Informatique - TD 4 | Tri de listes

Le but de ce TD est de (re)faire un petit tour de diverses méthodes de tri ici ***par ordre croissant*** *pour les listes. Les méthodes à connaitre au minimum issues de votre programme de première année sont : le tri par insertion, par sélection et par comptage. On introduit ici le tri rapide (appelé aussi tri par fusion), au programme de la seconde année, qui se fait par récursivité.*
Il faut savoir utiliser au moins une de ces méthodes sans rappel, notamment parmi les méthodes au programme de première année.

Tri de listes

Dans toute cette partie, on estime que l'on dispose d'une liste `L` *dont les éléments sont des réels (non nécessairement entiers et non nécessairement positifs !) dont on veut classer les éléments par ordre croissant. (On pourrait évidemment également s'entrainer à faire les programmes pour des classements par ordre décroissant.) On base ici nos explications (pour presque tous les cas de tri) sur la liste* `L=[12,-3,17,9,1]` *que l'on va trier, par diverses méthodes,* ***sans la modifier !***

Exercice 1: Le tri par comptage 1 : liste d'entiers sans répétitions

Le tri par comptage est une méthode plutôt simple à mettre en oeuvre. Notons ici `C` une nouvelle liste, qui à terme, contiendra les éléments de `L` classés.
En parcourant `L`, on compte pour chaque élément `L[i]` le nombre d'éléments n_i de `L` qui sont strictement plus petits. Alors on aura `C[`n_i`]=L[i]`.
Par exemple, si on veut trier la liste `[12,-3,17,9,1]`, la nouvelle liste `C` qui sera constituée des éléments de `L` classée contiendra successivement : *(les cellules grises sont celles qui viennent d'être mises à leur place)*

début :
`[ , , , , ]`
placer `L[0]`
`[ , , , 12 , ]` $n_0 = 3$. `C[3]=L[0]=12`
placer `L[1]`
`[ -3 , , , 12 , ]` $n_1 = 0$. `C[0]=L[1]=-3`
placer `L[2]`
`[ -3 , , , 12 , 17 ]` $n_2 = 4$. `C[4]=L[2]=17`
placer `L[3]`
`[ -3 , , 9 , 12 , 17 ]` $n_3 = 2$. `C[2]=L[3]=9`
placer `L[4]`
`[ -3 , 1 , 9 , 12 , 17 ]` $n_4 = 1$. `C[1]=L[4]=1`

Établir un programme Python qui :
- initialise une liste `C` à la bonne taille
- parcourt la liste `L`, et pour chaque `L[i]` compte le nombre d'élément de `L` strictement inférieurs à `L[i]` et remplir `C`.
- pour des besoins de vérification de méthode, imprime la liste `C` à chaque étape (à l'aide d'un `print`.)

Exercice 2: Le tri par comptage général : liste quelconque d'entiers

Cette méthode est particulièrement adaptée dans le but d'établir des diagrammes / histogrammes. Supposons donné une liste `L` **d'entiers**. Par exemple :

`L=[3,-2,4,-2,0,1,1,9,8,4,-2]`

Créer une fonction Python qui :
- Créer la liste du nombre d'occurence(s) dans `L` de chaque entier allant de min `L` à max `L`. Ici :

-2	-1	0	1	2	3	4	...	9
↓	↓	↓	↓	↓	↓	↓		↓
`[ 3`	`0`	`1`	`2`				...	`1 ]`

- En déduit la liste triée (par ordre croissant) issue de `L`.

Exercice 3: Le tri par insertion

Le tri par insertion est généralement le tri instinctif que l'on utilise pour classer des documents/des cartes. On prend ici encore l'exemple `L=[12,-3,17,9,1]`. Afin de ne pas prendre le risque de modifier `L`, on prendra soin de commencer par dupliquer `L` afin de ne pas prendre le risque de la modifier. Appelons `C` la copie de `L` et travaillons sur `C`.
1ère étape : On commence par prendre le premier élément de `C` que l'on laisse pour l'instant en position "0" :

début :
`[ 12 , -3 , 17 , 9 , 1 ]` on ne touche pas au premier

2ème étape : On prend le deuxième élément `C[1]` de `C` (ici -3). S'il est inférieur à `C[0]`, on l'insère juste avant. S'il est supérieur, on l'insère juste après. (**Attention** ce n'est pas un échange mais bien une insertion.)

