EXAMEN (Corrigé)

Module: Algorithmique avancée	**Niveau:** 3 Info
Documents: Non autorisés	**Durée:** 1h30 min
Professeur : Hamrouni K.	**Date :** 15 Juin 2010

Exercice-1 : *(5 points)*
Considérer la fonction TRUC placée dans l'encadré-1.

- *a- Quelle est la tâche réalisée par la fonction TRUC ? Expliquer votre réponse.*
- *b- Estimer la complexité de la fonction TRUC en nombre de comparaisons « X[j+1] < X[j] »*
- *c- Donner une version récursive de la fonction TRUC.*
- *d- Calculer la complexité de la version récursive*

Encadré-1

```
void TRUC (int *X, int N)
{   int k, j ,T;
    for (k =N-1 ; k >= 0 ; k = k -1)
    {   for (j= 0 ; j < k ; j++)
            if (X[j+1] < X[j] )
            {   T=X[j] ;
                X[j] = X[j+1];
                X[j+1] = T;
            }
    }
}
```

SOLUTION :

a- La fonction TRUC effectue un tri par ordre croissant du tableau X composé de N éléments par la méthode de « tri à bulles ». En effet, la for (j= effectue un parcours du tableau compare deux éléments adjacents et les permute si le 2ème est inférieur au précédent.

b- $T(N) = \sum_{k=N-1}^{1} k = \frac{N(N-1)}{2}$

En effet : la boucle « for (j= » effectue k comparaisons et la boucle « for (k » répète k de (N-1) à 1

c- Version récursive :

```
void TRUC (int *X, int N)
{   int k, j ,T;
    If(N<2) return   // condition d'arrêt
    for (j= 0 ; j < N-1 ; j++)
    {   if (X[j+1] < X[j] )
        {   T=X[j] ;
            X[j] = X[j+1];
            X[j+1] = T;
        }
    }
    TRUC(X, N-1);
}
```

d- Complexité:

$$T(N) = \begin{cases} 0 & si \quad N = 1 \\ T(N-1) + N - 1 & \sin on \end{cases}$$

C'est une formule récurrente linéaire d'ordre 1. Pour la résoudre il faut appliquer la formule :

$$u_n = a^n (u_0 + \sum_{i=1}^{n} \frac{f(i)}{a^i})$$

$$T(N) = 1^N (T(1) + \sum_{i=2}^{N} \frac{(i-1)}{1^i}$$

Donc : $$T(N) = \sum_{j=1}^{N-1} j = \frac{N(N-1)}{2}$$

Exercice-2 : *(4 points)*

- *a- Ecrire une fonction permettant de lire des entiers positifs et les afficher dans l'ordre inverse d'entré (ie. Le dernier lu sera le premier affiché). Attention : l'utilisateur n'annonce pas le nombre d'entiers à lire. Un entier nul signifie la fin de la liste des entiers. Il ne faut utiliser ni un tableau ni une pile.*
- *b- Estimer sa complexité en nombre d'instruction d'affichage.*

SOLUTION :

a- La fonction doit être récursive. Pour lire et afficher à l'envers, il faut que la lecture se fasse avant l'appel récursif et l'affichage après l'appel récursif.

```
void Afficher ()
{       int x ;
        cin >> x
        if (x != 0 ) Afficher () ;
        cout << x ;
}
```

a- Complexité :

La complexité doit être estimée en fonction de la taille de la donnée. Ici, cette taille n'est fournie explicitement. Nous allons supposer que le nombre d'entiers est N. L'appel à Afficher à l'intérieur de la fonction permettra de générer (N-1) entiers. Donc : $T(N)=\begin{cases}1 & si \quad N=0\\ T(N-1)+1\end{cases}$

Avec N : le nombre d'éléments tapés

On applique la formule et on trouve :

$$T(N)=1^N(T(0)+\sum_{i=1}^{N}\frac{1}{1^i}=N$$

Exercice-3 : *(6 points)*

*Soient deux entiers A et B strictement positifs. On voudrait écrire une fonction pour multiplier l'entier A par l'entier B en n'utilisant que des opérations d'addition. On remarque que A*B = A+A+…….+A (B fois).*

- *a- On vous demande d'écrire une fonction récursive « int Mul1 (int A, int B) » permettant de calculer et de retourner le résultat de la multiplication de A par B. Calculer sa complexité T(B) en nombre d'opérations d'addition après avoir établi une formule de récurrence.*
- *b- Sachant que B/2 donne le quotient entier de B divisé par 2, trouver une relation entre A*B et A*(B/2)*
- *c- Ecrire une fonction « int Mul2 (int A, int B) » permettant de calculer et de retourner le résultat de la multiplication de A par B selon la démarche « diviser pour régner ». Estimer sa complexité T(B) en nombre d'opérations d'addition après avoir établi une formule de récurrence.*

