Interrogation d'algorithmique du 2 octobre 2006

On s'intéresse à la suite de Fibonacci définie de la manière suivante:

$F_0=0$ $F_1=1$, pour $n>1$, $F_n=F_{n-1}+F_{n-2}$

Voici deux fonctions C permettant de calculer le nième nombre de Fibonacci:

```
int fiborec(int n)
{int s=1;
If (n==0) return(0);
If (n>1) s=fiborec(n-1)+fiborec(n-2);
return(s);
}

int fiboite(int n)
{int s=1,t=0,r, i;
If (n==0) return(0);
for(i=2;i<=n;i++)
{r=s;s=s+t; t=r;}
return(s);
}
```

Question 1:
Dérouler l'exécution d'un appel à fiborec(6), puis d'un appel à fiboite(6). Dans le premier cas, on précisera l'arbre des appels récursifs.

```
Fiborec(6)
        fiborec(5)
                fiborec(4)
                        fiborec(3)
                                fiborec(2)
                                        fiborec(1) =1
                                        fiborec(0)=0
                                =1
                                fiborec(1) =1
                        =2
                        fiborec(2)
                                fiborec(1) =1
                                fiborec(0)=0
                        =1
                =3
                fiborec(3)
                        fiborec(2)
                                fiborec(1) =1
                                fiborec(0)=0
                        =1
                        fiborec(1) =1
                =2
        =5
        fiborec(4)
                fiborec(3)
                        fiborec(2)
                                fiborec(1) =1
                                fiborec(0)=0
                        =1
                        fiborec(1) =1
                =2
                fiborec(2)
                        fiborec(1) =1
                        fiborec(0)=0
                =1
        =3
= 8
```

Pour l'appel à fiboite, on peut résumer l'évolution des variables dans un tableau

i	**Avant for**	**2**	**3**	**4**	**5**	**6**
r		1	1	2	3	5
s	1	1	2	3	5	8
t	0	1	1	2	3	5

Question 2:
Calculer la complexité d'un appel à fiborec(n) en fonction de n.

En lisant le code de fiborec, on peut observer que lorsque n=0 ou 1, il y a un nb constant d'opérations, que nous majorerons par une valeur **a.**
Si n>1, la complexité se compose d'un certain nombre de tests et d'opérations arithmétiques (somme, return, appels) que nous noterons par une constante **b** (indépendante de n et supposée >=a), auquel s'ajoute la complexité des appels récursifs.
Ainsi, la complexité de fiborec répond à l'équation de récurrence suivante:
$C_{fr}(n)=a$ si $n=0,1$, $C_{fr}(n)=b+C_{fr}(n-1)+C_{fr}(n-2)$

Pour évaluer l'ordre de grandeur de cette complexité, on peut déjà remarquer que $C_{fr}(n)>Fn$ (le nombre de Fibonacci) qui répond à la même équation de récurrence avec b=0.

On peut également remarquer qu'il est inférieur à la suite un définie par:
$u_0=1$, $u_n=b+2u_{n-1}$ (puisque $C_{fr}(n)=b+C_{fr}(n-1)+C_{fr}(n-2)<=b+2C_{fr}(n-1)$)
Or, la suite un est majorée par $b2^{n+1}$. D'où l'on peut dire déjà que $C_{fr}(n)$ est en $O(2^n)$
En fait plus précisément, la complexité est de l'ordre du nombre d'or à la puissance n.
Pour être sûr qu'il s'agit bien d'une exponentielle, on peut observer que $C_{fr}(n)$ est supérieur à la suite vn définie par: $v_n=a+2v_{n-2}$. Cette suite est en fait de la forme $a2^{n/2}$. On encadre donc bien $C_{fr}(n)$ par deux fonctions exponentielles.

Calculer celle d'un appel à fiboite(n).
fiboite comporte une boucle de n+1 itérations avec un nb constant d'opérations dans la boucle. On a donc une complexité en O(n).
Préciser également l'invariant de la boucle for.
A chaque fin d'itération de la boucle, on peut dire que $s=F_i$, et $t=r=F_{i-1}$.
Laquelle faut-il préférer et pourquoi?
La comparaison des deux complexités montre qu'il faut de toute évidence préférer fiboite quelles que soient les constantes en jeu. Une fonction linéaire en n est toujours très inférieure à une fonction exponentielle lorsque n devient grand.

Question 3:
On considère la fonction suivante :
*void tabfibo(int n, int *t)*
{for(i=0;i<=n;i++) t[i]=fiboite(i);}
Décrire d'une phrase ce que fait cette fonction et calculer sa complexité.

La fonction tabfibo remplit un tableau passé en paramètre avec les premiers nombres de Fibonacci jusqu'à l'ordre n (son deuxième paramètre).

Elle se compose d'une boucle, qui comporte n+1 itérations. De l'itération 1 à l'itération n, on a une complexité qui est une constante plus celle de l'appel à fiboite(i), qu'on a vu être inférieure à une constante a*i. Par conséquent, la complexité globale est b*(n+1) + somme pour i de 1 à n de a*i,

soit une expression : b*(n+1)+a*n(n+1)/2, qui est majorée par $c*n^2$, en choisissant correctement la constante c en fonction de a et b. D'où une complexité en $O(n^2)$.

Question 4:

Proposer une fonction qui fasse exactement la même chose mais qui soit de complexité O(n).

```
void tabfibo(int n, int *t)
{int  i;
if(n==0) t[0]=0;
else
if (n>0)
{t[0]=0;t[1]=1;
for(i=2;i<=n;i++)
t[i]=t[i-1]+t[i-2];
}
```

Cette fonction reprend la boucle for de fiboite (n-1 itérations). Une itération consiste en un nb d'opérations indépendant de n, d'où la complexité O(n).

Question 5:

Modifier cette fonction de sorte qu'elle n'ait plus qu'un paramètre n, et qu'elle renvoie un tableau de n+1 éléments contenant les nombres de Fibonacci de F0 à Fn.
ui fasse exactement la même chose mais qui soit de complexité O(n).

```
int *tabfibobis(int n)
{int *t ;
If (n>=0)
{t= (int *)malloc((n+1)*sizeof(int));
if(n==0) t[0]=0;
else
{t[0]=0;t[1]=1;
for(i=2;i<=n;i++)
t[i]=t[i-1]+t[i-2];
}
}
return(t)
}
```

Question 6:

Ecrire une fonction de test qui appelle les fonctions de la question 4 puis de la question 5 avec n=6 et affiche pour chacune le tableau résultat..

```
Void test()
{int t[7],*s,i;
tabfibo(6,t);
for(i=0;i<=6;i++) printf("%d \t",t[i]);
printf("\n");
s=tabfibobis(6);
for(i=0;i<=6;i++) printf("%d \t",s[i]);
}
```