TD : Complexité
- Correction -

Exercice 1

On considère le programme suivant:

```
Def Algo(n):
     for i in range(0,n):
       Algo1(n)

     Algo2(n)
     Algo3(n)
```

- Sachant que **Algo1(n)** s'effectue en temps **O(log(n)),Algo2(n) en temps O(n)**et**Algo3(n) en temps O(n^2),** Quelle est la complexité de **Algo(n)?**

La boucle for exécute **n** fois une procédure qui coute **O(log n),** donc la boucle for coutera **O(n.log n)**.
L'algorithme au total coutera donc : **O(n.log n) + O(n) + O(n^2).**
Donc la complexité de **Algo(n)** est : **O(n^2).**

Exercice 2 :

Quelle est la complexité du programme suivant :

```
def Algo(n):
  res = 1;
  for i in range(0,n):
        res = res + i
  for i in range(0,n):
    for j in range(0,i):
        res = res * (i+j)
   return     res
```

Calculons le nombre d'opérations : T(n).

T(n)= 2 + 2.n + $\sum_{i=0}^{n-1}\sum_{j=0}^{i-1} 3$

T(n)= 2 + 2.n + $\sum_{i=0}^{n-1} 3i$

T(n)= 2 + 2.n + $\sum_{i=0}^{n-1} 3i$

T(n)= $2 + 2.n + 3.\frac{n(n-1)}{2}$

Complexité : *O*(n²)

Exercice 3 :

Quelle est la complexité du programme suivant:

```
def f(x, n):
   if n < 1:                    O(1)
        return n
  S = 0
  for i in range(0,n):          O(n)
```

C(0)= 1
C(n)= 1 + n + C(n/2)

Complexité : *O*(n)

```
    S += x/(i+n)

  return S + f(x,n//2)          O(log2(n))
```

Exercice 4

1- Que vaut la complexité de l'algorithme suivant:

```
i=1
while i <= n :
   i=i*2
```

soit k le nombre d'itérations de la boucle while :

dans la dernière itération on aura :

((1*2)*2)*2)...*2) > n

$2^k <= n$

$k <= \log_2(n)$)

Complexité : $O(\log_2(n))$

2- Le code ci-dessous consiste à programmer la fonction ***remove***. Analysersacomplexité?

```
def SupprimerElement(L,a):
   i=0
   if a in L:                    O(n)
         while L[i]!=a:          O(n)
               i=i+1:
         L[i:i+1]=[]             O(1)
```

Complexité : $O(n)$

Exercice 5

Soient les 3 fonctions suivantes permettant de calculer la valeur d'un polynôme.

$$p(x) = a_0+a_1x+a_2x^2+...+a_nx^n$$

en un point x (c'est-à-dire pour une valeur x donnée).

Les coefficients du polynômes sont stockés dans une liste : **A=[a_0, a_1, a_2..., a_{n-1} , a_n]**

```
def f1(A,x) :#méthode naïve
    n =len(A)
    p=A[0]
    for i in range(1,n) :
          p = p + A[i]* x**i
    return p
```

xi** : n'est pas une opération élémentaire : **son cout pour i allant de 1 à n = n(n+1)/2**

T(n)=4+3n+(1+2+3...+n-1)

T(n)=4+3n+((n-1).n)/2

$T(n)= (n^2+5n)/2 +4$

Complexité : $O(n^2)$

```
def f2(A,x) :# elle utilise le fait que x^i = x^(i-1) * x
    n =len(A)
    p=A[0]
    q = 1
    for i in range(1,n) :
          q = q * x
          p = p  +  A[i] * q
    return p
```

T(n)= 4 +5(n-1)
T(n)= 5n-1

Complexité : O(n)

#algorithme de Horner ;

il calcule, DANS CET ORDRE, les valeurs de :a_n, a_nx+_{an-1}, puis (a_nx+a_{n-1}) x + a_{n-2}, etc.

```
def f3(A,x) :# utilisant la méthode de Horner
    n =len(A)
    p = A[n-1] ;
    for i in range(n-1,0,-1) :
          p = p*x + A[i-1]
    return p
```

