Table des matières

I Suites **1**
1 Calcul des termes d'une suite . . . 1
2 Calcul d'une somme de termes . . . 1
3 Syracuse . . . 2
4 Fibonacci . . . 2
5 Factorielle . . . 3

II Arithmétique **3**
1 Division euclidienne . . . 3
2 PGCD . . . 3
3 Écriture en base b . . . 4
4 Exponentiation : x^n . . . 4

III Tris **5**
1 Tri par selection . . . 5
2 Tri par insertion . . . 5
3 Tri rapide . . . 6
4 Tri fusion . . . 6

IV Autres **7**
1 Max . . . 7
2 Euler . . . 7
3 Palindromme . . . 7
4 Dichotomie . . . 8
5 Algorithme de Héron : évaluation de $\sqrt{n}$ . . . 8
6 Lecture dans un fichier . . . 9

I Suites

1 Calcul des termes d'une suite

expl : $\begin{cases} u_{n+1}=2u_n^3+5 \\ u_0 \ =2 \end{cases}$

```python
def suite(n):
    u=2
    for i in range(n):
        print(u)
        u=2*u**3+5
```

```python
def suite_liste(n):
    L=[2]
    for i in range(n):
        L.append(2*L[i]**3+5)
    return L
```

2 Calcul d'une somme de termes

expl : la somme des carrés

```python
def somme(n):
    s=0
    for i in range(1,n+1):
       s=s+i**2
    return s
```

```python
def depasse(M):
    """Renvoie le plus petit entier tel que la somme des n premiers entiers au carré soit supérieure à M.
    """
    assert M>0
    s=0
    n=0
    while s<M:
        n=n+1
        s=s+n**2
    return n
```

3 Syracuse

Algorithmes qui affichent les termes de la suite de Syracuse jusqu'au 1er un.

```python
def syracuse(n):
    print(n)
    while n!=1:
        if n%2==0:
            n=n//2
        else:
            n=3*n+1
        print(n)
```

```python
def syracuse_recursif(n):
   print(n)
   if n==1 :
      return # fin de la recursivite
   if n%2==0 :
      syracuse_recursif(n//2)
   else:
      syracuse_recursif(3*n+1)
```

4 Fibonacci

→ pour ces deux algorithmes, le coût temporel est en $\mathcal{O}(n)$.

```python
def fibonacci(n):
    if n==0: return 0
    a, b = 0, 1
    for i in range(1,n):
        a, b = b, a+b
    return b
```

```python
def fibonacci_recursif(n,a=0,b=1):
    if n==0: return a
    elif n==1: return b
    else :
        return fibonacci_recursif(n-1,b,a+b)
```

5 Factorielle

→ pour ces trois algorithmes, le coût temporel est en $\mathcal{O}(n)$.

```python
def factorielle(n):
    facto, i = 1, 1
    while i<n :
        i = i+1
        facto = facto*i
    return facto
```

```python
def factorielle_recursif_down(n):
    """calcul durant le depilement"""
    if n==0 :
        return 1
    return n*factorielle_recursif_down(n-1)
```

```python
def factorielle_recursif_up(n, m=1, facto=1):
    """calcul durant l'empilement"""
    if n==0:
        return facto
    return factorielle_recursif_up(n-1, m+1, facto*m)
```

II Arithmétique

1 Division euclidienne

```python
def divEucl(a,b):
    """ Ce programme effectue la division euclidienne de a par b (a, b entiers naturels
    ↪ avec b>0).
    e.g DivEucl(19,4) renvoie (4,3)
    """
    r=a
    q=0
    while r>=b:    # Invariant de boucle a=b*q+r
        r=r-b
        q+=1
    return q, r
```

```python
def divEucl_python(a,b):    # avec les fonctions Python
    return a//b, a%b
```

2 PGCD

```python
def pgcd(a,b):
    r0=a
    r1=b
    while b!=0:
        r0, r1 = r1, r0%r1
    return r0
```

```python
def pgcd_bezout(a,b):
    r0, r1 = a, b
    u0, v0 = 1, 0
    u1, v1 = 0, 1
```

```
    while r1!=0:
        q=r0//r1
        r0, r1 = r1, r0-q*r1
        u0, u1 = u1, u0-q*u1
        v0, v1 = v1, v0-q*v1
    return r0, u0, v0
```

```
def pgcd_recursif(p,q):
   if p<q : return pgcd_recursif(q,p)
   if ( p%q==0 ) : return q
   return pgcd_recursif(q,p%q)
```

