Méthodes de tri

Données
- Une liste L de flottants de longueur n.
- But : trier ses éléments par ordre croissant.

I Tri cousu main

Principe. C'est le tri du désespoir. Si vous avez oublié les méthodes vues cette année, faites-le ainsi. Il repose sur le fait que dans une liste triée, le premier élément est son minimum.

Algorithme.

a. on cherche donc la position `j` dans `L` du minimum `L[j]` de la liste `L`. Il sera donc le premier élément de la liste triée `LT` (supposée) initialement vide. Cette recherche est réalisée avec la fonction auxiliaire `position_min(L)`.

b. On place alors `L[j]` dans `LT` puis on l'ôte de la liste `L`.

c. On répète la procédure sur la liste `L` privée de `L[j]` jusqu'à épuiser tous les éléments de `L`

```
def position_min(L):
    n = len(L)
    mini = L[0]       # min temporaire
    j = 0             # j temporaire
    for i in range(n):
        if L[i]<mini:
            mini = L[i]
            j = i
    return j
```

```
def tri_basique(L):
    M = L[:] # copie pour ne pas modifier L
    LT = [ ]
    n  = len(M)
    for i in range(n):
        j = position_min(M)
        LT.append(M[j])
        M.pop(j)
    return LT
```

II Tri à bulles (bubblesort)

Principe. Il repose sur le fait qu'une suite finie $(u_j)_{0\leq j\leq n-1}$ est triée si et seulement si :

$$\forall j \in \{0, \dots, n-2\} \qquad (P_j) : \quad u_j \leq u_{j+1}$$

Algorithme.

a. On effectue un parcours des éléments de la liste `L`, et chaque fois que l'on trouve lors de ce parcours deux termes consécutifs `L[j]`, `L[j+1]` ne vérifiant pas la propriété (P_j), on les permute. Cette dernière sera réalisée avec la fonction auxiliaire `permute(L,j)`.

b. Au bout du premier parcours, le plus grand élément de la liste est nécessairement le dernier. On en conclut deux choses :

1) Si la liste est de longueur n, au bout d'un passage, on se retrouve à trier une liste de longueur $n-1$ à l'étape suivante.

2) Au parcours suivant, il est inutile d'examiner la dernière paire puisque cette dernière n'aura pas à être permutée.

Ce qui donne les scripts suivants :

```
def permute(L,i):
    M = L[:]
    M[i],M[i+1] = M[i+1],M[i]
    return M
```

```
def tribulles(L):
    M = L[:]
    n = len(L)
    for k in range(n-1):       # cf. b.1)
        for j in range(n-1-k): # cf. b.2)
            if M[j]>M[j+1]:
                M = permute(M,j)
    return M
```

III Tri par insertion

Principe. C'est la méthode la plus intuitive de tri, puisque c'est celle que vous utiliseriez naturellement si vous deviez trier les cartes de votre main dans une partie de belote ou de tarot.

Algorithme.

a. On examine les éléments de la liste à partir du début en supposant qu'à chaque étape, tous les éléments situés **avant** l'élément considéré (appelé **pivot**) sont triés correctement.

b. À la première étape, le pivot est le premier élément de la liste.

c. À chaque étape, on compare le pivot p à l'élément z qui le **précède** dans la liste. Le but est d'insérer p correctement :
- Si $z > p$, c'est que p est mal placé, donc on le rétrograde dans la liste jusqu'à ce qu'il soit en bonne position dans la liste.
- Sinon c'est que p est bien placé.

d. On répète ce procédé avec le terme qui était à droite de p à l'étape précédente (il devient le nouveau pivot).

e. Comme la position du pivot augmente strictement à chaque étape, à la fin tous les éléments auront été des pivots et la liste est triée.

```python
def retrograde(L,i):
    M = L[:]
    i1 = i      # i1 : position finale de L[i]
    while M[i1]<M[i1-1] and i1 > 0:
        M[i1],M[i1-1] = M[i1-1],M[i1]
        i1-=1               # décrémentation
    return M
```

Rem.1 | Ligne 4 : ne pas oublier la condition `i1>0`

```python
def triInsertion(L):
    n = len(L)
    M =L[:]
    for i in range(1,n):
        M = retrograde(M,i)
    return M
```

IV Tri rapide (quicksort)

Le tri rapide est par définition un tri récursif. Voici son implémentation :

```python
def qs(L):
    n  = len(L)
    if n<=1:
        return L
    else:
        pivot = L[0]
        Lgauche =[]
        Ldroite =[]
        for i in range(1,n):
            if L[i] < pivot:
                Lgauche.append(L[i])
            else:
                Ldroite.append(L[i])
    return qs(Lgauche)+[pivot]+qs(Ldroite)
```

V Application au calcul de la médiane d'une statistique univariée

[médiane] **Déf.1**

Soit $x = (x_1 < \cdots < x_n)$ une série statistique de longueur n (triée). La médiane Q_2 de la série est :

$$Q_2 = \begin{cases} x_{1+\frac{n-1}{2}} & \text{si n est impair} \\ \frac{1}{2}\left(x_{\frac{n}{2}} + x_{1+\frac{n}{2}}\right) & \text{si n est pair} \end{cases}$$

Rem.2 | Attention, en Python, la série est numérotée $x_0, \ldots x_{n-1}$

```python
def mediane(L):
    LT = tri_basique(L)
    n  = len(L)
    p = n//2
    if (-1)**n ==-1:   # teste la parité
        return L[p]
    else:
        return 0.5*(L[p]+L[p-1])
```