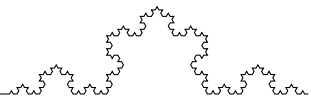
Tris



Dans tout le TP, on suppose que l'on dispose d'une liste L constituée de nombres (entiers ou flottants) dont on souhaite ordonner les éléments par ordre croissant. On note n la longueur de cette liste.
Nous allons étudier différents algorithmes de tris.

♦ Exercice 1 : Tri par sélection

Principe : Le tri par sélection est l'un des tris les plus instinctifs.
Il consiste à trouver la position du plus grand élément ; le plus grand élément est alors échangé avec le dernier élément de la liste. Puis, on recommence ces opérations sur les $(n-1)$ premiers éléments de la liste, et ainsi de suite ... jusqu'à ce qu'il n'y ait plus qu'un élément à trier ; celui-ci est alors le plus petit élément de la liste.

Exemple :

♦ Par exemple, si `L=[6,2,8,5,4,1]`, alors on obtient successivement :

(les cases en gris correspondent aux éléments bien positionnés)

6	2	8	5	4	1	on échange « 1 » et « 8 »
6	2	1	5	4	8	on échange « 4 » et « 6 »
4	2	1	5	6	8	le « 5 » est à sa place
4	2	1	5	6	8	on échange « 4 » et « 1 »
1	2	4	5	6	8	le « 2 » est à sa place
1	2	4	5	6	8	la liste est triée

1°) Écrire une fonction `rg_max(L)` qui prend en argument une liste `L` et qui renvoie la position dans la liste `L` du plus grand élément.

2°) En utilisant la fonction précédente, écrire une fonction `tri_selection(L)` qui trie la liste `L` selon la méthode de tri par sélection.

♦ Exercice 2 : Tri par insertion

Principe : Le tri par insertion est un autre algorithme « naïf », que la plupart des personnes utilisent naturellement pour trier des cartes (prendre les cartes mélangées une à une sur la table, et former une main en insérant chaque carte à sa place).
Au début, on considère que la liste constituée du seul premier élément est trié ; puis on trie les deux premiers éléments ; ensuite on met le troisième élément à sa place parmi les deux premiers, et ainsi de suite ...
De manière générale, au moment où l'on considère le k-ième élément, les éléments qui le précèdent sont tous déjà triés. Il faut donc trouver le rang où cet élément doit être inséré en le comparant aux autres, puis décaler les éléments afin de pouvoir effectuer l'insertion.

Exemple :

♦ Par exemple, si `L=[6,2,8,5,4,1]`, alors on obtient successivement :

(les cases en gris correspondent aux éléments déjà traités)

6	2	8	5	4	1	
2	6	8	5	4	1	on insère le « 2 » à sa place
2	6	8	5	4	1	on insère le « 8 » à sa place
2	5	6	8	4	1	on insère le « 5 » à sa place
2	4	5	6	8	1	on insère le « 4 » à sa place
1	2	4	5	6	8	on insère le « 1 » à sa place, et la liste est triée

Programmation :
Écrire une fonction `tri_insertion(L)` qui trie la liste `L` selon la méthode de tri par insertion.

♦ Exercice 3 : Tri fusion

Principe : Le tri fusion est un algorithme très efficace, de type récursif.
La méthode consiste à couper la liste en 2 sous-listes de taille similaire.
On trie récursivement chacune des sous-listes, puis on les fusionne. La fusion consiste à parcourir simultanément les 2 sous-listes, et intercaler les élements dans le tableau résultat au fur et à mesure du parcours, en gardant des compteurs du nombre d'éléments déjà placés dans chacune des sous-listes.

