Séance 18 :
Tri rapide

1. Cours

1. ***Ecrire une procédure*** `echange(A, i, j)` ***qui échange les cases*** `A[i]` ***et*** `A[j]` ***d'une liste*** `A`***.***

> main.py
```
def echange(A, i, j):
    A[i], A[j] = A[j], A[i]
```

2. ***Ecrire une fonction*** `partition(A, g, d)` ***qui renvoie la nouvelle place qu'aurait l'élément*** `V = A[g]` ***appelée pivot dans la partie*** `A[g:d]` ***si celle-ci était triée sans tenir compte des autres parties de la liste.***

> main.py
```
def partition(A, g, d):
    V, c = A[g], 0
    for i in range(g+1, d):
        if (A[i] < V):
            c += 1
            echange(A, i, g+c)
    echange(A, g, g+c)
    return c
```

3. ***Modifier la fonction*** `partition(A, g, d)` ***En ajoutant des échanges de cases de manière à ce que tous les éléments inférieurs au pivot de la partie*** `A[g:d]` ***soit en tête de liste et que le pivot lui-même soit à la bonne place.***

> main.py
```
def partition(A, g, d):
    c = 0
    for i in range(g+1, d):
        if (A[i] < A[g]):
            echange(A, i, g+c)
            c += 1
    echange(A, g+c, d+1)
    return c
```

4. ***On donne la procédure suivante de tri rapide récursif : compléter la ligne 9***

> main.py
```
def tri_rec(A, n, p):
    if n < p:
        # le pivot V=A[n] devient V=A[m] a la bonne place m dans A
        m = partition(A, n, p)
        tri_rec(A, n, m)  # On trie avant A[m]
        tri_rec(A, m+1, p)  # On trie apres A[m]

def tri_rapide(A):
    # A completer !
```

Je la complète par : `tri_rec(A, 0, len(A))`

5. ***Construire une fonction*** `temps_rapide(n)` ***donnant le temps de calcul du tri rapide pour une liste remplie de*** n ***nombres aléatoires.***

Imports :

```
> main.py
1 import time as t
2 import random as r
```

`temps_rapide` :

```
> main.py
1 def temps_rapide(n):
2     A = [r.randint(0, 50) for k in range(n)]
3     t1 = t.time()
4     tri_rapide(A)
5     return t.time()-t1
```

6. ***Vérifier expérimentalement que le temps de calcul du tri rapide est proportionnel à*** $nlog(n)$ ***pour*** $n \in [200, 500, 1000, 5000, 10000, 100000]$

Imports

```
> main.py
1 import numpy as np
2 import matplotlib.pyplot as plt
```

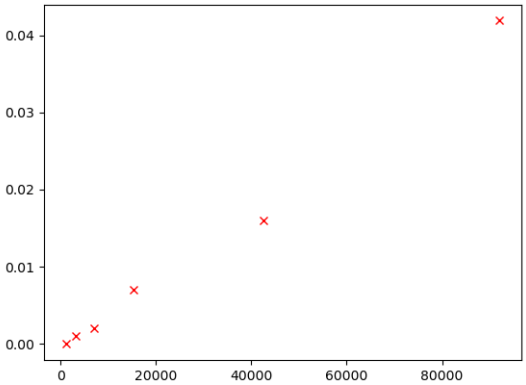
`traceTemps`

```
> main.py
1 def traceTemps(N):
2     Y = []
3     for n in N:
4         Y.append(temps_rapide(n))
5     plt.plot(N*np.log(N), Y, 'rx')
6     plt.show()
```

Je le teste, mais pour `n = 10000`, il y a une erreur python : on a fait trop de tour de boucle récursive...

```
> main.py
1 N = [200, 500, 1000, 2000, 5000, 10000]
2 traceTemps(N)
```

Et j'obtiens :



C'est presque une droite, ce qui prouve expérimentalement la comlexité en $n \cdot log(n)$.

