TD – algorithmique avancée

Exercice 1 :

Un tableau X est trié par ordre croissant si x(i) <= x(i+1) pour tout i

1. Elaborer un programme récursif permettant de vérifier qu'un tableau X est trié ou non

- *Corrigé*

```
def ordonee(L):
        if len(L)==1:
                return True
        elif L[0]>L[1]:
                return False
        else:
                return ordonee(L[1:])
```

1. Estimer sa complexité

- *Corrigé*

T(n)=O(n)

Exercice 2 :

Rendre récursive la fonction somme suivante :

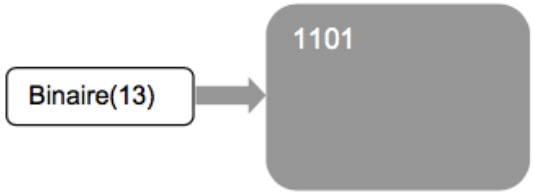
```
def somme(L) :
        s=0 :
        for val in L :
                s+=val
        return s
```

- *Corrigé*

```
def somme(L):
        if len(L)<1:
                return 0
        else:
                return L[0]+somme(L[1:])
```

Exercice 3 :

Pour convertir un nombre entier positif N de la base décimale à la base binaire, il faut opérer par des divisions successives du nombre N par 2. Les restes des divisions constituent la représentation binaire.



1. Ecrire une fonction récursive « Binaire » permettant d'imprimer à l'écran la représentation binaire d'un nombre N (voir exemple en face).
2. Donner la formule récurrente exprimant sa complexité en nombre de divisions. Estimer cette complexité.

- *Corrigé*

```
def binaire(N):
        if N<1:
                return 0
        else:
                rest=N%2
                return str(binaire(N//2))+str(rest)
```

T(n)=T(N/2)+1

T(n) s'écrite sous la forme T(n)=a.T(n/b)+ f(n) avec (a=1,b=2, f(n)=1)

Donc $T(n) = O(n^k \log n) = O(\log n)$

Exercice 4 :

La suite de Fibonacci est définie comme suit :

$$Un\begin{cases} 1 \ si\ n < 2 \\ U_{n-1} + U_{n-2} \ sinon \end{cases}$$

1. Ecrire un programme récursif calculant Fib(n)

- *Corrigé*

```
def Fibonacci(N):
        if n<2:
                return 1
        else:
                return Fibonacci(N-1)+Fibonacci(N-2)
```

2. Déterminer sa complexité

Soit T(n) le nombre d'additions effectuées par cet algorithme. T(n) vérifie les équations suivantes :

$$T(n) = \begin{cases} 0 \ si\ n = 0 \\ 0 \ si\ n = 1 \\ 1 + T(n-1) + T(n-2) \ si\ n > 1 \end{cases}$$

En posant S(n) =T(N)+1 on obtient :

$$S(n) = \begin{cases} 1 \ si\ n = 0 \\ 1 \ si\ n = 1 \\ S(n-1) + T(n-2) \ si\ n > 1 \end{cases}$$

Donc une équation de récurrence sans second membre : $S(n) - S(n-1) - S(n-2) = 0$

Son polynôme caractéristique est : x^2-x-1=0

Il possède deux racines : $\frac{1+\sqrt{5}}{2}$ et $\frac{1-\sqrt{5}}{2}$

La solution de l'équation de récurrence est donc : $a(\frac{1+\sqrt{5}}{2})^n + b(\frac{1-\sqrt{5}}{2})^n$

Les coefficients a et b peuvent être déterminés à l'aide des conditions aux limites :

S(0)=1=a+b

S(1)=1=$\frac{a+b}{2} + \frac{(a-b)\sqrt{5}}{2}$

d'où : a= $\frac{5+\sqrt{5}}{10}$ et b= $\frac{5-\sqrt{5}}{10}$ donc S(n)= $\frac{5+\sqrt{5}}{10}(\frac{1+\sqrt{5}}{2})^n + \frac{5-\sqrt{5}}{10}(\frac{1-\sqrt{5}}{2})^n$

S(n)=T(n)=O$\left((\frac{1+\sqrt{5}}{2})^n\right)$

Exercice 5 :

Soit la suite définie par :

$$Un\begin{cases} 1 \ si\ n<2 \\ 3U_{n-1}+2U_{n-2} \ sinon \end{cases}$$

1. Ecrire un programme récursif permettant de calculer le nième terme de la suite.
2. Estimer sa complexité.

- *Corrigé*

```
def suite(N):
        if n<2:
                return 1
        else:
                return 3*suite(N-1)+2suite(N-2)
```

La complexité de la fonction U en nombre d'opération + est donnée par :

$T(n) = T(n-1) + T(n-2) + 1$

Avant de donner le polynôme caractéristique il faut annuler le second membre par un changement de variable. Si on pose T(n)=S(n)-1 on obtient :

S(n)−1=S(n−1)−1+S(n−2)−1+1 et donc: S(n)−S(n−1)−S(n−2)=0

Le polynôme caractéristique est : P(x)=x2−x−1 Ses racines sont : (1+$\sqrt{5}$)/2 et (1− $\sqrt{5}$)/2

La complexité de la fonction U est: T(n)=T(n−1)+T(n−2)+1

T(n)= $a(\frac{1+\sqrt{5}}{2})^n$ + b$(\frac{1-\sqrt{5}}{2})^n$=O($(\frac{1+\sqrt{5}}{2})^n$)

Exercice 6 :

Un nombre N est pair si (N-1) est impair, et un nombre N est impair si (N-1) est pair.