Exemple :

Par exemple, si `L=[8,2,6,5,4,1,9,3,0,7]` :

- On coupe en 2 : `A=[8,2,6,5,4]` `B=[1,9,3,0,7]`.
- On trie (récursivement) : `A=[2,4,5,6,8]` `B=[0,1,3,7,9]`
- On fusionne :

A= | 2 | 4 | 5 | 6 | 8 | B= | 0 | 1 | 3 | 7 | 9 |

♦

										na=0, nb=0
0										na=0, nb=1
0	1									na=0, nb=2
0	1	2								na=1, nb=2
0	1	2	3							na=1, nb=3
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	
0	1	2	3	4	5	6	7	8	9	na=5, nb=5

Programmation : Écrire une fonction `tri_fusion(L)` qui trie la liste `L` selon la méthode de tri fusion. On s'assurera que la complexité de la fusion est bien linéaire.

♦ Exercice 4 : Tri rapide

Principe : Le tri rapide (ou « quicksort ») est un algorithme très efficace, de type récursif.
La méthode consiste à choisir un élément au hasard dans la liste, appelé pivot. Ensuite on permute les éléments de telle sorte que tous ceux qui sont inférieurs au pivot soient au début de la liste, ensuite viennent les éléments égaux au pivot, puis enfin tous ceux qui sont strictement supérieurs au pivot. Cette opération s'appelle le partitionnement. Le pivot est alors à sa place définitive.
Puis on recommence sur chacune des deux sous-listes, et ainsi de suite ... jusqu'à obtenir des sous-listes vides ou réduites à un élément.

Exemple :

Par exemple, si `L=[8,2,6,5,4,1,9,3,0,7]`, alors on obtient successivement :

(les cases en bleu correspondent aux pivots choisis pour passer à la ligne suivante et les cases en gris aux éléments bien positionnés)

♦

8	2	6	5	4	1	9	3	0	7	on choisit « 6 » comme pivot
2	5	4	1	3	0	6	8	9	7	le « 6 » est à sa place définitive on recommence avec les deux sous-listes on choisit « 4 » et « 9 » comme pivots
2	1	3	0	4	5	6	8	7	9	et ainsi de suite ...
1	0	2	3	4	5	6	7	8	9	
0	1	2	3	4	5	6	7	8	9	
0	1	2	3	4	5	6	7	8	9	la liste est triée

Programmation :

1°) Écrire une fonction `tri_rapide(L)` qui trie la liste `L` selon la méthode de tri rapide, **sans chercher à faire un tri en place.** Pour cela, on pourra utiliser la syntaxe suivante :

```
L1 = [x for x in L if x < pivot]
```

2°) *(en option, difficile)*
On veut maintenant écrire une fonction de tri rapide en place. Écrire une fonction de partition qui prend en argument une liste L, deux indices $i < j$, le rang du pivot dans $[\![i, j[\![$ et qui effectue la partition de la sous liste $L[i : j]$ et renvoie le rang du pivot une fois à sa place.
Écrire une fonction `tri_rapidebis(L)` qui trie la liste `L` selon la méthode de tri rapide en place.

♦ Exercice 5 : Comparaison des méthodes de tris

La fonction suivante permet de mesurer le temps nécessaire à une fonction `f` pour trier la liste `liste`

```
import timeit
def temps(liste,f):
        test = timeit.Timer(lambda : f(liste))
        temps = test.timeit(1)
        return temps
```

1°) Écrire une fonction `comparaison(n,fonct)` qui prend un entier n en argument et `fonct` une liste contenant les différentes méthodes de tris.
Pour k entre 1 et n et par pas de 10, la fonction va créer une liste `liste` de longeur k et stocker le temps nécessaire pour la trier par chacune des méthodes.
Attention : bien faire une copie (avec deepcopy) avant d'utiliser la fonction `temps` sinon votre `liste` va se retrouver triée dès le premier appel.
On renverra le résultat sous la forme d'une liste contenant une sous-liste regroupant les résultats obtenus pour chacune des méthodes.

2°) Représenter sur un graphe les résultats obtenus à la question précedente. On utilisera `label=fonct[k].__name__` pour légender le graphe.
Représenter également $\log(t)$ en fonction de $\log(k)$ où t est le temps nécessaire pour trier une liste de longeur k. Que peut-on en déduire ?