**Lycée Lavoisier** **Année 2024-2025**
**PC – Informatique**

**TP : Tris de listes**

Dans le cours de première année, vous avez rencontré trois algorithmes de tris pour les listes : le tri à bulles, le tri fusion et le tri rapide.

Le but de ce TP est de présenter deux autres algorithmes de tri : le tri par sélection et le tri par insertion. Puis de comparer ces différents algorithmes de tri.

Afin de tester les fonctions, vous pouvez utiliser le code de la fonction ci-dessous qui crée une liste d'entiers aléatoire de taille $n$.

# 1 En guise d'introduction

**Exercice 1.**

1. Afin de tester les fonctions codées dans la suite, taper le code de la fonction ci-dessous.

```
import numpy as np
def liste_alea(n):
        return [int(numpy.random.rand()*1000) for i in range(n)]
```

   Vérifier ce qu'elle renvoie sur quelques exemples.

2. Ecrire le code d'une fonction `triee(t)` qui prend un tableau `t` (d'entiers ou de flottants) en entrée et renvoie un booléen égal à `True` si la liste est triée et à `False` sinon.

   On pourra se servir de cette fonction pour tester la validité des différentes fonctions de tris implémentées : on pourra vérifier facilement, après exécution des fonctions de tris, que les listes sont maintenant triées.

# 2 Tri par insertion et tri par sélection

## 2.1 Tri par insertion

Le tri par insertion est l'un des plus intuitifs qui soit : on trie des tranches croissantes du tableau en insérant chaque nouvel élément à sa bonne place dans la tranche déjà triée.

**Exercice 2.** Ecrire le code d'une fonction Python `insertion_elt(t,j)` qui prend en argument un tableau `t` et un entier `j` (que l'on suppose compris entre 0 et `len(t)-1`). On suppose que la tranche `t[:j]` est triée. La fonction `insertion_elt(t,j)` ne renvoie rien mais modifie le tableau de manière à insérer `t[j]` à la bonne place dans la tranche `t[:j+1]`. Ainsi, à l'issue de l'exécution de la fonction, la tranche `t[:j+1]` est triée.

**Exercice 3.**

1. A l'aide de la fonction `insertion_elt(t,j)`, écrire le code du fonction `tri_insertion(t)` qui prend un tableau `t` en argument et le trie sans rien renvoyer.
2. Estimer la complexité de la fonction `tri_insertion()` dans le pire des cas et dans le meilleur des cas (la complexité en moyenne est la même complexité que dans le pire des cas).

### 2.2 Tri par sélection

Le principe du tri par sélection est le suivant : on parcourt des tranches décroissantes du tableau ; dans chaque tranche, on cherche le minimum et on le met à la bonne position dans la tranche correspondante. Quand on arrive à une tranche avec un seul élément, le tableau est trié.

**Exercice 4.** Ecrire le code d'une fonction Python `pos_min_tranche(t,j)` qui prend en argument un tableau `t` et un entier `j` (que l'on suppose compris entre 0 et `len(t)-1`) et qui renvoie le minimum de la tranche `t[j::len(t)]`.

**Exercice 5.**

1. Ecrire le code d'une fonction `tri_selection(t)` qui implémente l'algorithme du tri par sélection. La fonction prend en argument un tableau `t` et le trie sans rien renvoyer.
   On utilisera la fonction `pos_min_tranche(t,j)`
2. Estimer la complexité de la fonction `tri_selection(t)` dans le pire des cas et dans le meilleur des cas.

## 3 Rappel : le tri à bulles

Le principe du tri à bulles est de comparer de manière répétée les éléments consécutifs d'un tableau et de les permuter lorsqu'ils sont mal triés. On arrête dès que le tableau est trié.

**Exercice 6.** Ecrire le code d'un fonction `echanges(t)` qui prend en argument un tableau `t`. La fonction va parcourir la liste et comparer tous les éléments consécutifs. Elle échange les valeurs de `t[i]` et de `t[i+1]` si `t[i]>t[i+1]`. De plus la fonction renvoie `True` si elle a effectué au moins un échange de valeurs et `False` sinon.

**Exercice 7.**

1. Ecrire le code d'une fonction `tri_bulles(t)` qui prend en argument un tableau `t` et le trie suivant la méthode du tir à bulles.
   On utilisera la fonction `echanges(t)`.
2. Estimer la complexité de la fonction `tri_bulles(t)` dans le pire des cas et dans le meilleur des cas.

## 4 Comparaison des différents tris

On rappelle la commande qui permet de calculer le temps mis par un calcul. Cela repose sur l'utilisation de la bibliothèque `time` de Python.

```
import time
tic=time.time()
# code du calcul a effectuer
tac=time.time()
temps=tac-tic # le temps mis est donne en secondes
```

**Exercice 8.**

1. Pour $n$ fixé, créer $N$ listes au hasard. Calculer le temps moyen de calcul pour trier ces listes avec chacune des trois fonctions de tris implémentées précédemment.
2. Tracer sur un même graphique le temps moyen nécessaire pour trier une liste en fonction de de la taille des listes pour chacune des trois fonctions. On pourra

**Exercice 9.**

1. Ecrire le code de trois fonctions `tri_insertion2()`, `tri_selection2()` et `tri_bulles2()` fusionnant en une seule fonction chacune des fonctions de tris précédentes et la fonction auxiliaire qui l'accompagne.

   Insérer également au bon endroit les lignes de code suivantes, permettant d'afficher la liste à chaque modification (on suppose que `matplotlib.pyplot` est déjà chargée) :

```
plt.clf() #Efface la figure
plt.bar([i for i in range(len(t))],t) #Affiche la liste sur un graphique
plt.pause(0.2) #Met une pause de 0.2 secondes
```

2. Tester les fonctions précédentes sur quelques exemples, observer les résultats et expliquer le pourquoi du nom de chaque méthode de tri.

## 5 Si vous avez fini : autour de la suite de Syracuse

La suite de Syracuse estt définie de la manière suivante. On fixe un entier $n \geqslant 2$. Si $n$ est pair, on le remplace par $n/2$ ; s'il est impair, on le remplace par $3n+1$. On poursuit le processus jusqu'à tomber sur 1. Par exemple, en partant de $n = 42$, on obtient la suite :

$$42 \to 21 \to 64 \to 32 \to 16 \to 8 \to 4 \to 2 \to 1$$

On obtient ainsi une chaîne de longueur 9.

On conjecture que pour $n \geqslant 2$ quelconque, on finit toujours par retomber sur 1 en un nombre fini d'étapes et que l'on obtient donc une chaîne de longueur finie. Ce résultat a été vérifié pour de nombreuses valeurs de $n$ mais n'a toujours pas été prouvé en toute généralité – il est donc peut-être faux...

**Exercice 10.** Ecrire le code d'une fonction `longueur_max()` qui prend en entrée un entier $N$ et renvoie l'entier $n \in [\![2, N]\!]$ pour lequel la longueur de la chaîne menant à 1 est maximum – on renverra l'entier $n$ et la longueur de la chaîne correspondante.

On veut un code qui calcule le résultat pour $n = 10^6$ en 5 secondes.