TD 8 : Algorithmes de Tri

PC* - Lycée Thiers

Exercice 1 : Tri rapide non en place

1. Ecrire en `python` l'algorithme de tri rapide vu en cours.
2. Mesurer son temps d'exécution sur un tableau de nombres aléatoires, sur le modèle :

```
In[1]: from random import random
In[2]: T = [100*random() for k in range(1000)]
In[3]: %timeit triRapide(T)
```

Exercice 1 : corrigé

- Algorithme de tri rapide :

```
# Tri rapide
def tri_rapide(T):
    N = len(T)
    if N <= 1:      # Appel terminal
        return T
    e = T.pop(N//2)      # Retirer l'élément au milieu du tableau
    T1, T2 = [ ], [ ]
    for x in T:      # Constitution de T1 et T2
        if x <= e:
            T1.append(x)
        else:
            T2.append(x)
    return tri_rapide(T1) + [e] + tri_rapide(T2)      # Appel rec.
```

Exercice 1 : corrigé

2)

```
In [1]: from random import random
In [2]: L=[100*random() for k in range(1000)]
In [3]: %timeit tri_rapide(L)
100 loops, best of 3: 1.67 ms per loop
```

Exercice 2 : Tri rapide en place

But : écrire une version "en place" du tri rapide.

Pour cela écrire une fonction `partition(T,debut, fin, pivot)` qui :

- prend en paramètre le tableau `T`, les indices du début (`debut`), de la fin (`fin`), et du pivot (`pivot` qui devra être entre `debut` et `fin`),
- réordonne les éléments de `T` entre les indices `debut` et `fin` de sorte que : les éléments inférieurs (respectivement supérieurs) au pivot lui soient placés avant (respectivement après).
- Renvoie l'indice du pivot. Pour celà :

- `e` ← `T[pivot]`
- échanger les positions de `T[pivot]` et `T[fin]`
- `j` ← `debut`

Après chaque passage dans la boucle for suivante, `T[j]` sera le premier élément $\geqslant$ `e` parmi ceux parcourus.

- pour `i` variant de `debut` à `fin - 1` :

  Si `T[i] < e` alors :
  échanger `T[i]` et `T[j]`
  `j` ← `j+1`

- échanger `T[fin]` et `T[j]`

Exercice 2 : Tri rapide en place

après chaque passage dans la boucle :

elts < pivot	T[j]⩾ pivot	elts ⩾ pivot	T[i]	·····	**pivot**

à la fin de la boucle for :

elts < pivot	T[j]⩾ pivot	elts ⩾ pivot	**pivot**

à la fin :

elts < pivot	**pivot**	elts ⩾ pivot	T[j]⩾ pivot

```
def partitionner(T,debut,fin,pivot):
    e = T[pivot]
    T[pivot], T[fin] = T[fin], T[pivot]
    j = debut
    for i in range(debut,fin):
        if T[i] < e:
            T[i], T[j] = T[j], T[i]
            j = j+1
    T[fin], T[j] = T[j], T[fin]
    return j
```

Exercice 4 : Tri rapide en place

```
def tri_rapideRec(T, debut, fin):
    if debut < fin:
        pivot = (debut+fin)//2
        pivot = partitionner(T,debut,fin,pivot)
        tri_rapideRec(T,debut,pivot-1)
        tri_rapideRec(T,pivot+1,fin)
    return T

def tri_rapide(T):
    return tri_rapideRec(T,0,len(T)-1)
```

On travaille à chaque appel récursif sur la portion de `T` allant de l'indice `i` inclus à l'indice `j` exclu.

Exercice 4 : Calcul de la médiane en place

1. Ecrire l'algorithme de recherche de la médiane d'un tableau de nombres basé sur le tri rapide, vu en cours, sur le modèle :

```
recherche(T,s):
    Soit e=T.pop(len(T)//2)
    Soient T1, T2 obtenu par tri rapide avec pivot e.
    Si len(T1)==s : renvoyer e.
    Sinon, si len(T1)>s : renvoyer recherche(T1,s)
    Sinon (len(T1)<s) : renvoyer recherche(T2,s-len(T1)-1)
```

