2 T.P.2 PROCEDURES DE TRIS

1 Introduction

Il s'agit dans ce TP de présenter quatre méthodes usuelles pour trier une liste de façon **croissante**.
Une fonction Python, déjà utilisée dans le chapitre de statistiques, permet de faire de tels tris, croissants ou décroissants.
Notre objectif est de pouvoir, lors d'épreuves d'écrit ou d'oral, réécrire les fonctions qui permettent de faire cette même opération ainsi que le recommande le programme officiel.
Les procédures à connaître sont au nombre de quatre : Il s'agit du *tri par sélection*, du *tri par insertion*, du *tri à bulles* vus en première année et enfin du *tri rapide* connu également sous le nom de *quicksort* au programme de deuxième année.

2 Les méthodes Python

Si `L` est une liste. Il est possible de les trier par ordre croissant ou décroissant.
`L.sort()` : Tri croissant de la liste.
`L.sort(reverse=True)` : Tri décroissant de la liste.

✎ Toutes ces *méthodes* font des listes des objets **mutables**. On ne créé pas grâce à elles de nouveaux objets, on se contente de les modifier (Pour `L.sort()` on perd la liste initiale... Si on veut garder la liste originale, il faut commencer par la copier ou écrire `L1=sorted(L)`.)
`L1=list(L)` ou `L1=L[:]` : Copie de la liste.

Il est possible de trier un tableau selon un ligne ou une colonne. Pour cela, on créera une « clef » qu'on passera en paramètre dans la méthode `sort` appliquée à la liste à trier.

Exemple : Imaginons des points donnés par leurs coordonnées GPS avec latitude et longitude. On souhaite trier ces points selon leurs longigudes...
Si `coordonnees = [(43,7,'A2'),(46,-7,'A1'),(45,0,'A3')]`

```
def longitude(point):# création de la clef
    return point[1]

coordonnees.sort(key=longitude) # opère le tri selon les longitudes et retourne :
```

`[(46,-7,'A1'),(45,0,'A3'),(43,7,'A1')]`

✎ Il est aussi possible de trier des listes formées de caractères. Il faut pourtant, dans ce cas, avoir conscience de certains pièges...

`L=['biologie','chimie','science','terre','bcpst']`
`L.sort()` retourne la liste `L=['bcpst','biologie','chimie','science','terre']`
mais
`L=['biologie','Chimie','science','terre','bcpst']`
`L.sort()` retourne la liste `L=['Chimie','bcpst','biologie','science','terre']`
ou encore
Si `L=['zèbre','mouton','ane']`, alors `L.sort()` retourne `L=['ane','mouton','zèbre']` mais
si `L=[' zèbre','mouton','ane']`, alors `L.sort()` retourne `L=[' zèbre','ane','mouton']`

3 Tri par sélection

On parle parfois de « tri naïf »parce que c'est l'une des plus intuitives.

Soit L la liste à trier de longueur n. L'idée est de commencer par chercher le plus grand élément de la liste puis de le placer à la n-ième position en l'échangeant avec le dernier élément de celle-ci.
Il suffit alors de recommencer l'opération avec les $n-1$ premiers éléments puisque le dernier élément de la liste est correctement placé. On réitère cette opération jusqu'à ce qu'il ne reste plus que deux éléments à trier.

① Écrire une fonction `maxi(L,i)` qui prend en argument une liste `L` et un entier `i` et qui retourne le maximum **et** son indice parmi les i premiers éléments de la liste.

② En utilisant la fonction `maxi`, écrire une fonction `tri_selec` qui trie une liste selon la méthode décrite ci-dessus.

solution - tri par sélection

```
def maxi(l,n)
    indice=0
    for i in range(n):
       if l[i]>l[indice]:
          indice=i
    return indice
```

```
def tri_select(l):
    i=len(l)-1
    while i>0:
       j=maxi(l,i+1)
       if j != i:
          l[j],l[i]=l[i],l[j]
       i -= 1
    return l
```

4 Tri à bulles

Cette méthode consiste à parcourir une première fois l'ensemble du tableau et à faire « remonter » son plus grand élément en dernière place à l'aide d'échanges deux à deux.
On recommence ensuit cette opération sur les $(n-1)$ premiers termes du tableau, les $(n-2)$ premiers termes, etc. jusqu'aux deux premiers termes d'indices 0 et 1.

