Corrigé de l'Interrogation d'algorithmique de graphes Licence MIAGE 2008-2009

Claire Hanen. Le 15 octobre 2008, durée 1h. Les documents ne sont pas autorisés.

Exercice : tri par insertion

Le tri par insertion d'un tableau de n entiers consiste à insérer un à un les éléments d'un tableau dans le sous- tableau trié des éléments déjà insérés.

Q1 : Rappeler les instructions C permettant d'allouer dynamiquement un tableau d'entiers de Nbmax cases.

```
T=(int *)malloc(Nbmax*sizeof(int)) ;
```

Q2 : Ecrire une fonction C itérative permettant d'insérer un nouvel élément dans un tableau supposé trié en ordre croissant d'entiers. Sont passés en paramètres le tableau, son nombre d'éléments (inférieur à Nbmax), et l'élément à insérer et la fonction retourne le nombre d'éléments.

```
int insere( int *t , int n, int el)
/* on suppose bien entendu que le tableau est suffisamment grand et que l'on peut écraser la
/*valeur de t[n]
{ t[n]=el;
int i=n-1;
while( (i>=0)&& (t[i]>el))
        {t[i+1]=t[i]; t[i]=el; i--;}
return (n+1);
}
```

Q3 : Analyser sa complexité dans le pire des cas, et établir un invariant de la boucle.

a)La boucle while comporte au plus n itérations. A chaque itération, on fait 2 comparaisons et 3 affectations. A cela s'ajoute les deux affectations initiales et le return, d'où au pire 3+5n, donc un ordre de grandeur O(n)

b)

```
int insere( int *t , int n, int el)
/* On note t0 le tableau tel qu'il est initialement. On sait qu'il est trié en ordre croissant pour
/* ses indices de 0 à n-1.
{int i=n-1;
t[n]=el;
while( (i>=0)&& (t[i]>el))
        {t[i+1]=t[i]; t[i]=el; i--;}
/* à la fin de  chaque itération de cette boucle, t[i+1]=el et
/* pour tout i+1< k<=n, t[k]=t0[k-1] et t[k]>el
/* pour tout 0<= k <=i, t[k]=t0[k]

return (n+1);
}
```

On pourrait aussi en donner une version récursive:

```
int insere( int *t , int n, int el)
```

```
int aux ;
{if (n= = 0) t[0]=el;
else {if (el>t[n-1])
        t[n]=el;
        else
        {t[n]=t[n-1];
        aux=insere(t,n-1,el) ;
        }
        }
return (n+1);
}
```

Q4 : Utiliser la fonction de la question 1 pour écrire une fonction récursive qui réalise le tri par insertion d'un tableau.

```
void tri_insert_rec(int *t, int n)
/* n est le nombre d'éléments du tableau
{int x ;
if (n>1)
        { tri_insert_rec(t, n-1) ;
        x=insere(t,n-1,t[n-1]) ;
        }
}
```

Q5: Utiliser la fonction de la question 1 pour écrire une fonction itérative qui réalise le tri par insertion d'un tableau.

```
void tri_insert_ite(int *t, int n)
{int i ;
for(i=1 ;i<n ;i++) x=insere(t,i-1,t[i]) ;
}
```

Q6 : Analyser la complexité dans le pire des cas de l'une ou de l'autre des deux fonctions.

Pour la fonction récursive :

Un appel à tri_insert_rec consiste à faire 1 comparaison, un appel récursif au tri pour un tableau de taille n-1, et un appel à insere sur un tableau de taille n-1, qui a donc une complexité en O(n-1).

On en déduit l'inégalité suivante :

$C_{tri_insert_rec}(n) \leq 1 + C_{tri_insert_rec}(n-1) + a.(n-1)$

Où $a \geq 1$ est une constante indépendante de n.

D'autre part, $C_{tri_insert_rec}(1)=1$ (une comparaison)

Montrons par récurrence que $C_{tri_insert_rec}(n) \leq b.n^2$, où b est une constante indépendante de n.

C'est vrai pour n=1 dès que b>1.

Soit $n \geq 2$.

Supposons (hypothèse de récurrence) que c'est vrai jusqu'à l'entier n-1, c'est-à-dire que $C_{tri_insert_rec}(n-1) \leq b.(n-1)^2$

On sait que $C_{tri_insert_rec}(n) \leq 1 + C_{tri_insert_rec}(n-1) + a.(n-1) \leq 1-a+a.n + b.(n-1)^2$

D'où $C_{tri_insert_rec}(n) \leq 1-a+a.n + -2.n.b + b + bn^2$

Or, pour $n \geq 2$, $1-a+a.n + -2.n.b + b \leq 0$ dès que $b \geq (a(n-1)+1)/(2n-1)$

Or (n-1)/(2n-1) est toujours inférieur à 1, de même que 1/(2n-1).
Par conséquent, pour b=a+1, on a $C_{tri_insert_rec}(n) \leq bn^2$
Pour la fonction itérative :
La boucle for comporte exactement n-1 itérations. Dans chaque itération i, on a une comparaison, deux affectations et un appel à insere pour un tableau de taille i-1.

Donc le temps passé dans les comparaisons affectations globalement (en dehors des appels à insère) est de 3(n-1).
Pour exprimer la complexité dans le pire des cas des appels à insère, on doit donc réaliser une somme, pour i variant de 1 à n-1 de la complexité de insere pour un tableau de taille i-1, soit a.(i-1), où a est une constante.
On a donc une complexité :$a \sum_{k=1}^{n-1} k = a.n(n-1)/2$

D'où une complexité totale 3(n-1)+an(n-1)/2 qui est en $O(n^2)$. En effet, on peut trouver une constante b telle que $3(n-1)+an(n-1)/2 \leq bn^2$
Il suffit de chosir b=3+a/2.

Q7 : En supposant que le tableau est au départ déjà trié par ordre croissant, est-ce que l'algorithme est plus rapide que dans le pire des cas ? (on pourra montrer un exemple)

Avec les algorithmes tels qu'ils sont proposés dans ce corrigé, oui, la complexité est différente lorsque le tableau est déjà trié. En effet, dans l'algorithme insere, si l'élément à insérer est plus grand que tous les autres éléments, il n'y a aucune itération de la boucle while, et donc la complexité du calcul est juste 2 affectations et 2 comparaisons, plus le return. Par conséquent, dans la complexité du tri itératif, comme à chaque itération on va chercher à insérer un élément plus grand que tous les autres, la contribution des appels à insere n'est plus de a.n(n-1)/2 mais de 4(n-1). On en déduit que la complexité globale est alors de 7(n-1) et donc en O(n).

Q8 : La structure de tableau vous semble-t-elle adaptée pour cet algorithme ?
Le fait d'avoir une structure de tableau oblige, lorsque l'on insère un élément, à décaler vers la droite tous ceux qui sont après sa position d'insertion. Cela fait donc au pire n affectations (si l'élément à insérer est plus petit que tous).
Si l'on a une liste chaînée au lieu d'un tableau, cela prend un peu plus de place en mémoire (2 fois en fait) mais par contre, l'insertion d'un maillon ne doit conduire qu'à quelques affectations de pointeurs. Donc, le coût des affectations diminue de manière importante.