♦ TP6 – Algorithmes de tri de listes

On s'intéresse ici à des listes de nombres (entiers ou flottants). Le but du TP est de programmer différents algorithmes pour trier ces listes dans l'ordre croissant. Il existe dans Python la fonction `sorted` et la méthode `.sort` qui réalisent ce type de tris mais l'idée ici est évidemment de ne pas les utiliser.

I. — Préliminaires

1) Rappels

Question 1. On considère la liste

```
L = [1.3, 3, -1.4, 5, 6, -7.89, 12, 0, -15]
```

1. Déterminer ce que renvoie les instructions suivantes puis vérifier à l'aide de l'ordinateur.

`L[3]` `L[9]` `L[-1]` `len(L)` `L[2:5]` `L[2:]` `L[:5]`

2. Si on exécute le code suivant, la liste L sera-t-elle modifiée et si oui quelle sera sa nouvelle valeur? Vérifier à l'aide de l'ordinateur.

```
L[3], L[7] = L[7], L[3]
```

3. Déterminer l'affichage obtenu à l'aide du code suivant puis vérifier à l'aide de l'ordinateur.

```
for k in range(-1, -len(L)-1, -1):
  print(L[k])
```

2) Fonctions auxiliaires

La plupart des algorithmes de tri sont basés sur l'échange de deux termes d'une liste. On aura donc besoin d'une fonction effectuant ce travail.

Question 2. Écrire une fonction `echange` qui prend en arguments une liste `L` et deux entiers admissibles `i` et `j`, qui échange les termes `L[i]` et `L[j]` et qui renvoie la nouvelle liste ainsi obtenue.

Vérifier ensuite que `echange(L,3,7)` renvoie bien la liste

```
[1.3, 3, -1.4, 0, 6, -7.89, 12, 5, -15]
```

Dans l'un des algorithmes suivants, nous aurons également besoin de déterminer le maximum d'une liste.

Question 3. Écrire une fonction `maximum` qui prend en argument une liste `L` de nombres et qui renvoie le plus grand élément de `L`.

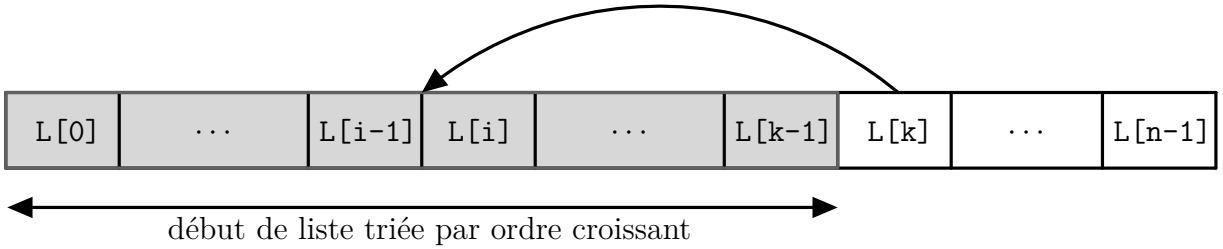
Vérifier ensuite que `maximum(L)` renvoie bien `12`.

II. — Tri par insertion

1) Principe

Considérons une liste de nombres `L` de longueur n. Le principe du tri par insertion est de parcourir la liste et d'insérer chaque élément parmi les éléments de la liste qui le précèdent de sorte à obtenir une sous-liste triée par ordre croissant. De façon plus détaillée,

- **Étape 1.** Initialement, on considère seulement l'élément `L[0]`. Il n'y a pas de comparaison possible donc il n'y a rien à faire ;
- **Étape 2.** Ensuite, on considère `L[0]` et `L[1]`. Si `L[0]` est inférieur ou égal à `L[1]`, les deux premiers éléments de la liste sont dans l'ordre croissant : il n'y a rien à faire. Sinon, on échange `L[0]` et `L[1]` pour les ordonner dans l'ordre croissant.
- On continue ainsi, de sorte qu'à l'issue de l'étape k, les k premiers éléments de la liste sont triés dans l'ordre croissant.
- **Étape $k+1$.** On insère l'élément `L[k]` dans l'ordre croissant parmi les k premiers éléments de la liste, de sorte les $k+1$ premiers éléments de la liste sont alors triés dans l'ordre croissant.



- À l'issue de l'étape n, la liste est triée dans l'ordre croissant.