2. Exercices

1. (a) ***Ecrire une fonction*** `temps(n)` ***qui renvoie le rapport des temps*** $\frac{t_i(n)}{t_r(n)}$ ***pour une même liste*** T_0 ***de*** n ***nombres aléatoires entiers où*** $t_i(n)$ ***est le temps du tri par insertion et*** $t_r(n)$ ***le temps du tri rapide.***

> exercices.py

```
1  from time import perf_counter
2
3  def temps(n):
4      A = [r.randint(0, 50) for k in range(n)]
5      B = A[:]
6      t1 = perf_counter()
7      tri_rapide(A)
8      t_rap = perf_counter() - t1
9
10     t1 = perf_counter()
11     tri_insertion(B)
12     t_ins = perf_counter() - t1
13
14     return (t_ins/t_rap)
```

(b) ***Vérifier que*** $10 \leq$ `temps(1000)` ≤ 20

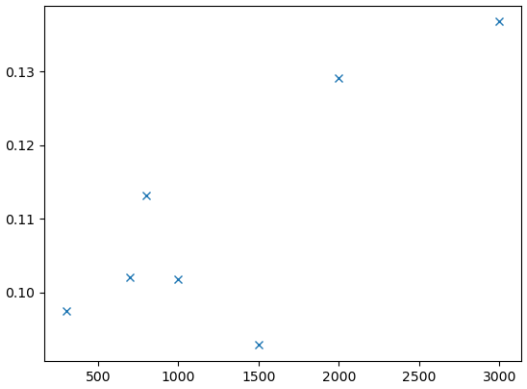
`print(temps(1000))` donne : `15.828425459447944`

(c) ***Vérifier graphiquement en faisant varier*** n ***dans l'ensemble*** `np.array([300,700,800,1000,1500,2000])` ***qu'on a la relation :*** $\frac{t_i(n)}{t_r(n)} \underset{+\infty}{\sim} C\frac{n}{ln(n)}$ ***et donner la valeur de la constante*** C***.***

> exercice.py

```
1  def graphe():
2      x = np.array([300, 700, 800, 1000, 1500, 2000, 3000])
3      y = [temps(n)*np.log(n)/n for n in x]
4      plt.plot(x, y, 'x')  # donne la constante C
5      plt.show()
```

Mais lorsque j'exécute `graphe()`, je n'obtiens pas une droite...même en répétant l'opération...



2. ***Ecrire une procédure*** `tri_insertion_inv(T)` ***qui trie la liste*** `T` ***dans le sens décroissant par insertion.***

Ok, je suppose que ce n'était pas l'exercice, mais :

> exercice.py

```
1  def tri_insertion_inv(T):
2      tri_insertion(T)
3      return T.reverse()
```

3. ***Ecrire une fonction*** `pascal(n)` ***qui donne le triangle de Pascal avec*** n ***lignes*** n ***colonnes donc contenant*** $\begin{pmatrix} i \\ j \end{pmatrix}$ ***pour les lignes*** $i \in [\![0, n-1]\!]$ ***et les colonnes*** $j \in [\![0, n-1]\!]$***.***

```
> exercice.py
```

```python
def pascal(n):
    A = np.zeros((n, n), int)
    for k in range(n-1):
        A[k, 0] = 1
        for j in range(1, n):
            A[k+1, j] = A[k, j]+A[k, j-1]
    A[n-1, 0] = 1
    return A
```

4. ***Ecrire alors une fonction*** `tri_insertion_tab(A)` ***qui renvoie le tri d'un tableau*** `A` ***carré trié dans le sens décroissant en plaçant les coefficients dans l'ordre des diagonales montantes.***

```
> exercice.py
```

```python
def tri_insertion_tab(A):
    n = np.shape(A)[0]
    T = []
    for k in range(n):
        T.extend(A[k])
    tri_insertion_inv(T)
    B = np.zeros((n, n), int)
    for d in range(2*n-1):
        for j in range(0, n):
            i = d-j
            if i >= 0 and i <= n-1:
                B[i, j] = T[0]
                T.remove(T[0])
    return B
```

En exécutant `print(tri_insertion_tab(pascal(10)))`, j'obtiens :

```
[126  84  56  35  21   9   6   2   1   1]
[126  70  36  21  10   6   3   1   1   0]
[84  36  28  10   7   3   1   1   0   0]
[56  28  15   7   4   1   1   0   0   0]
[35  15   8   4   1   1   0   0   0   0]
[20   8   5   1   1   0   0   0   0   0]
[9   5   1   1   0   0   0   0   0   0]
[6   1   1   0   0   0   0   0   0   0]
[1   1   0   0   0   0   0   0   0   0]
[1   0   0   0   0   0   0   0   0   0]
```