TD4
Algorithmes de Tri

On veut trier une séquence de nombres entiers par ordre croissant in situ. Les fonctions de tri que vous allez écrire doivent être toutes enregistrées dans un fichier nommé `mesFonctionsTri.py` qui jouera le rôle de module externe. On rappelle que pour importer et utiliser les fonctions de `mesFonctionsTri.py` dans un autre fichier (tel que `TD4.py` par exemple), il suffit que les deux fichiers se trouvent dans le même dossier et d'écrire au début de `TD4.py` :

```
from mesFonctionsTri import *
```

Exercice 1: Tri par sélection

Le tri par sélection consiste à rechercher le plus petit élément de la séquence et à le mettre en position initiale (en l'échangeant avec le 1er élément), puis à réitérer en recherchant le 2eme plus petit élément pour le mettre en 2eme position....

1. Dérouler l'algorithme sur la séquence : (9 3 12 5 1)
2. Ecrire l'algorithme en pseudo-code.
3. Traduire cet algorithme en une fonction `triSelection()` qui prend une liste en argument et ne renvoie rien. Cette fonction va permettre de trier les éléments de la liste par ordre croissant.

Correction

1.
```
[1, 3, 12, 5, 9]
[1, 3, 12, 5, 9]
[1, 3, 5, 12, 9]
[1, 3, 5, 9, 12]
```

2. On peut écrire du pseudo-code à divers niveaux de complexités. On peut par exemple résumer les fonctions essentielles de l'algorithme sans rentrer dans le détail :

```
Algorithme : Tri selection
Arguments : Liste L de taille n
Renvoie : rien (modifie la liste en place)

Pour cpt allant de 0 a n-2
    m ← min(L[cpt:])
    Echanger m et L[cpt]
```

Ici, on voit que l'on est très proche de la description de l'algorithme donnée dans l'énoncé. On ne donne cependant aucune indication de comment le minimum est trouvé à chaque fois ni comment l'échange des valeurs dans le tableau est effectué. On peut donner plus de détails :

```
Algorithme : Tri selection
Arguments : Liste L de taille n
Renvoie : rien (modifie la liste en place)

Pour cpt allant de 0 a n-2
    indMin ← cpt
    Pour k allant de cpt+1 a n-1
        Si L[k] < L[indMin] alors
            indMin ← k
        temp ← L[cpt]
        L[cpt] ← L[indMin]
        L[indMin] ← temp
```

On est ici beaucoup plus proche (voire quasi identique) à une implémentation en code (voir question suivante).

3. ```
   def triSelection(L):
       for cpt in range(len(L)-1):
           # on cherche la position du min de L[cpt:]
           indMin = cpt
           for k in range(cpt+1, len(L)):
               if L[k] < L[indMin] :
                   indMin = k
           L[cpt], L[indMin] = L[indMin], L[cpt]   # on echange le terme a la
       position cpt et le min de L[cpt:]
   ```

   On se rappelle qu'en Python, on peut s'affranchir de l'usage d'une variable `temp` comme on l'a écrit dans le pseudo-code.

Exercice 2 : Tri par insertion

Le tri par insertion consiste à trier les éléments d'une séquence au fur et à mesure de la lecture de cette séquence. Ainsi, à l'issue de l'itération k, les k premiers éléments de la séquence sont triés. On considère alors l'élément $k+1$ de la séquence en l'insérant à la bonne place parmi les k précédents. Cette insertion doit se faire par échanges successifs.

1. Dérouler l'algorithme sur la séquence : (9 3 12 5 1)
2. Traduire cet algorithme en une fonction `triInsertion()` qui prend une liste en argument et ne renvoie rien. Cette fonction va permettre de trier les éléments de la liste par ordre croissant.

Correction

1. ```
   [3, 9, 12, 5, 1]
   [3, 9, 12, 5, 1]
   [3, 5, 9, 12, 1]
   [1, 3, 5, 9, 12]
   ```

2. ```
   def triInsertion(L):
       for i in range(1, len(L)):
           j = i-1
           while j >= 0 and L[j+1] < L[j]:
               L[j], L[j+1] = L[j+1], L[j]   # on echange les elements aux
       positions j et j+1
               j = j-1
   ```
   ```

Exercice 3 : Tri rapide ou quick sort

Le tri rapide est un tri récursif dont l'idée est la suivante : on commence par choisir aléatoirement un élément quelconque de la séquence (appelé pivot), on parcourt une fois le tableau de sorte à séparer les éléments inférieurs, égaux, et supérieurs au pivot. Cela fournit un tableau se découpant en trois parties : une première sous-séquence contenant les éléments inférieurs au pivot, une deuxième sous-séquence contenant les éléments égaux au pivot (dont le pivot lui-même), et une troisième sous-séquence contenant tous les éléments supérieurs au pivot. On recommence cette opération récursivement sur la première et la troisième sous-séquences jusqu'à ce qu'elles soient de longueur 0 ou 1 ; une sous-liste vide ou une liste contenant un unique élément est en effet déjà triée, et cette sous-liste a été placée à la bonne place par rapport aux autres éléments de la liste complète à trier, grâce aux appels récursifs du tri rapide.