Appliquons l'algorithme de tri par insertion à la liste `[6, 2, 8, 5, 4, 1]` :

6	2	8	5	4	1	
2	6	8	5	4	1	on insère 2 avant 6
2	6	8	5	4	1	8 est déjà ordonné
2	5	6	8	4	1	on insère 5 entre 2 et 6
2	4	5	6	8	1	on insère 4 entre 2 et 5
1	2	4	5	6	8	on insère 1 avant 2, et la liste est triée

Question 4. En reprenant le modèle de l'exemple ci-dessus (sans la dernière colonne), appliquer à la main l'algorithme de tri par insertion sur la liste `L` de la **Question 1.**

2) Programmation du tri par insertion

Pour programmer le tri par sélection, il faut trouver une façon, à l'étape $k+1$, d'insérer l'élément `L[k]` à sa place dans l'ordre croissant dans la liste formée par les k éléments précédents.

Pour ce faire, on peut procéder de deux manières différentes.

Première méthode : par échanges successifs

- on compare `L[k]` et `L[k-1]` : si le premier est supérieur second, `L[k]` est déjà bien placé et il n'y a rien à faire ; sinon, on les échange et on recommence.
- on compare `L[k-1]` (qui possède à présent la valeur initiale de `L[k]` qu'on cherche à placer) et `L[k-2]` : si le premier est supérieur second, `L[k-1]` est bien placé et on s'arrête ; sinon, on les échange et on recommence.

- on continue tant qu'on n'a pas placé correctement la valeur initialement stockée en `L[k]`.

Question 5. Écrire une fonction `tri_insertion_echanges` qui prend en argument une liste `L` et qui renvoie la liste ordonnée par ordre croissant en utilisant la méthode d'insertion par échanges successifs.

Vérifier ensuite que `tri_insertion_echanges(L)` renvoie bien

```
[-15, -7.89, -1.4, 0, 1.3, 3, 5, 6, 12]
```

Seconde méthode : par décalages successifs

- on détermine le rang `i` que la valeur de `L[k]` doit occuper parmi les $k+1$ premières valeurs de la liste ;
- on stocke la valeur de `L[k]` dans une variable `temp` ;
- on décale successivement d'un cran vers la droite `L[k-1]` puis `L[k-2]` puis `L[k-3]` et ainsi de suite jusqu'à `L[i]` (i.e. on stocke `L[k-1]` dans `L[k]`, puis `L[k-2]` dans `L[k-1]` et ainsi de suite jusqu'à stocker `L[i]` dans `L[i+1]`) ;
- on affecte à `L[i]` la valeur stockée dans `temp`.

Question 6. Écrire une fonction `tri_insertion_decalages` qui prend en argument une liste `L` et qui renvoie la liste ordonnée par ordre croissant en utilisant la méthode par décalages successifs.

Vérifier ensuite que `tri_insertion_decalages(L)` renvoie bien

```
[-15, -7.89, -1.4, 0, 1.3, 3, 5, 6, 12]
```

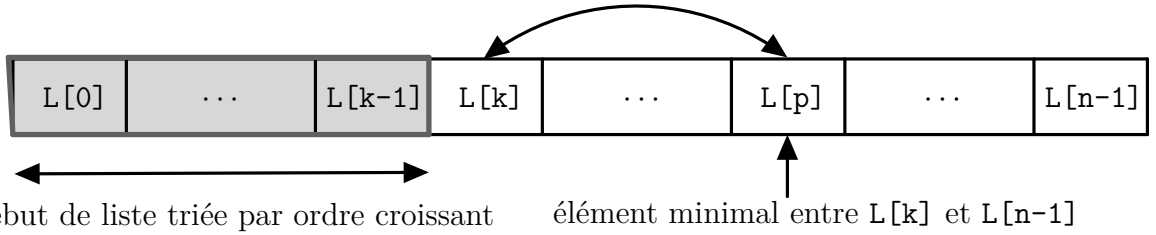
III. — Tri par sélection

1) Principe

La méthode de tri par sélection consiste à déterminer le plus petit élément de la liste et à le placer en tête de liste, puis à chercher le second plus petit élément et à le placer en seconde position de la liste et ainsi de suite.

Plus formellement, si `L` est une liste de nombres de longueur n alors :

- **Étape 1** : on détermine le rang `i` du plus petit élément de `L` puis on échange `L[i]` et `L[0]` : ainsi, (le nouveau) `L[0]` est à présent à sa place ;
- **Étape 2** : on détermine le rang j dans la liste `L` du plus petit élément de `L[1:]` puis on échange `L[j]` et `L[1]` : ainsi (les nouveaux) `L[0]` et `L[1]` sont à présent à leurs places ;
- on recommence ainsi de sorte qu'à l'issue de l'étape k, les k premiers éléments de `L` sont à leurs places ;
- **Étape** $k+1$: on détermine le rang `p` dans la liste `L` du plus petit élément de `L[k:]` puis on échange `L[p]` et `L[k]` : ainsi, les $k+1$ premiers éléments de `L` sont à présent à leurs places ;



- à l'issue de l'étape n, la liste est triée.