`[ -3 , 12 , 17 , 9 , 1 ]` on a placé le -3 avant le 12

l'indice de 12 est donc décalé !

3ème étape : On prend le troisième élément de `C` et on cherche à le mettre à sa place en parcourant à nouveau les deux éléments déjà triés. S'il est inférieur à l'élément en position "0", on l'insère juste avant. Sinon, s'il est inférieur à l'élement en position "1", on l'insère entre "0" et "1". Sinon, on l'insère juste après.
etc. . . :

```
[ -3 , 12 , 17 , 9 , 1 ]   on insère 17 après 12
```
puis
```
[ -3 , 9 , 12 , 17 , 1 ]   on insère 9 entre -3 et 9
```
puis
```
[ -3 , 1 , 9 , 12 , 17 ]   on insère 1 avant 9
```
c'est fini.

Écrire une fonction `Tri_insertion(L)` qui effectue un tri par insertion de `L`. Faire là encore un `"print"` à chaque étape.

Exercice 4: Le tri par sélection

Le tri par sélection est une méthode passe cette fois-ci par la recherche d'un élément maximum. Elle se fait directement par échanges à l'intérieur de la liste `L` (dont on prendra soin de faire d'abord une copie, appelée ci-dessous `C` afin de ne pas prendre le risque de modifier `L`.) Pour les besoins de l'explication, on note `n=len(L)`.
On rappelle le principe :

1ère étape : On commence par rechercher **le plus grand** élément de la `C`, qui sera ensuite **échangé** avec le dernier. *(Le plus grand est alors à sa place)* et `C` est modifiée.
Par exemple, avec `C=L=[12,-3,17,9,1]` *(les cellules grises sont toutes celles qui ont été mises à leur place)* :

début, `C` :
```
[ 12 , -3 , 17 , 9 , 1 ]
```
1ère étape :
```
[ 12 , -3 , 1 , 9 , 17 ]   on a échangé 17 et 1
```

2ème étape : On cherche le maximum des $n-1$ **premiers éléments** de `C` (i.e. on ne considère pas le dernier élément.) On l'échange ensuite avec l'**avant dernier** élément de `C`. *(Les deux derniers sont alors à leur place.)*

```
[ 9 , -3 , 1 , 12 , 17 ]   on a échangé 12 et 9
```

3ème étape et suite : On cherche le maximum des $n-2$ **premiers éléments** de la nouvelle liste `C` (i.e. on ne considère pas les deux derniers éléments.) On l'échange ensuite avec l'élément d'indice $n-3$ de `L`. *(Les 3 derniers sont alors à leur place.)*
- etc. . .

```
[ 1 , -3 , 9 , 12 , 17 ]   on a échangé 9 et 1
```
puis
```
[ -3 , 1 , 9 , 12 , 17 ]   on a échangé -3 et 1
```
et c'est fini
```
[ -3 , 1 , 9 , 12 , 17 ]
```

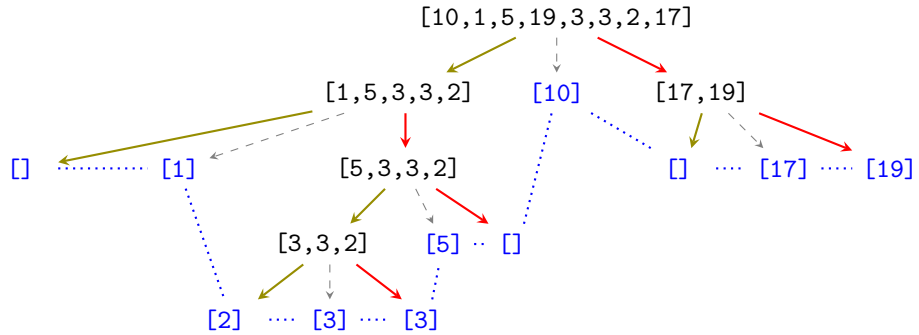
Les questions ci-dessous vont permettre de mettre en place cette stratégie de deux manières différentes.