1. Dérouler l'algorithme sur la séquence (9 3 12 5 1) si on choisit comme indices de pivot 1 puis 0 à droite.

2. Écrire une fonction qui prend en argument une liste et deux indices `d` et `f`. Cette fonction choisit aléatoirement un indice de pivot entre les indices `d` et `f`, et réorganise la liste indicée de `d` à `f` de façon à classer en premier les éléments inférieurs, puis les éléments égaux, puis les éléments supérieurs au pivot. Cette fonction renvoie deux indices délimitant la séquence des éléments égaux au pivot.

3. Écrire une fonction récursive `triRapideEnPlace()` qui prend en argument une liste et deux indices `d` et `f` et qui trie les éléments de la liste allant de d à f en utilisant la fonction écrite à la question précédente.

4. On veut maintenant écrire une version qui ne fait pas le tri en place, c'est-à-dire qui prend en argument une liste et qui au lieu de ne rien retourner, retourne la liste triée. On n'a alors plus besoin des arguments de début et de fin `d` et `f`. Ecrire une nouvelle version de la fonction précédente nommée `triRapide()` et qui n'effectue pas le tri en place. On devra pour cela utiliser une nouvelle version de la fonction définie dans la question 2, qui cette fois prend en argument seulement une liste, qui choisit aléatoirement un pivot dans cette liste et retourne les 3 listes des éléments inférieurs, égaux et supérieurs au pivot.

5. Tester votre fonction pour trier des listes générées aléatoirement de taille 1000, 10000 et 100000 contenant des entiers entre -50 et 50. Que constatez-vous ?

Correction

1. Le premier pivot est 3, il reste à la position d'indice 1 : `(1 3 9 12 5)`
   A gauche de 3, plus besoin de trier. A droite de 3, c'est-à-dire dans `(9 12 5)`, on choisit le pivot d'indice 0, donc 9. On réorganise pour obtenir `(5 9 12)` et la liste finale est donc `(1 3 5 9 12)`.

2. ```
   import random as rd

   def separationEnPlace(L, d, f):
       # on choisit aleatoirement un pivot
       indPivot = rd.randint(d, f)
       pivot = L[indPivot]

       # on construit les listes avec les termes inferieurs, egaux et superieurs
       au pivot
       moins, egal, plus = [], [], []
       for x in L[d:f+1]:
           if x < pivot:
   ```
   ```

```
            moins.append(x)
        elif x == pivot:
            egal.append(x)
        else:
            plus.append(x)
    Lp = moins + egal + plus

    # on remplace la sous-liste entre d et f
    L[d:f+1] = Lp[:]

    return len(moins), len(moins+egal)
```

3. ```
def triRapideEnPlace(L, d=0, f=-1):
    # pour eviter de donner d=0, f=len(L)-1 systematiquement
    if f == -1:
        f = len(L)-1

