL faire p := p.next;
            p.next := tmp ;
        sinon si taille < t alors
            /* On met dans cpt le nombre d'éléments manquants              */
            cpt := t − taille ;
            /* On place un pointeur à la fin de la liste trop courte       */
            p := V[i] ;
            tant que p.next ≠ NIL faire p := p.next ;
            /* On parcourt les listes suivantes à la recherche des éléments
               manquants                                                   */
            j := i + 1 ;
            tant que cpt ≠ 0 faire
                tant que V[j] = NIL faire j := j + 1 ;
                /* On a trouvé un élément en tête de V[j], on l'insère après p */
                p.next := V[j] ;
                V[j] := V[j].next ;
                p := p.next ;
                cpt := cpt − 1 ;
            p.next := NIL ;
        /* On passe maintenant à la liste suivantes, en adaptant t         */
        i := i + 1 ;
        si i > E%n alors t := E div n ;
end
```