Introduction :

Dans ce compte-rendu nous allons voir deux types de tris de données, essentiels en informatique :

- Tri par insertion
- Tri par sélection

1) Tri par insertion :

Algorithme :

```
def tri_insertion(tabl):
 for step in range(1, len(tabl)):
 # Je parcours les données pour chaque nombre présent dans la liste du rang 1 à la longueur maximale de la liste.

  key = tabl[step]    # Je retiens temporairement le nombre qui se trouve dans la liste, à l'index de l'étape actuelle.
  j = step - 1          # Je retiens temporairement dans une variable j l'index précédent au nombre étudié actuellement.

  while j >= 0 and key < tabl[j] :
   # Tant que ma variable j est supérieur ou égale à 0 (qui représente donc l'index minimal de ma liste) et que ma valeur étudié est inférieur à la valeur qui se trouve juste avant dans la liste.
   tabl[j+1] = tabl[j]        # Alors je place l'élément inférieur au rang précédent.
   j = j - 1

  tabl[j+1] = key             # Sinon je place la valeur temporaire après la valeur qui est plus petite qu'elle.

 return tabl
```

Exécution :

```
tabl_a_traiter = [7, 5, 13, 8, 9]
print(tri_insertion(tabl_a_traiter))
```

Résultat dans la console :

```
[5, 7, 8, 9, 13]
```

2) Tri par sélection :

Algorithme :

```
def tri_selection(tabl):
 for step in range(len(tabl)):
  min_index = step      # Valeur actuellement sélectionnée, on cherche donc une valeur inférieur dans toute la liste.

  for j in range(step + 1, len(tabl)):       # Je parcours l'entièreté de la liste

   if tabl[j] < tabl[min_index]:             # Si je trouve une valeur inférieur à celle sélectionné :
    min_index = j                            # Alors je retiens son index, afin qu'il soit le nouveau minimum.

  (tabl[step], tabl[min_index]) = (tabl[min_index], tabl[step])       # Inversion des deux valeurs sélectionnés.
  return tabl
```

Exécution :

```
tabl_a_traiter = [7, 5, 13, 8, 9]
print(tri_selection(tabl_a_traiter))
```

Résultat dans la console :

```
[5, 7, 8, 9, 13]
```

Conclusion :

On peut donc en conclure que bien que nous ayons utilisés deux méthodes de tris différentes : le résultat est le même, à noter que la complexité d'exécution des deux méthodes est elle aussi identique, en effet il s'agit d'une **complexité quadratique**.

3) Mesurer les temps d'exécution des fonctions de tris :

Dans cette partie, nous allons pouvoir comparer la complexité des fonctions de tris en prenant pour paramètre le temps d'exécution.

Plusieurs comparaisons seront possibles :

- Entre tri par sélection & tri par insertion
- Entre les différentes tailles de liste à traiter (100 valeurs, 1 000 valeurs, 5 000 valeurs, 10 000 valeurs, 50 000 valeurs, 100 000 valeurs)

```python
import random
import time


def tri_insertion(tabl):
    start_time = time.perf_counter()
    for step in range(1, len(tabl)):
        # Je parcours les données pour chaque nombre présent dans la liste du rang 1
        # à la longueur maximale de la liste.

        key = tabl[step]     # Je retiens temporairement le nombre qui se trouve
                             # dans la liste, à l'index de l'étape actuelle.
        j = step - 1         # Je retiens temporairement dans une variable j l'index précédent
                             # au nombre étudié actuellement.

        while j >= 0 and key < tabl[j]:
            # Tant que ma variable j est supérieur ou égale à 0 (qui représente donc
            # l'index minimal de ma liste) et que ma valeur étudié est inférieur à la valeur
            # qui se trouve juste avant dans la liste.
            tabl[j + 1] = tabl[j]  # Alors je place l'élément inférieur au rang précédent.
            j = j - 1

        tabl[j + 1] = key    # Sinon je place la valeur temporaire après la valeur
                             # qui est plus petite qu'elle.
    stop_time = time.perf_counter()
    exec_time = round((stop_time - start_time)*1000, 3)
    return exec time


def tri_selection(tabl):
    start_time = time.perf_counter()
    for step in range(len(tabl)):
        min index = step     # Valeur actuellement sélectionnée,
                             # on cherche donc une valeur inférieur dans toute la liste.

        for j in range(step + 1, len(tabl)):  # Je parcours l'entièreté de la liste

            if tabl[j] < tabl[min_index]:  # Si je trouve une valeur inférieur à celle sélectionné
                min_index = j  # Je retiens son index, afin qu'il soit le nouveau minimum.

        (tabl[step], tabl[min_index]) = (tabl[min_index], tabl[step])  # Inversion des deux
valeurs sélectionnés
    stop_time = time.perf_counter()
    exec_time = round((stop_time - start_time)*1000, 3)
    return exec_time


min = 0
max = 1000000



nb_elements = 100
tabl_a_traiter = [random.randint(min, max) for iter in range(nb_elements)]

print("\n",'Tri par insertion de',nb_elements,'valeurs aléatoires. Traité en :',
tri_insertion(tabl_a_traiter), "milliseconde(s)", "\n")
print('Tri par sélection de',nb_elements,'valeurs aléatoires. Traité en :',
tri_selection(tabl_a_traiter), "milliseconde(s)", "\n")

nb_elements = 1000
tabl_a_traiter = [random.randint(min, max) for iter in range(nb_elements)]

print("\n",'Tri par insertion de',nb_elements,'valeurs aléatoires. Traité en :',
tri_insertion(tabl_a_traiter), "milliseconde(s)", "\n")
print('Tri par sélection de',nb_elements,'valeurs aléatoires. Traité en :',
tri_selection(tabl_a_traiter), "milliseconde(s)", "\n")

nb_elements = 5000
tabl_a_traiter = [random.randint(min, max) for iter in range(nb_elements)]

print("\n",'Tri par insertion de',nb_elements,'valeurs aléatoires. Traité en :',
tri_insertion(tabl_a_traiter), "milliseconde(s)", "\n")
print('Tri par sélection de',nb_elements,'valeurs aléatoires. Traité en :',
tri_selection(tabl_a_traiter), "milliseconde(s)", "\n")