3 Écriture en base b

→ pour ces deux algorithmes, le coût temporel est en $\mathcal{O}(log_b(n)) = \mathcal{O}(ln(n))$.

```
def baseB(n,b):
    if n==0: return [0]
    q=n
    L=[]
    while q>0:
        x=divEucl(q,b)    # fonction definie dans la sous-section arithmetique
        L.append(x[1])
        q=x[0]
    L.reverse()
    return L
```

```
def baseB_bis(n,b):
    """ avec les fonctions % et // de Python"""
    if n==0: return [0]
    L=[]
    while n>0:
        L.append(n%b)
        n=n//b
    L.reverse()
    return L
```

```
def baseB_recursif(n, b, L=None):
    if L==None :
        if n==0: return [0]
        L=[]
    if n==0:
        L.reverse()
        return L
    L.append(n%b)
    return baseB_recursif(n//b, b, L)
```

4 Exponentiation : x^n

→ pour ces deux algorithmes, le coût temporel est en $\mathcal{O}(n)$.

```
def puissance_naive(x,n):
    res=1
    for i in range(n):
        res=res*x
    return res
```

```python
def puissance_naive_recursif(x,n):
    if n==0: return 1
    return x*puissance_naive_recursif(x,n-1)
```

→ pour ces deux algorithmes, le coût temporel est en $\mathcal{O}(ln(n))$.

```python
def puissance_rapide(x,n):
    q = n
    res = 1
    puiss = x
    while q > 1:      #Invariant de boucle ((puiss)**q)*res
        if q % 2 == 1:
            res = res*puiss
        puiss = puiss*puiss
        q = q//2
    return res*puiss
```

```python
def puissance_rapide_recursif(x,n):
    if n==0: return 1
    result=puissance_rapide_recursif(x,n//2)
    if n%2==0 :
        return result*result
    else :
        return x*result*result
```

III Tris

1 Tri par selection

→ complexité en N^2

```python
def imin(T,a,b):
    """ fonction qui renvoie la position de la valeur minimale de T entre les positions a
    ↪  et b
    """
    imin=a
    N=len(T)
    for i in range(a+1,b):
        if T[i] < T[imin]:
            imin = i
    return imin

def triselection(T):
    N=len(T)
    for i in range(N-1):
        j=imin(T,i,N)
        T[i],T[j] = T[j],T[i]
    return T # optionnel
```

2 Tri par insertion

→ complexité dans le pire cas en N^2, meilleur cas en N si déjà trié, en moyenne $\frac{N^2}{2}$:

```python
def triinsertion(T):
    n=len(T)
```

```
    for i in range (1,n):
        valeur=T[i] # valeur a inserer
        j=i-1
        while (j>=0) and (T[j]>valeur):
            T[j+1]=T[j]
            j=j-1
        T[j+1]=valeur
```

3 Tri rapide

→ complexité dans le pire cas en N^2, meilleur cas $N\log_2(N)$, en moyenne $2N\log_2(N)$:

```
from random import randint

def echange(T,i,j):
    T[i],T[j]=T[j],T[i]

def partition(T,g,d):
    m=randint(g,d-1)        # choix du pivot
    p=T[m]        # on place le pivot a gauche au depart
    echange(T,g,m)
    m=g
        for i in range (g+1,d): # on considere tous les elements
        if T[i]<p :
            echange(T,i,m+1)
            m=m+1
    echange(T,g,m)    # on met le pivot a sa place
    return m

def tri(T,g=0,d=len(T)):    # par defaut tri du tableau entier
    if g<d-1 :    # si le tableau contient au moins 2 elements
        m=partition(T,g,d)
        tri(T,g,m)
        tri(T,m+1,d)
```