SOLUTION :

a- Mul1

```
Int Mul1( int A, int B)
{       if (B==1) return A ;
        Return ( Mul1(A, B-1) +A) ;
}
```

Complexité:

$$T(B)=\begin{cases}0 & Si \quad B=1\\ T(B-1)+1\end{cases}$$

C'est une récurrence linéaire d'ordre1. On applique la formule et on trouve:

$$T(B)=1^B(T(1)+\sum_{i=2}^{B}\frac{1}{1^i}=B-1$$

b- La relation demandée est :

$$A*B=\begin{cases}(A*\frac{B}{2})*2 & si \quad B \quad est \quad pair\\ (A*\frac{B}{2})*2+A & si \quad B \quad impair\end{cases}$$

c- Mul2

```
Int Mul2(int A, int B)
{       int x;
        if (B==1 ) return A;
        x= Mul2 (A, B/2);
        if ( B % 2 == 0)
                return (x+x) ;
                else
                return (x+x+A);
}
```

Complexité:

$$T(B)=\begin{cases}0 & si \quad B=1\\ T\left(\frac{B}{2}\right)+2\end{cases}$$

C'est une recurrence de type "diviser régner". On applique le théorème :

a = 1 ; b=2 ; k=0

$a=b^k$ ➔ 2ème cas du théorème ➔

$$T(B)=\Theta(B^0\log(B))=\Theta(\log(B))$$

Encadré-2

```
struct node
{   int X ;
    struct node *G ;
    struct node *D;
};
struct arbre
{   struct node *racine ;} ;
```

Exercice-4 : *(5 points)*

Etant donné un arbre binaire A composé de N nœuds. Chaque nœud contient : X : une valeur entière, G : un pointeur sur le fils gauche et D : un pointeur sur le fils droit. La structure « arbre » contient un seul pointeur « racine » pointant sur le premier nœud de l'arbre (voir encadré-2). On suppose que l'arbre A est équilibrée (toutes les branches ont la même longueur).

*a- On suppose dans cette question que l'arbre binaire A n'est pas un ABR (il ne vérifie pas les conditions d'un arbre binaire de recherche). Ecrire une fonction « int Maximum (arbre *A) » permettant de chercher et de retourner la valeur maximale se trouvant dans l'arbre. Donner une formule récurrente donnant sa complexité T(N) en nombre de comparaisons, puis estimer T(N).*

*b- On suppose dans cette question que A est un arbre binaire de recherche. Ecrire une fonction « int Maximum (arbre *A) » permettant de chercher et de retourner la valeur maximale se trouvant dans l'arbre. Estimer sa complexité T(N) en nombre de comparaisons.*

SOLUTION :

a- Si A n'est pas un ABR, il faudra parcourir tout l'arbre pour trouver le maximum. Au niveau de chaque famille, prendre le plus grand entre la valeur du père, le maximum de la sous-famille gauche et le maximum de la sous-famille droite.

```
Int Maximum (Arbre A)
{       return Max (A.Racine) ; }
Int Max (Node *pere)
{       int Sup,SupGauche,SupDroit ;
        if (pere == NULL) return -9999;
        Sup =pere->X;
        SupGauche = Max(pere->G);
        SupDroit    = Max (pere -> D);
        if (SupGauche> Sup) Sup=SupGauche;
        if (SupDroit   > Sup) Sup=SupDroit;
        return Sup ;
}
```

Complexité :

L'arbre étant supposé équilibré. Donc toutes les branches ont la même longueur. La complexité mesure le nombre de comparaisons en fonction du nombre de nœuds qu'on suppose égal à N.

$$T(N) = \begin{cases} 0 & si \quad N = 1 \\ 2T\left(\frac{N}{2}\right) + 1 \end{cases}$$

a = 2 ; b= 2 ; k =0 ; $a > b^k$ ➔ 1er cas du théorème ➔ $T(N) = \Theta(N^{\log_b a}) = \Theta(N)$

b- Si A est un ABR, la valeur maximale se trouve au bout de la branche droite. Il suffit de descendre dans l'arbre à l'aide d'une boucle while

```
Int Maximum (Arbre A)

{       Node *père ;

        Père=A.Racine ;

        While ( père -> D != NULL)

        {       Père = père -> D ; }

        Return ( père -> X) ;

}
```

Complexité :

Puisque nous avons supposé que l'arbre est équilibré, la complexité de cette fonction est égale à la longueur d'une branche qui est égale à la hauteur de l'arbre qui est logN. N étant le nombre de nœuds de l'arbre

⇨ T(N) = logN