T(n)= 4 +4(n-1)
T(n)= 4n+1

Complexité : O(n)

Exercice 6

1- Écrire une fonction qui prend un tableau d'entiers ***T*** en argument et renvoie le tableau des sommes cumulées croissantes correspondantes, autrement dit un tableau **S** de même taille dont la ***k-ième*** composante vaut :

$$S[k] = \sum_{i=0}^{k} t[i]$$

2- Evaluer la complexité de cette fonction.
3- Est-il possible d'en écrire une version plus efficace ? Donner alors sa complexité

Correction :

1.

```
def somme(L):
    n= len(L)
    s=[0]*n
    for i in range(n):
        for j in range(i+1):
            s[i]=s[i]+L[j]

    return s
```

2.

La complexité :

On compte le nombre d'opérations :

Complexité de cette fonction est en $\boldsymbol{O(n^2)}$

3.*Une version plus efficace :*

```
def  somme2(L):
    n= len(L)
    s=[0] * n
    s[0]=L[0]
    for k in range(1, n):
        s[k]= s[k-1] + L[k]
    return s
```

On compte le nombre d'opérations :

La complexité de cette fonction est en $\boldsymbol{O(n)}$

Exercice 7

Soit u la suite définie par $\mathbf{u_0 = 1}$ et pour $\mathbf{n \in N}$, $\mathbf{u_{n+1} =}$ ***sin*****$\mathbf{(u_n)}$**.

1. Écrire une fonction nommée suite prenant $\mathbf{n \in N}$ en argument et renvoyant u_n.
2. Que calcule la fonction suivante :

```
def mystere(n):
  s = 0
  for  k  in  range(n+1):
        s = s + suite(k)
  return s
```

3. Déterminer en fonction de **n** le nombre **d'additions** et de calculs de ***sinus*** nécessaires pour calculer ***mystere(n).***
4. Écrire une autre fonction qui renvoie les mêmes valeurs que ***mystere*** mais avec une meilleure complexité.

Correction :

1. *Écrire* une fonction nommée suite prenant $\mathbf{n \in N}$ en argument et renvoyant u_n.

```
from  math import sin
def suite(n):
      u = 1
      for i in range(n):
            u = sin(u)

      return u
```

2. Que *calcule* la fonction suivante :

```
def mystere(n):
    s = 0
    for  k  in  range(n+1):
       s = s + suite(k)

  return s
```

La fonction **`mystere`** calcule la somme des termes de u_0(c-a-d *suite*(0)) à u_n (c-a-d *suite*(n)) : $\sum_{k=0}^{n} u_k$

3. *le nombre d'additions et de calculs de sinus nécessaires pour calculer* ***mystere(n)*** *:*

A chaque passage dans la boucle de ***mystere*** , on a une addition et un appel de la fonction *suite*.

A chaque passage dans la boucle de *suite*, on a un calcul de sinus.

$$\sum_{k=0}^{n}\left(1 + \sum_{i=0}^{k-1} 1\right) = (n + 1) + \sum_{k=0}^{n} k = (n + 1) + \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{3n}{2} + 1$$

Donc la fonction mystère est de complexité = $\boldsymbol{O(n^2)}$

4. *Écrire une autre fonction qui renvoie les mêmes valeurs que mystere mais avec une meilleure complexité.*

```
def mystere(n):
      u = 1
      s= 0
      for k in
range(n):
            u = sin(u)
                  s = s
+ u
```

Cette fonction a une complexité linéaire (c'est à dire en $O(n)$) car elle est composée d'une boucle contenant 4 opérations élémentaires (2 affectations, une addition et le sinus). (on a supposé que le cout de sin(u)=1)

Si on avait appelé la fonction *suite* dans la boucle pour calculer u_k, on aurait eu une complexité quadratique (en $O(n^2)$).

Un appel avec $n = 10^6$ est raisonnable avec la version linéaire, mais pas avec la version quadratique.