4 Tri fusion

→ complexité en $N\log_2(N)$:

```
def fusion(T1,T2):
    n1=len(T1)
    n2=len(T2)
    T=[]
    i=0
    j=0
    while(i<n1 or j<n2):
        if i==n1 :    # on a deja insere tous les elements du tableau 1
            T.append( T2[j] )
            j=j+1
        elif j==n2 :  # on a deja insere tous les elements du tableau 2
            T.append( T1[i] )
            i=i+1
        elif T1[i]<T2[j] :
            T.append( T1[i] )
            i=i+1
        else :
            T.append( T2[j] )
```

```
            j=j+1
    return T

def trirec(T):
    if len(T)<=1 :
        return T
    else :
        m=len(T)//2
        T=fusion( trirec( T[0:m] ), trirec( T[m:len(T)] ) )
    return T
```

IV Autres

1 Max

→ pour ces deux algorithmes, le coût temporel est en $\mathcal{O}(n)$.
Le minimum d'un tableau se calcule de manière similaire (*cf.* changer la ligne `# M n'est pas le Max`).

```
def maxTab(T):    # T un tableau non vide
   M=T[0]
   i=1
   while i<len(T):
      if M<T[i] : # M n'est pas le Max
         M=T[i]
      i=i+1
   return M
```

```
def maxTab_recursif(T, M=None):
   if len(T)==0 :
      return M
   if M==None :
      M=T[0]
   if M<T[0]: # M n'est pas le Max
      return maxTab_recursif( T[1:], T[0] )
   else :
      return maxTab_recursif( T[1:], M )
```

2 Euler

```
def Eulerexplicite(F,a,b,y0,n):
    h=(b-a)/n
    t=a
    y=y0
    T=[t]
    Y=[y]
    for i in range(n):
        y=y+h*F(t,y)
        t=t+h
        Y.append(y)
        T.append(t)
    return T,Y
```

3 Palindromme

→ pour ces deux algorithmes, le coût temporel est en $\mathcal{O}(n)$.

```python
def palindromme(mot):
    n=len(mot)
    i=0
    while i<n//2:
        if mot[i] != mot[n-1-i]:
            return False
        i = i+1
    return True
```

```python
def palindromme_recursif(mot):
    if mot=="":
        return True
    if mot[0]==mot[-1]:
        return palindromme_recursif(mot[1:-1])
    else :
        return False
```

4 Dichotomie

→ pour ces deux algorithmes, le coût temporel est en $\mathcal{O}\left(ln\left(\frac{b-a}{\varepsilon}\right)\right)$.

```python
def dichotomie(f,a,b):
    epsilon = 10**(-6)
    an, bn, cn = a, b, (a+b)/2
    while bn-an>epsilon:
        if f(an)*f(cn)>0:
            an = cn
        else :
            bn = cn
        cn=(an+bn)/2
    return cn   # un antecedent de 0 a epsilon pres
```

```python
def dichotomie_recursive(f,a,b):
    epsilon = 10**(-6)
    if b-a<epsilon:
        return (a+b)/2  # un antecedent de 0 a epsilon pres
    if f(a)*f((a+b)/2)>0:
        return dichotomie_recursive(f,(a+b)/2,b)
    else :
        return dichotomie_recursive(f,a,(a+b)/2)
```

5 Algorithme de Héron : évaluation de $\sqrt{n}$

L'algorithme de Héron est un cas particulier de la méthode de Newton pour résoudre $f(x) = 0$ (pour Héron, on a $f(x) = x^2 - n$).

```python
def heron_recursif(n, a=1):
    r = 1/2*(a+n/a)
    epsilon = 10**(-10)
    if abs(r-a)<epsilon :
        return r
    return heron_recursif(n, r)
```

6 Lecture dans un fichier

On considère un fichier `monFichier.txt` dont la première ligne est composée du mot `DONNEES` puis chaque ligne contient un nombre.
On souhaite faire la moyenne de ces nombres.

DONNEES
1
2
5

```
file = open("monFichier.txt", 'r')
line = file.readline()   # on passe la 1ere ligne avec le mot DONNEES
line=file.readline()
S, nb = 0, 1
while line != "" :   # tant que la ligne n'est pas vide
   S = S+int(line)   # line est une chaine de caracteres qu'il faut convertir
   nb = nb + 1
    line=file.readline()
file.close()
print(S/nb)   # la moyenne est affichee
```