Appliquons l'algorithme de tri par insertion à l'exemple de la liste `[6, 2, 8, 5, 4, 1]` :

6	2	8	5	4	①	on échange 1 et 6
1	②	8	5	4	6	2 est bien placé
1	2	8	5	④	6	on échange 8 et 4
1	2	4	⑤	8	6	5 est bien placé
1	2	4	5	8	⑥	on échange 8 et 6
1	2	4	5	6	8	la liste est triée

Question 7. En reprenant le modèle de l'exemple ci-dessus (sans la dernière colonne), appliquer à la main le principe d'insertion sur la liste `L` de la **Question 1.**

2) Programmation du tri par sélection

Question 8. Écrire une fonction `indice_minimum_fin_liste` qui prend en arguments une liste de nombres `L` et un indice admissible `k` et qui renvoie l'indice de la première occurrence du plus petit élément de `L` parmi ceux dont l'indice est supérieur ou égal à `k`. Ainsi, `indice_minimum_fin_liste([3, -2, 4, 7, 1, 3, 1], 2)` doit renvoyer `4` car le plus petit élément de `[4, 7, 1, 3, 1]` est `1` et sa première occurrence dans `L` est au rang `4`.

Question 9. Écrire une fonction `tri_selection` qui prend en argument une liste de nombres `L` et qui renvoie la liste triée par ordre croissant en utilisant la méthode du tri par sélection.

Vérifier ensuite que `tri_selection(L)` renvoie bien

`[-15, -7.89, -1.4, 0, 1.3, 3, 5, 6, 12]`

IV. — Tri par comptage

1) Principe

Ici, on ne va s'intéresser qu'à des listes contenant des entiers naturels. Considérons une telle liste `L` et notons m la valeur du maximum de ses éléments. Ainsi, tous les éléments de la liste appartiennent à $[\![0, m]\!]$.

Le principe du tri par comptage consiste à construire une liste auxiliaire `M` de longueur $m+1$ telle que, pour tout entier `k` compris entre 0 et m, `M`[k] soit égal au nombre d'occurences de l'entier `k` dans la liste `L`.

Ensuite, on construit une liste `P` qui contient `M`[0] fois `0` puis `M`[1] fois `1` puis `M`[2] fois `2` et ainsi de suite jusqu'à `M`[m] fois `m`. Cette nouvelle liste `P` est une version triée dans l'ordre croissant de la liste `L`.

Prenons par exemple la liste

`L = [2, 3, 5, 1, 0, 3, 2, 5, 5, 3, 2, 0, 2]`.

Le maximum des éléments de `L` est $m = 5$. De plus, `0` apparaît 2 fois dans la liste, `1` apparaît 1 fois dans la liste, `2` apparaît 4 fois dans la liste, `3` apparaît 3 fois dans la liste, `4` apparaît 0 fois dans la liste et `5` apparaît 3 fois dans la liste donc `M = [2, 1, 4, 3, 0, 3]`. On en déduit que

`P = [0, 0, 1, 2, 2, 2, 2, 3, 3, 3, 5, 5, 5]`.

On remarquera que, contrairement aux autres, cette méthode ne modifie pas la liste `L` initiale (ce qui peut être un avantage) mais crée deux autres listes `M` et `P` (ce qui peut être un inconvénient en termes de mémoire pour des listes très longues).

2) Programmation du tri par comptage

Question 10. Écrire une fonction `liste_effectifs` qui prend en argument une liste d'entiers naturels `L` et qui renvoie une liste `M` telle que, pour tout entier `k` compris entre 0 et le maximum m de `L`, `M[k]` est égal au nombre d'occurrences de `k` dans `L`. Pour cela, on pourra commencer par créer un liste de longueur $m+1$ constituée de `0` puis parcourir la liste `L` et ajouter `1` à `M[k]` à chaque fois qu'on rencontre l'élément `k` dans `L`. (On utiliser également la fonction `maximum` de la **Question 3**.)

Vérifier qu'avec la liste `L` de l'exemple précédent, on obtient bien `M = [2, 1, 4, 3, 0, 3]`.