2. Ecrire une version "en place" du calcul de la médiane par tri rapide.

Exercice 4 : Calcul de la médiane en place

1.

```
def recherche(T,s):
    e = T.pop(len(T)//2)
    T1, T2 = [], []
    for x in T:
        if x<=e:
            T1.append(x)
        else:
            T2.append(x)
    if len(T1) == s:
        return e
    elif len(T1) > s:
        return recherche(T1,s)
    else:
        return recherche(T2,s-len(T1)-1)

def mediane(L):
    T = L[:]    # Copie superficielle pour ne pas modifier L
    return recherche(T,len(T)//2)
```

Exercice 4 : Calcul de la médiane en place

2. On aura encore besoin de la fonction `partition` :

```
def partitionner(T,debut,fin,pivot):
    e = T[pivot]
    T[pivot], T[fin] = T[fin], T[pivot]
    j = debut
    for i in range(debut,fin):
        if T[i] < e:
            T[i], T[j] = T[j], T[i]
            j = j+1
    T[fin], T[j] = T[j], T[fin]
    return j
```

Exercice 4 : Calcul de la médiane en place

```
def recherche(T, debut, fin, s):
    pivot = (debut+fin)//2
    pivot = partitionner(T,debut,fin,pivot)
    if pivot == s:
        return T[pivot]
    elif pivot > s:
        return recherche(T,debut,pivot-1,s)
    else:
        return recherche(T,pivot+1,fin,s)

def mediane(L):
    T = L[:]
    return recherche(T,0,len(T)-1,len(T)//2)
```

Exercice 5 : calcul de médiane en temps linéaire

L'algorithme décrit permet de déterminer la médiane, et plus généralement le k-ième élément d'un tableau, sans le trier intégralement. Il est linéaire dans le pire des cas.
Nous allons écrire une fonction `partile(L,i)` qui retourne l'élément d'indice i dans ce que serait le tableau L après avoir été trié.

1. Ecrire une version du tri par insertion qui s'applique à un sous-tableau ; elle prendra trois arguments, une liste `L` et deux entiers `i` et `j` et triera (en place) son sous-tableau allant de l'indice `i` inclus à l'indice `j` exclu.
2. Ecrire une fonction qui prend en paramètre un tableau et des indices `i`, `j` délimitant 5 éléments et qui retourne la médiane du sous-tableau de ces 5 éléments.
3. Ecrire une fonction qui prend en argument un tableau quelconque, le divise en groupes de cinq éléments (ou moins) et retourne la liste des médianes de chacun de ces groupes.
4. Modifier cette fonction pour qu'elle s'appelle récursivement sur le tableau des médianes construit.
5. Ecrire une fonction qui effectue une partition en 3 sous-listes (inférieur/médiane des médianes/supérieur) du tableau de départ avec pour pivot la médiane des médianes obtenue à l'aide de la fonction précédente.
6. Pour trouver le k-ième élément où faut-il le chercher en fonction de la taille des deux sous-tableaux donnés par la partition. Programmer l'appel récursif correspondant dans une fonction `partile(L,k)`.

Exercice 5 : correction

1) Tri par insertion sur le sous-tableau de `T` allant de l'indice `i` inclus à l'indice `j` exclu :

```
def tri_ins(T,i,j): # de l'indice i inclu à j exclu
    for k in range(i,j):
        x = T[k]
        pos = k
        while pos > i and T[pos-1] > x :
            T[pos] = T[pos-1]
            T[pos-1] = x
            pos = pos-1
    return T
```

Exercice 5 : correction

2)

```
def mediane5(L,i,j):
    tri_ins(L,i,j)
    return L[i+2]
```

3)

```
def tab_mediane(L):
    N = len(L)
    Med = [ ]
    for i in range(0,N,5):
        if i+5 <= N:
            Med.append(mediane5(L,i,i+5))
        else:
            tri_ins(L,i,N)
            Med.append(L[i+(N-i)//2])
    return Med
```

Exercice 5 : correction

4)

```
def medianeMed(L):
    if len(L) == 1: return L[0]
    return medianeMed(tab_mediane(L))
```

5)

```
def partition(L):
    G, D = [], []
    e = medianeMed(L)
    L.remove(e)
    for x in L:
        if x <= e:
            G.append(x)
        else:
            D.append(x)
    return [e], G, D
```

Exercice 5 : correction

6)

```
def partile(L,k):
    assert k <= len(L)
    L0, Lg, Ld = partition(L)
    N = len(Lg)
    if N == k-1:
        return L0[0]
    if N > k-1:
        return partile(Lg,k)
    else:
        return partile(Ld,k-N-1)
```