```

```
70 nb_elements = 10000
71 tabl_a_traiter = [random.randint(min, max) for iter in range(nb_elements)]
72
73 print("\n",'Tri par insertion de',nb_elements,'valeurs aléatoires. Traité en :',
tri_insertion(tabl_a_traiter), "milliseconde(s)", "\n")
74 print('Tri par sélection de',nb_elements,'valeurs aléatoires. Traité en :',
tri_selection(tabl_a_traiter), "milliseconde(s)", "\n")
75
76 nb_elements = 50000
77 tabl_a_traiter = [random.randint(min, max) for iter in range(nb_elements)]
78
79 print("\n",'Tri par insertion de',nb_elements,'valeurs aléatoires. Traité en :',
tri_insertion(tabl_a_traiter), "milliseconde(s)", "\n")
80 print('Tri par sélection de',nb_elements,'valeurs aléatoires. Traité en :',
tri_selection(tabl_a_traiter), "milliseconde(s)", "\n")
81
82 nb_elements = 100000
83 tabl_a_traiter = [random.randint(min, max) for iter in range(nb_elements)]
84
85 print("\n",'Tri par insertion de',nb_elements,'valeurs aléatoires. Traité en :',
tri_insertion(tabl_a_traiter), "milliseconde(s)", "\n")
86 print('Tri par sélection de',nb_elements,'valeurs aléatoires. Traité en :',
tri_selection(tabl_a_traiter), "milliseconde(s)", "\n")
```

La méthode « time.perf_counter() » permet de mesurer un time code au début de la fonction ainsi qu'à la fin, pour ensuite faire la différence entre les deux mesures.

Exécution :

Tri par insertion de 100 valeurs aléatoires. Traité en : 0.222 milliseconde(s)

Tri par sélection de 100 valeurs aléatoires. Traité en : 0.192 milliseconde(s)

Tri par insertion de 1000 valeurs aléatoires. Traité en : 23.104 milliseconde(s)

Tri par sélection de 1000 valeurs aléatoires. Traité en : 21.382 milliseconde(s)

Tri par insertion de 5000 valeurs aléatoires. Traité en : 568.946 milliseconde(s)

Tri par sélection de 5000 valeurs aléatoires. Traité en : 516.116 milliseconde(s)

Tri par insertion de 10000 valeurs aléatoires. Traité en : 2241.141 milliseconde(s)

Tri par sélection de 10000 valeurs aléatoires. Traité en : 2049.823 milliseconde(s)

Tri par insertion de 50000 valeurs aléatoires. Traité en : 57264.707 milliseconde(s)

Tri par sélection de 50000 valeurs aléatoires. Traité en : 58429.96 milliseconde(s)

Tri par insertion de 100000 valeurs aléatoires. Traité en : 241932.506 milliseconde(s)

Tri par sélection de 100000 valeurs aléatoires. Traité en : 247136.72 milliseconde(s)

Premièrement, il faut savoir que les listes sont générées de manière **totalement aléatoire** avec des valeurs entières positives comprises entre **0** et **1 000 000**. On utilise pour cela la librairie ***random***.

Comme on pouvait le prévoir, le temps d'exécution des deux types de tri pour une même liste est très semblable, étant donné qu'il s'agit dans les deux cas d'**une même complexité quadratique.**

A noter qu'ici le temps d'exécution de la fonction ne comptabilise pas l'affichage de la liste triée afin d'éviter tout ralentissements superflus.

4) Comparaison avec la méthode native « sorted() »:

Je rajoute une troisième méthode de tri, qui sera la méthode « sorted » et qui est intégrée nativement avec Python. Nous allons donc voir si cela nous permet de trier notre liste de manière **encore plus rapide**.

```
1 import time
2
3 def tri_via_sort(tabl):
4     start_time = time.perf_counter()
5     tabl.sort()
6     stop_time = time.perf_counter()
7     exec_time = round((stop_time - start_time)*1000, 3)
8     return exec_time
```

Je vais donc traiter des tableaux aléatoires de différentes tailles pour mesurer le temps d'exécution de chaque fonction !

	100 valeurs	1 000 valeurs	5 000 valeurs	10 000 valeurs	50 000 valeurs	100 000 valeurs
Tri par insertion	0.222 ms	23.104 ms	568.946 ms	2241.141 ms	57264.707 ms	241932.506 ms
Tri par sélection	0.192 ms	21.382 ms	516.116 ms	2049.823 ms	58429.96 ms	247136.72 ms
Tri par fonction native	0.003 ms	0.01 ms	0.048 ms	0.054 ms	0.58 ms	2.233 ms

Comme on peut le voir, étant donné qu'il s'agit d'une complexité quadratique si l'on prend par exemple : **100^2 = 10 000**, dans ce cas on multiplie bien le nombre de valeurs pour trouver le temps d'exécution par rapport à la racine, soit **0,222 ms pour 100 valeurs qui multiplié par 10 000 donne bel et bien environ 2 220 millisecondes** !

On peut aussi remarquer que la fonction native a un temps d'exécution **extrêmement faible** par rapport aux fonctions que j'ai moi-même mises en place. Cela peut s'expliquer par l'**optimisation de la fonction** directement dans le cœur du langage de programmation.