Question 11. Écrire une fonction `tri_comptage` qui prend en argument une liste `L` constituée d'entiers naturels et qui renvoie la liste triée par ordre croissant obtenue par la méthode de comptage.

Vérifier qu'avec la liste `L` de l'exemple précédent, on obtient bien

`P = [0, 0, 1, 2, 2, 2, 2, 3, 3, 3, 5, 5, 5]`.

V. — Prolongements

1) Tri à bulles

Considérons une liste `L` de longueur n. Le principe de la méthode du tri à bulles est de parcourir les éléments de la liste $n-1$ fois en échangeant à chaque passage deux éléments consécutifs s'ils ne sont pas dans le bon ordre. Plus précisément,

- **Étape 1 :**
  - ▶ on compare `L[0]` avec `L[1]` et on les échange s'ils ne sont pas dans le bon ordre ;
  - ▶ on compare `L[1]` (qui a éventuellement été modifié à l'étape précédente) avec `L[2]` et on les échange s'ils ne sont pas dans le bon ordre ;
  - ▶ on continue ainsi jusqu'à arriver à `L[n-2]` et `L[n-1]`.

  Lors de ce premier passage, on a nécessairement rencontré le plus grand élément de la liste et, celui-ci étant toujours supérieur aux termes suivants dans la liste, il a donc été systématiquement permuté. Ainsi, à la fin de cette première étape, le plus grand élément se retrouve à la fin de la liste : c'est-à-dire à sa place sans l'ordre croissant.
- **Étape 2 :** on recommence avec les $n-1$ premiers éléments de la liste :
  - ▶ on compare `L[0]` avec `L[1]` et on les échange s'ils ne sont pas dans le bon ordre ;
  - ▶ on compare `L[1]` avec `L[2]` et on les échange s'ils ne sont pas dans le bon ordre ;
  - ▶ on continue ainsi jusqu'à arriver à `L[n-3]` et `L[n-2]`.

  À l'issue de cette deuxième étape, le plus grand élément figurant parmi les $n-1$ premiers se retrouve au rang $n-1$ et ainsi les deux derniers éléments de la liste sont bien rangés.
- On recommence ainsi jusqu'à l'étape $n-1$, à laquelle les $n-2$ derniers éléments de la liste seront bien placés et il ne restera plus qu'à comparer `L[0]` avec `L[1]` et à les échanger s'ils ne sont pas dans le bon ordre : la liste sera alors triée.

Ainsi, le principe est de faire « remonter » les éléments les plus grands en fin de liste au fur et à mesure, comme les plus grosses bulles d'air remontent en premier dans un verre d'eau, ce qui explique le nom de la méthode.

Appliquons l'algorithme de tri à bulles à l'exemple de la liste `[6, 2, 8, 5, 4, 1]`.

Étape 1 :

⑥	②	8	5	4	1	on échange 6 et 2
2	⑥	⑧	5	4	1	on laisse 6 et 8 en place
2	6	⑧	⑤	4	1	on échange 8 et 5
2	6	5	⑧	④	1	on échange 8 et 4
2	6	5	4	⑧	①	on échange 8 et 1
2	6	5	4	1	8	8 est à présent bien placé

Étape 2 :

②	⑥	5	4	1	8	on laisse 2 et 6 en place
2	⑥	⑤	4	1	8	on échange 6 et 5
2	5	⑥	④	1	8	on échange 6 et 4
2	5	4	⑥	①	8	on échange 6 et 1
2	5	4	1	6	8	6 et 8 sont bien placés

Étape 3 :

②	⑤	4	1	6	8	on laisse 2 et 5 en place
2	⑤	④	1	6	8	on échange 5 et 4
2	4	⑤	①	6	8	on échange 5 et 1
2	4	1	5	6	8	5, 6 et 8 sont bien placés

Étape 4 :

②	④	1	5	6	8	on laisse 2 et 4 en place
2	④	①	5	6	8	on échange 4 et 1
2	1	4	5	6	8	4, 5, 6 et 8 sont bien placés

Étape 5 :

②	①	4	5	6	8	on échange 2 et 1
1	2	4	5	6	8	la liste est triée

Question 12. En utilisant deux boucles imbriquées, écrire une fonction `tri_a_bulles` qui prend en argument une liste de nombres `L` et qui renvoie la liste triée par ordre croissant obtenue par la méthode du tri à bulles. Vérifier ensuite avec la liste `L` de la **Question 1**.

