1) Vérification

Afin de vérifier les algorithmes nous avons créé un algorithme permettant de créer un tableau non trié avec des valeurs choisit aléatoirement et de renvoyer leurs valeurs pour ensuite appeler la fonction et tester grâce à une fonction si le tableau est trié.

Fonction de test :

```python
def testTri(tab):
    # Test pour savoir si le tableau est trié
    for i in range(1, tab.size):
        v = i - 1
        if tab[v] <= tab[i]:
            v += 1
        else:
            return False
    return True
```

Algo_X

Test avec des valeurs positives

```
Tableau non trié
False
Tableau trié
True
```

Test avec des valeurs négative

```
Tableau non trié
False
Tableau trié
True
```

Tes avec des grandes valeurs

```
Tableau non trié
False
Tableau trié
True
```

Test avec un tableau long

```
Tableau non trié
False
Tableau trié
True
```

On peut voir que l'algo_x trie correctement les tableau.

Algo_Y

Test avec des valeurs positives

```
Tableau non trié
False
Tableau trié
True
```

Test avec des valeurs négatives

```
Tableau non trié
False
Tableau trié
True
```

Test avec de grandes valeurs

```
Tableau non trié
False
Tableau trié
True
```

Test avec beaucoup de valeurs

```
Tableau non trié
False
Tableau trié
True
```

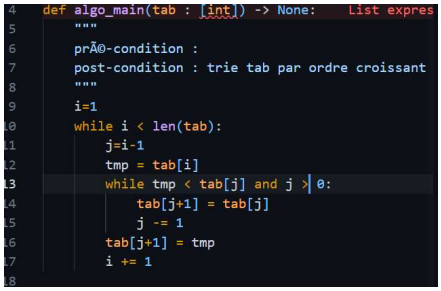
On peut voir que l'algo_Y trie correctement le tableau

Algo_Z

```
Tableau non trié
False
Tableau trié
False
```

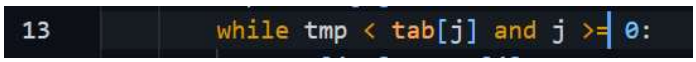
L'algo Z ne passe pas le premier test. On peut constater que tous les éléments sont triés sauf le premier.

Analysons l'algorithme



On peut voir ici que ligne 13 j doit être strictement supérieur à 0, et donc il ne traite jamais le 1ere élément.

Pour corriger cela il faut que j soit supérieur ou égal à 0



Retestons l'algorithme.

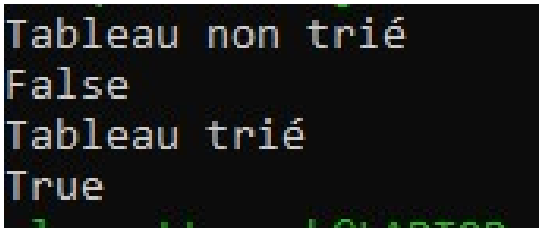
```
claquetteuuuh@LAPTOP-A0JERQ2D:/mnt/c/Users/Thomas/Desktop/IUT/SAE/algo$ python3.9 algo_z.py
Tableau non trié
2.0, 80.0, 44.0, 69.0, 1.0, 97.0, 62.0, 54.0, 91.0, 49.0,

Tableau triéI
1.0, 2.0, 44.0, 49.0, 54.0, 62.0, 69.0, 80.0, 91.0, 97.0,
```

Maintenant l'algorithme fonctionne, continuons les test.

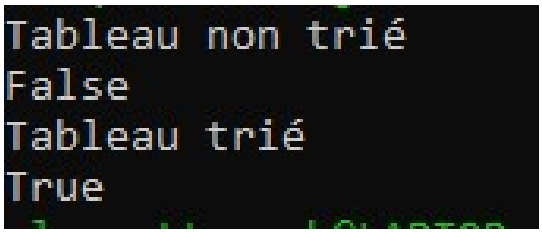
Test avec des valeurs négatives :

```
Tableau non trié
False
Tableau trié
True
```

Test avec de grandes valeurs :



Test avec beaucoup de valeur



2) Description du fonctionnement des algos

Algo_x :

Cet algo est le tri rapide ou quicksort, il consiste à prendre le dernière élément de la liste et le placer au centre en tant que pivot, le but étant ensuite de séparer la liste en 2 liste, les éléments plus petit que le pivot d'un côté et les élément les plus grand de l'autre. L'opération est répété jusqu'à ce que tout les éléments soient triés.

C'est un algorithme en place, qui a une complexité quadratique dans le pire des cas et en $\Theta(n \log n)$ dans le meilleur des cas (celui ou les 2 sous listes sont de même taille). L'instance d'entré pour laquelle le quicksort est le moins efficace est le tri d'une liste déjà trié, car les sous listes n'auront qu'un seul élément en moins.