1. Ecrire deux fonctions récursives mutuelles pair (N) et impair (N) permettant de savoir si un nombre N est pair et si un nombre N est impair.
2. Estimer la complexité de la fonction Pair (N) en nombre d'appels récursifs.

- *Corrigé*

```
def pair(n) :
        if n==0 :
                return True
        else :
                return impair(n-1)
def impair(n) :
        if n==0 :
                return False
        else :
                return pair(n-1)
```

T(n)=1+T(N-1)=O(n)

Exercice 7 :

Etant donné un tableau X composé de N éléments entiers. On voudrait déterminer son maximum par un programme récursif basé sur le paradigme « diviser pour régner » :

1. En considérant que le maximum est le plus grand entre le dernier terme et le maximum des (n-1) premiers termes. Estimer sa complexité.

- *Corrigé*

```
def maximum(L):
        if len(L)==1:
                return L[0]
        nb1=L[-1]
        nb2=maximum(L[:len(L)-1])
        if nb1>nb2:
                return nb1
        else:
                return nb2
```

T(n)=1+1+T(n-1)+1=O(n)

2. En considérant que le maximum est le plus grand entre les maximums des deux moitiés du tableau. Estimer sa complexité.

```
def maximum2(L):
        if len(L)==1:
                return L[0]
        m=len(L)//2
        nb1=maximum2(L[:m])
        nb2=maximum2(L[m:len(L)])
        if nb1>nb2:
                return nb1
        else:
                return nb2
```

T(n)=1+T(n/2)+T(n/2)+1=1+2T(n/2)

T(n) s'écrite sous la forme T(n)=a.T(n/b)+ f(n) avec (a=2,b=2, f(n)=1)

Donc $T(n) = O(n^{\log_b a}) = O(n^{\log_2 2}) =$ **O(n)**

Exercice 8 :

Soit un tableau X composé de N entiers pouvant être 0 ou 1. Une coupe (i,j) de X est le sous tableau commençant à i et finissant à j. on voudrait déterminer la plus longue coupe ne contenant que des 1.

Exemple : dans le tableau suivant, la coupe (7,14) est équilibrée.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	1	1	1	0	1	0	1	1	1	1	1	1	1	1	0	0	0	1	1

1. Ecrire une fonction « **PlusLongueCoupe** » récursive basée sur le paradigme « diviser pour régner » qui détermine la plus longue coupe ne contenant que des 1 d'un tableau X.

```
def PlusLongueCoupe(L):
        if len(L)<2:
                return 0
        m=len(L)//2
        max1=0
        max2=0

        for i in range(m,-1,-1):
                if L[i]==0:
                        break
                max1+=1
        for i in range(m+1,len(L)):
                if L[i]==0:
                        break
                max2+=1
        maximum=max2+max1
        somme=PlusLongueCoupe(L[:m])
        if somme>maximum:
                maximum=somme
        somme1=PlusLongueCoupe(L[m:])
        if somme1>maximum:
                maximum=somme1
        return maximum
```

2. Estimer sa **complexité moyenne** en nombre de comparaisons des éléments du tableau.

T(n)=2T(n/2)+ O(n)

T(n) s'écrite sous la forme T(n)=a.T(n/b)+ f(n) avec (a=2,b=2, f(n)=n)

$T(n)=O(n^{\log_b a}\log_b n)$ donc ***T(n)=O(n log n)***

Exercice 9 :

On voudrait écrire une fonction « multiplier » permettant de multiplier deux entiers **positifs ou nuls a et b** (ce n'est pas la peine de vérifier la validité des données) en utilisant **uniquement des opérations d'addition**. Le prototype de la fonction doit être « **def multiplier (a, b)** ».

1. Proposer une version récursive de la fonction « multiplier ». Estimer sa complexité en nombre d'additions après avoir déterminé une formule récurrente de la complexité.

```
def multiplier(a,b) :
        if b==0 :
                return 0
        else :
                return multiplier(a,b-1)+a
```

Complexité : T(a,b)=T(a,b-1)+1, T(a,0)=0 Donc T(a,b)=b

2. Proposer une autre version récursive de la fonction « multiplier » basée sur le paradigme « diviser pour régner ». Donner une formule récurrente de sa complexité en nombre d'additions et estimer la.

```
def multiplier(a,b) :
        if b==0 :
                return 0
        c= multiplier(a,b//2)
        if b%2==0 :
                return c+c
        return c+c+a
```

Complexité : T(a,b)=T(a,b/2)+3/2 T(a,b)= O(log(b))