    # condition d'arret : la sous-liste est de longueur 0 ou 1
    if f-d <= 1:
        return

    # on place le(s) pivot, on recupere son (ses) indice(s)
    m, p = separationEnplace(L, d, f)

    # on trie recursivement les 2 sous-listes separees par le(s) pivot(s)
    triRapideEnplace(L, d, d+m)
    triRapideEnplace(L, d+p, f)
```

## Exercice 4 : Tri fusion

Le tri fusion exploite le principe suivant : partant de deux listes déjà triées, il est possible de les fusionner de façon obtenir une liste triée en un temps linéaire.

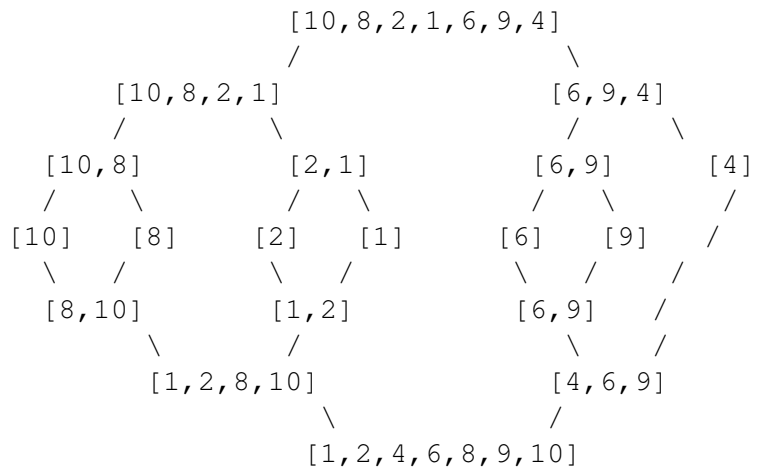
Par exemple, soient `L1=[5,6,10,11]` et `L2=[2,4,9]`.
Le plus petit élément des deux listes est soit `L1[0]`, soit `L2[0]`. On va donc mettre dans `L` (la liste dans laquelle on fusionne les deux listes `L1` et `L2`) l'élément 2. Et on avance dans le parcours de `L2`. Comme `L1[0]` est supérieur à `L2[1]`, on ajoute 4 à `L` et on avance dans le parcours de `L2`. Puis `L1[0]` est inférieur à `L2[2]`, on ajoute `L1[0]=5` à `L` et on avance dans le parcours de `L1`... On s'arrête lorsque l'on a entièrement parcouru les élèments de l'une des deux listes `L1` ou/et `L2`. Pour finir on obtient `L=[2,4,5,6,9,10,11]`.

1. Dans `mesFonctionsTri.py`, écrire une fonction prenant en argument deux listes triées `L1` et `L2` et qui les fusionne en une nouvelle liste triée selon l'algorithme décrit ci-dessus.

L'algorithme de tri fusion se décrit maintenant de façon récursive:

- Si le tableau n'a qu'un élément, il est déjà trié et aucun appel récursif.
- Sinon, on sépare le tableau en son milieu (à 1 près si la longueur est impaire). On trie récursivement les deux parties avec l'algorithme du tri fusion. On fusionne les deux tableaux triés en un tableau trié.

**Exemple** : on trie la liste `L=[10,8,2,1,6,9,4]`. Les appels récursifs seront :
```



2. Dans `mesFonctionsTri.py` et en utilisant la fonction de la question 1, implémenter l'algorithme du tri fusion dans la fonction `triFusion()`.

Exercice 5 : Tri à bulles

Le tri à bulles ou tri par propagation consiste à comparer répétitivement les éléments consécutifs d'un tableau, et à les permuter lorsqu'ils sont mal triés. Il doit son nom au fait qu'il déplace rapidement les plus grands éléments en fin de tableau, comme des bulles d'air qui remonteraient rapidement à la surface d'un liquide.

1. On donne le pseudo-code du tri à bulles :

```
Algorithme : Tri a bulles
Arguments : Liste L de taille n
Renvoie : rien (modifie la liste en place)

Pour i allant de n-1 a 1
    Pour j allant de 0 a i-1
        Si L[j+1] < L[j], alors
            Echanger L[j+1] et L[j]
```

   Dérouler l'algorithme sur la séquence (9 3 12 5 1) et expliquer ce qu'il fait. On tentera notamment d'expliquer pourquoi il se termine bien avec la liste `L` triée en place.

2. Implémenter l'algorithme dans la fonction `triBulles()` dans `mesFonctionsTri.py`.

3. On donne le pseudo-code d'une version optimisée du tri à bulles :

```
Algorithme : Tri a bulles optimise
Arguments : Liste L de taille n
Renvoie : rien (modifie la liste en place)

Pour i allant de n-1 a 1
    fini ← Vrai
    Pour j allant de 0 a i-1
        Si L[j+1] < L[j], alors
            Echanger L[j+1] et L[j]
            fini ← Faux
    Si fini est Vrai, alors
        Fin
```

   Expliquer ce qu'apporte cette modification et implémenter l'algorithme dans la fonction `triBullesOpti()` dans `mesFonctionsTri.py`.

Exercice 6 : Comparaison expérimentale des algorithmes

1. Ecrire une fonction qui prend en argument une valeur entière n et renvoie une liste de taille n. Les éléments de la liste sont générés aléatoirement dans l'intervalle [-1000,1000].

2. Générer aléatoirement des listes de taille 1000 jusqu'à 10000 par pas de 1000. Pour chaque taille, générer 10 listes et calculer les temps moyens d'exécution de chacune des fonctions de `mesFonctionsTri.py` : `triBullesOpti()`, `triSelection()`, `triInsertion()`, `triRapide()`, `triFusion()` et la méthode `sort` de Python (il sera difficile de faire mieux !). Stocker les résultats dans un fichier. L'affichage des résultats dans le fichier doit être de la forme :

Taille	TB	TS	TI	TR	TF	PS
1000	tps en s	tps en s	tps en s	tps en s	tps en s	tps en s
⋮						
10000	tps en s	tps en s	tps en s	tps en s	tps en s	tps en s

3. En utilisant `matplotlib`, afficher sur une même figure les courbes de temps moyen d'éxécution en fonction de la taille pour chaque algorithme et sauvegarder cette figure. On légendera et on donnera un titre à la figure, on mettra l'axe des ordonnées en échelle logarithmique (commande `plt.yscale('log')`) et on nommera les axes.

4. Expliquer les résultats observés en estimant les complexités des algorithmes définis dans les exercices 1 à 5.