Algo_y :

Cet algo est le tri à bulle, son principe est de faire remonter le plus gros élément de la liste à la fin et faire cela pour tous les éléments jusqu'à ce que le tableau soit trié.

Il a dans le meilleur des cas une complexité de $\Theta(n)$ c'est-à-dire dans le cas ou il tri un tableau déjà trié. L'instance d'entré la moins efficace pour le tri à bulle est un tableau avec des éléments trié dans le sens inverse, sa complexité ici est de $\Theta(n^2)$

Sa complexité spatiale est de 5 car il :

- Crée la fonction algo_main soit 1 case
- Crée la variable i soit 1 case
- Crée la variable condition soit 1 case
- Crée la variable j soit 1 case
- Crée la variable tmp soit 1 case

Algo_z :

Cet algo est le tri par insertion, le principe de celui-ci est de prendre un élément et l'insérer à la bonne place en fonction des éléments précédent.

Dans le meilleur des cas, celui où les données d'entrée sont déjà triée, ce tri à une complexité de $\Theta(n)$, tout comme le tri à bulle sa bête noir sont les tableaux trié à l'envers ce qui lui donne dans ce cas là une complexité de $\Theta(n^2)$.

Il a une complexité spatiale de 4 car il crée :

- La fonction algo_main soit 1 case
- La variable i soit 1 case
- La variable j soit 1 case
- La variable tmp soit 1 case

3) Comparaison des algos

#	Très petit tableau - taille 10	Petit tableau - taille 100	Tableau moyen - taille 1000	Grand tableau - taille 10000	Tableau déjà trié - taille 10000
Algo_x	0	0.002023458	0.025102615	0.284471273	0.294255018
Algo_y	0	0.005215406	0.633231163	61.69444847	49.93724871
algo_z	0	0.002211571	0.185587645	18.52155948	0.010456562

Pour comparer les algos, nous avons calculé les temps d'exécution grâce à la fonction time.time() puis nous avons effectué le tableau ci-dessus qui montre le temps d'exécution en fonction des tableaux à trier.

Pour un tableau de très petite taille, les temps d'exécution sont presque égaux, on peut donc choisir n'importe quel algorithme.

Pour un tableau de petite taille, les temps d'exécution sont similaire bien que le quicksort s'en sort mieux que les autres.

Pour un tableau de taille moyenne, la différence du quicksort est flagrante, il est donc à privilégier, de plus on commence déjà à constater la faiblesse du tri à bulle qui met plus d'une demi seconde à trier ce tableau.

Pour un grand tableau, le quicksort est 200 fois meilleur que le tri à bulle (qui reste le tri le moins efficace) et 60 fois meilleur que le tri par insertion.

Dans le cas d'un tableau déjà trié, le quicksort s'en sort toujours très bien, il reste 160 fois meilleur que le tri à bulle, cependant il est moins efficace que le tri par insertion qui lui est 29 fois meilleur que le quicksort.

Conclusion :

Pour la majorité des cas, il est préférable d'utiliser le quicksort, cependant si le tableau est déjà trié, le tri par insertion est meilleur. Il faut éviter d'utiliser le tri à bulle qui est systématiquement moins performant que les 2 autres.

4) Ecrire

(Voir algo_p.py) Cet algorithme le tri par sélection, ce tri consiste à trouver le minimum et à le placer au début de la partie non trié du tableau. Ce tri à une complexité qui ne change pas selon la liste, elle est de $\Theta(n^2)$. Ce tri a une complexité spatiale de 5 car il crée 4 variable et 1 fonction.

#	Très petit tableau - taille 10	Petit tableau - taille 100	Tableau moyen - taille 1000	Grand tableau - taille 10000	Tableau déjà trié - taille 10000
Algo_x	0	0.002023458	0.025102615	0.284471273	0.294255018
Algo_y	0	0.005215406	0.633231163	61.69444847	49.93724871
Algo_z	0	0.002211571	0.185587645	18.52155948	0.010456562
Algo_a	0	0.001998186	0.274695396	26.02384543	24.69933748

Comme on peut le voir, ce tri est toujours meilleur que le tri à bulles, mais n'a pas les avantages des 2 autres tris, l'efficacité du tri rapide dans le cas de tableaux longs et l'efficacité du tri d'une liste déjà triée du tri par insertion.

J'ai contribué à ce projet en faisant la description de l'algorithme x, l'écriture de l'algorithme p. Puis j'ai fait aussi son analyse et sa comparaison avec les autres algorithmes.