2) Comparaison des complexités temporelles

Question 13. Écrire une fonction `liste_aleatoire` qui prend en arguments deux entiers naturels non nuls `n` et `p` et qui renvoie une liste de longueur `n` constituée d'entiers choisis aléatoirement et de façon équiprobable entre 0 et p.

Question 14. En utilisant la fonction `time` du module `time` (voir TP5), écrire une fonction `temps` qui prend en argument une fonction de tri `fonc_tri` et trois entiers naturels non nuls `n`, `p` et `nb` et qui renvoie le temps moyen mis par le fonction `fonc_tri` pour trier une liste renvoyée par `liste_aleatoire(n,p)` lors de `nb` répétitions.

Question 15. En s'inspirant de ce qui a été fait dans le TP4, vérifier expérimentalement que la complexité temporelle du tri par insertion, du tri par sélection et du tri à bulles sont quadratiques i.e., pour n assez grand, approximativement proportionnelles à n^2 où n est la longueur de la liste.

Question 16. Vérifier expérimentalement que, si les éléments d'une liste de longueur n sont des entiers compris entre 0 et n alors la complexité temporelle du tri par comptage est linéaire i.e., pour n assez grand, approximativement proportionnelle à n.

3) Intérêt du tri pour la recherche d'éléments

a) Recherche séquentielle

Lorsqu'on cherche à savoir si un nombre fait partie d'une liste, on peut tester successivement, les uns après les autres, les éléments de la liste jusqu'à en trouver un qui est égal à x et, sinon, cela signifie que x n'est pas un élément de la liste. C'est ce qu'on appelle la recherche séquentielle.

Question 17. Écrire une fonction `recherche_sequentielle` qui prend en arguments une liste `L` et un élément `x` et qui renvoie, à l'aide d'une recherche séquentielle, l'indice de la première occurrence de `x` dans `L` si `x` est un élément de `L` et `-1` sinon

b) Recherche dichotomique

Lorsqu'on n'a aucune information particulière sur la liste `L`, on ne peut faire autrement que procéder par recherche séquentielle pour déterminer si un nombre `x` appartient à `L`.

En revanche, si la liste `L` est triée par ordre croissant, on peut utiliser une autre méthode : la recherche dichotomique.

Le principe est le suivant. On considère l'élément `m` situé au milieu de la liste. Si `m` est égal à `x` alors on renvoie son indice et la fonction se termine. Si `m` est strictement inférieur à `x` alors, comme la liste est triée, tous les éléments précédents dans la liste sont aussi strictement inférieurs à `x`. On va donc chercher `x` parmi les éléments situés après `m` dans la liste. Enfin, si `m` est strictement supérieur à `x` alors, comme la liste est triée, tous les éléments suivants dans la liste sont aussi strictement supérieurs à `x`. On va donc chercher `x` parmi les éléments situés avant `m` dans la liste.

Plus précisément, pour la programmation de la recherche dichotomique, on peut procéder de la manière suivante.

- On initialise deux variables `debut` et `fin` respectivement à `0` et `len(L)`.
- Tant qu'il reste des éléments entre `L[debut]` et `L[fin-1]` i.e. tant que `debut` est strictement inférieur à `fin` :
  - ▶ on affecte à la variable `milieu` le plus grand entier inférieur ou égal à la moyenne de `debut` et de `fin`.
  - ▶ si `L[milieu]` est égal à `x`, on renvoie l'indice `milieu` et la fonction s'arrête.
  - ▶ si `L[milieu]` est strictement inférieur à `x`, on affecte à `debut` la valeur `milieu + 1`.
  - ▶ si `L[milieu]` est strictement supérieur à `x`, on affecte à `fin` la valeur `milieu - 1`.
- Si la fonction ne s'est pas arrêtée avant, on renvoie `-1`.

Question 18. Écrire une fonction `recherche_dichotomique` qui prend en argument une liste de nombres triée `L` et un nombre `x` et qui utilise la méthode de recherche dichotomique pour renvoyer un indice `k` tel que `L[k]` soit égal à `x` si `x` est une élément de `L` et `-1` sinon.

c) Comparaison des complexités

Question 19. En s'inspirant de ce qui a été fait en **V. 2)**, vérifier expérimentalement que la complexité temporelle de la recherche séquentielle est linéaire i.e. que pour une liste de longueur n suffisamment grande, le temps moyen de recherche d'un élément est approximativement proportionnel à n alors que la complexité temporelle de la recherche dichotomique est logarithmique i.e. que pour une liste de longueur n suffisamment grande, le temps moyen de recherche d'un élément est approximativement proportionnel à $\ln(n)$.