1. *Sans récursivité et sans utiliser de fonction déjà existante dans Python.*
   a) Créer une fonction `ind_max(L,n)` qui rend l'indice du plus grand élément parmi les n premiers éléments d'une liste `L`.
   b) En utilisant la fonction précédente, créer une fonction `tri_select(L)` qui trie une liste `L` selon la méthode de tri par sélection. Afin de vérifier que c'est bien la bonne méthode qui a été utilisée, on fera un `"print"` à chaque étape pour afficher la liste obtenue à chaque échange.
2. *Sans récursivité et en utilisant des fonctions déjà existante dans Python.*
   On pourra se demander comment faire en utilisant des outils élémentaires déjà présents sous Python. On pourra par exemple utiliser `max(L)` qui rend le maximum d'une liste `L` et l'attribut `L.index(a)` qui rend l'indice du premier élément "a" trouvé dans la liste `L`.
3. *Avec récursivité* (à la limite du programme de Bcpst ; à faire seulement si on a fini la feuille)
   En utilisant le principe de la récursivité, écrire une fonction `tri_select_rec(L)` qui trie directement `L` sans appel à une autre fonction mais en faisant appel à elle-même.

Exercice 5: Tri récursif : Le tri rapide ou "quicksort" ou tri fusion

Le tri rapide est, comme son nom l'indique, une des méthodes les plus rapides. L'idée est de prendre un élément de la liste (pour faire simple : le premier) que l'on appelle *pivot* et de le mettre à sa place définitive en plaçant tous les éléments qui sont plus petits à sa gauche et tous ceux qui sont plus grands à sa droite. On recommence ensuite le tri de la même façon sur les deux sous-listes obtenues.
Par exemple, pour la liste `[10,1,5,19,3,3,2,17]` :
- le premier pivot choisi est `10` .
- On construit la liste des éléments plus petits (strictement) que `10` et de ceux plus grands que `10`, ici : `[1,5,3,3,2]` et `[19,17]` .
- On recommence par récursivité le classement sur les sous-listes `[1,5,3,3,2]` et `[19,17]`, ce qui donne `[1,2,3,3,5]` et `[17,19]`
- on remet les listes classées dans le bon ordre : `[1,2,3,3,5, 10, 17,19]` .

Concrètement, il se passe la chose suivante :



où ici, :

- → : liste “de gauche”
- → : liste “de droite”
- - → : pivot
- [i] : liste réduite
- ····· : liste ordonnée finale

1. Écrire un algorithme qui construit les parties “gauche” et “droite” d’une liste `L` à la façon demandée ci-dessus.
2. En déduire un algorithme triant une liste `L` (toujours sans la modifier) en utilisant l’algorithme ci-dessus
   - **a)** avec le principe de la récursivité.
   - **b)** sans le principe de la récursivité *(seulement si vous avez fini le reste de la feuille)*

Comparaison de la vitesse des méthodes

Exercice 6:

Avec la fonction `time()` du module `time`, on peut calculer la vitesse d’exécution d’une fonction `fct` pour répondre au problème de la manière suivante :

```
1  from time import time
2
3  def vitesse(fct,L):
4      debut=time()
5      fct(L)
6      fin=time()
7      return(fin-debut)
```

1. Écrire une fonction `Liste_alea(n)` qui construit une liste aléatoire de n nombres entiers compris dans $[\![0, 5n]\!]$.
2. Le temps mis par une même fonction pour faire un même calcul n’est pas toujours exactement identique. Construire une fonction `Temps_moyen(fct,n)` permettant de calculer le temps de tri moyen mis par la fonction `fct` pour trier 10 listes de même longueur n.
3. Construire, pour chacun des 4 types de tris vus précédemment dans les exercices 2 à 5, une liste dont les éléments sont les vitesses de tri $T(n)$ pour $n = 10, 50, 100, 100, 2000$.
4. Tracer sur un même graphique (avec la légende associée) la courbe des $T(n)$ en fonction de n pour chacun des 4 types de tris. **Penser à mettre les "print" de vérification en commentaires dans les différentes fonctions, sinon vous allez augmentez le temps de résolution de manière artificielle.** Observer les résultats. Quels semblent être les tris les plus efficaces ?