Exemple : On considère la liste `L=[4,6,3,7,1]`. On obtiendra successivement :

première étape = première « bulle » :
`[4,6,3,7,1]` # L[0]<L[1] : on ne fait rien
`[4,3,6,7,1]`# L[1]>=L[2] : on a échangé 3 et 6
`[4,3,6,7,1]` # L[2]<L[3] : on ne fait rien
`[4,3,6,1,7]` # L[3]>=L[4] : on a échangé 1 et 7

seconde étape = deuxième « bulle » : on s'intéresse aux 4 premiers termes
`[3,4,6,1,7]`# L[0]>=L[1] : on a échangé 4 et 3
`[3,4,6,1,7]`# L[1]<L[2] : on ne fait rien
`[3,4,1,6,7]`# L[2]>=L[3] : on a échangé 6 et 1

troisième étape = troisième « bulle » : on s'intéresse aux 3 premiers termes
`[3,4,1,6,7]`# L[0]<L[1] : on ne fait rien
`[3,1,4,6,7]`# L[1]>=L[2] : on a échangé 4 et 1

quatrième étape = 4-ième « bulle » : on s'intéresse aux 2 premiers termes
`[1,3,4,6,7]`# L[0]>=L[1] : on a échangé 3 et 1

```
def tri_bulles(l):
   n=len(l)
    for k in range(n-1,0,-1):# k de n-1 à 1 par pas de -1
       for i in range(k): # i de 0 à k-1
          if l[i]>l[i+1]:
             l[i],l[i+1]=l[i+1],l[i] # échange
    return l
```

5 Tri par insertion

Cette méthode de tri est celle du joueur de carte qui reçoit successivement les cartes qui lui sont distribuées. Une fois la première carte prise, il insère la seconde à sa place pour un tri croissant. Ensuite, à chaque nouvelle carte, il l'insère dans l'ordre en plaçant dans sa main triée chaque élément à sa bonne place.

Pour programmer un tel tri, on traite les éléments de la liste tour à tour. On les compare aux éléments précédents préalablement triés jusqu'à trouver la sa place. Il suffit alors de décaler les éléments du tableau pour insérer l'élément considéré à sa place.

Exemple : On considère la liste `L=[4,6,3,5,7,1]`. On obtiendra successivement :

```
[4, ,6,3,5,7,1]
[4,6, ,3,5,7,1]
[3,4,6, ,5,7,1]
[3,4,5,6, ,7,1]
[3,4,5,6,7, ,1]
[1,3,4,5,6,7, ]
```

① Écrire une fonction `insertion(L,i)` qui prend en argument une liste `L` dont on suppose que les $(i-1)$ premiers éléments sont triés et qui insère l'élément `L[i]` à sa place pour que les i premiers éléments de `L` soient triés.

② Écrire une fonction `tri_insert(L)` qui effectue un tri par insertion de la liste `L`.

```
def insertion(L,i):
   while i>0 and L[i]<L[i-1]:
         L[i-1],l[i]=L[i],L[i-1]
         i -= 1
   return L
```

```
def tri_insert(L):
    n=len(L)
    for i in range(1,n):
       L=insertion(L,i)
    return L
```

6 Tri rapide - quickSort

Le *tri rapide* est une méthode récursive de tri.
Elle consiste à prendre un élément au hasard (le plus souvent le premier de la liste) appelé *pivot* et de le mettre à sa place en plaçant tous les éléments plus petits à sa gauche et les plus grands à sa droite.
On recommence alors sur les deux sous-listes obtenues jusqu'à obtenir des sous-listes vides. La liste est alors triée.

Exemple :

- On suppose `L=[15,19,3,20,5,1,6,11]`
- Le pivot est 15. On place dans `L1` les éléments qui lui sont plus petits, dans `L2` sont qui lui sont plus grands.

$$[3,5,1,6,11] + \boxed{[15]} + [19,20]$$

- On répète la même opération sur les deux sous-listes `L1` et `L2` :

$$[1] + \boxed{[3]} + [5,6,11] + \boxed{[15]} + \boxed{[19]} + [20]$$

- On termine en traitant de la même manière les trois sous-listes restantes.

Exercice : Écrire une fonction Python récursive `triRapide(L)` permettant de trier une liste `L` selon cette procédure.

```
def trirapide(L):
    if L==[]:
       return []
    else:
       n=len(L)
       L1=[]
       L2=[]
       for k in range(1,n):
          if L[k]<=L[0]:
             L1.append(L[k]) #L1 reçoit les éléments les plus petits
          else:
             L2.append(L[k]) # L2 reçoit les éléments les plus grands
       L=trirapide(L1)+[L[0]]+trirapide(L2)
       return L
```

7 Application : recherche du k-mère le plus fréquent

Considérons à titre d'exemple la recherche des 2-mères les plus fréquents au sein de la séquence

`S='AAGCAAAGGTGGG'`

– On commence par lister tous les 2-mères parmi les $4^2 = 16$ possibles, dans l'ordre d'apparition dans la chaîne de caractère `S`.

2-mères	AA	AG	GC	CA	AA	AA	AG	GG	GT	TG	GG	GG

– On associe à chacun des seize 2-mères possibles un entier naturel de la façon suivante :

2-mères	AA	AC	AG	AT	CA	CC	CG	CT	GA	GC	GG	GT	TA	TC	TG	TT
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

L'idée est d'associer à chaque lettre une valeur entre 0 et 4.
Ici, on a choisit : $A : 0$, $C : 1$, $G : 2$, $T : 3$.
Alors : $AA = 0 * 4^1 + 0 * 4^0 = 0$, $CA = 1 * 4^1 + 0 * 4^0 = 4$ ou encore $GC = 2 * 4^1 + 1 * 4^0 = 9$
Tandis que pour un 3-mère : $GCT = 2 * 4^2 + 1 * 4^1 + 3 * 4^0 = 39$

Écrire une fonction `motif2Nb` permettant d'associer à chaque mot de k lettre un entier naturel formé selon cette procédure.

– On convertit chaque 2-mère en un entier grâce à la fonction `motif2Nb` pour produire un tableau `index` comme ci-dessous :

2-mères	AA	AG	GC	CA	AA	AA	AG	GG	GT	TG	GG	GG
Index	0	2	9	4	0	0	2	10	11	14	10	10

– On trie de façon croissante la liste `index` pour former une liste `indexTrie`.
Sur l'exemple précédent, on obtient :

2-mères	AA	AA	AA	AG	AG	CA	GC	GG	GG	GG	GT	TG
Index Trié	0	0	0	2	2	4	9	10	10	10	11	14

– Comme les k-mères identiques sont accolés dans l'index trié (comme par exemple $(0, 0, 0)$ ou $(10, 10, 10)$) les k-mères les plus fréquents sont révélés par la plus longue chaîne d'entiers égaux dans la liste `indexTrie`
Il suffit alors de créer une liste `compte` qui dénombre les apparitions consécutives d'un même k-mère à partir de `indexTrie` puis d'en rechercher le maximum.

2-mères	AA	AA	AA	AG	AG	CA	GC	GG	GG	GG	GT	TG
Index Trié	0	0	0	2	2	4	9	10	10	10	11	14
Compte	1	2	3	1	2	1	1	1	2	3	1	1

– Enfin il faut pouvoir faire l'opération inverse de la fonction `motif2Nb` pour pouvoir retourner, à partir d'un entier naturel, le motif correspondant de longueur k.
Ecrire une fonction `nb2Motif` dont les variables d'entrée sont deux entiers n et k et retournant le motif associé.

– L'encadré ci-dessous présente l'algorithme permettant de rechercher les mots les plus fréquents de k lettres dans un `Texte` donné.

```
mots_frequents_par_tri(Texte,k)
    motsFrequents ← un ensemble vide
    Pour i ← 0 à |Texte| - k
       motif ← Texte(i,k)
       index(i) ← motif2Nb(motif)
       compte(i) ← 1
    indexTrie ← triRapide(index)
    Pour i ← 1 à |Texte| - k
       Si indexTrie(i) = indexTrie(i-1)
          compte(i) ← compte(i-1)+1
    compteMax ← valeur maximum dans compte
    Pour i ← 0 à |Texte| - k
       Si compte(i) = compteMax
          motif ← nbr2Motif(indexTrie(i),k)
          ajouter motif à l'ensemble motsFrequents
    return motsFrequents
```

Exercice : Traduisez cette algorithme en Python et en faire l'application à la recherche des 9-mères les plus fréquents dans *oriC* de *vibrio Cholerae*.