	NUMÉRIQUE ET SCIENCES INFORMATIQUES	ALGORITHMIQUE	TP : Tris par Insertion et par Sélection	1ere

1 EFFICACITÉ

a. Construction (A l'aide de pyzo)

Dans un programme **Tris.py**. Ecrire les programmes suivant.

Créez un tableau **t1**, qui est composé des entiers de 0 à 9 :

Créez un tableau **t2**, qui est composé des entiers de 99 à 0 :

Grâce à la bibliothèque **random** et à la fonction **randint(valeur_min, valeur_max)**, créez un tableau **t3**, qui est composé de 1000 entiers choisis aléatoirement entre 0 et 500:

b. Calcul du temps

Récupérez maintenant les algorithmes tri par sélection et tri par insertion vu en cours et écrivez les dans le programme **Tris.py**. Importez la bibliothèque time (**from time import ***). Nous allons utiliser cette bibliothèque afin de comparer l'efficacité des algorithmes de tris dans différentes situations.

Tout d'abord, dans vos fonctions de tris, ajoutez le code suivant à la première ligne :

```
timeStart = time()
```

puis à la fin de vos fonctions, avant le **return** final, ajoutez les deux lignes suivantes :

```
timeEnd = time()
print( timeEnd  - timeStart)
```

L'**epoch unix** est une date de référence (1er janvier 1970 à 00:00:00.). La fonction **time()** mesure le "**timestamp**" ou temps en secondes depuis cette référence. Au lancement d'une de vos deux fonctions de tri, le temps depuis l'**epoch** est sauvegardé dans la variable **timeStart**, puis lorsque le tri est terminé, le temps depuis l'**epoch** est sauvegardé dans **timeEnd**. Il suffit donc de soustraire la variable **timeStart** à la variable **timeEnd** afin de connaître le temps écoulé**,** c'est -à -dire le temps d'exécution de la fonction.

Quel est selon vous "le pire des cas" pour un algorithme de tri :

_____

Quel est selon vous "le meilleur des cas" pour un algorithme de tri :

_____

c. Comparatif

Nous allons enfin pouvoir comparer le temps d'exécution de nos deux algorithmes dans "le pire des cas", "le meilleur des cas" et "aléatoirement".
Pour cela, lancez la fonction **tri_selection()** avec en paramètre un tableau trié par ordre croissant ("meilleur des cas") de 1000 éléments (modifier pour cela le programme qui donne **t1**) , puis de 2000, 4000, 8000 et 16000 éléments. Répertoriez les résultats dans le tableau suivant avec **2 décimales après la virgule**. Faites de même pour le "pire des cas" (programme qui donne **t2**), et "aléatoirement" (programme qui donne **t3**) puis recommencez avec la fonction **tri_insertion()**.

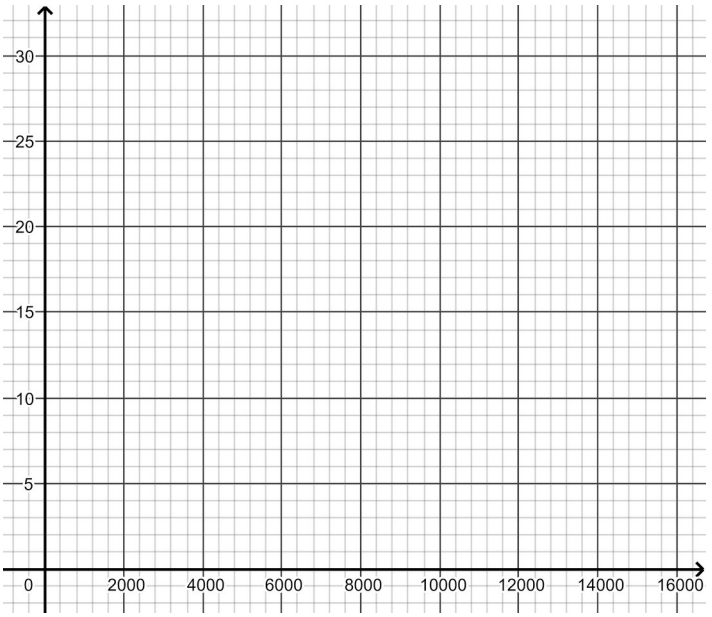
Exemple :

```
t1 = [x for x in range(0,1000)]

tri_selection( t1 )
```

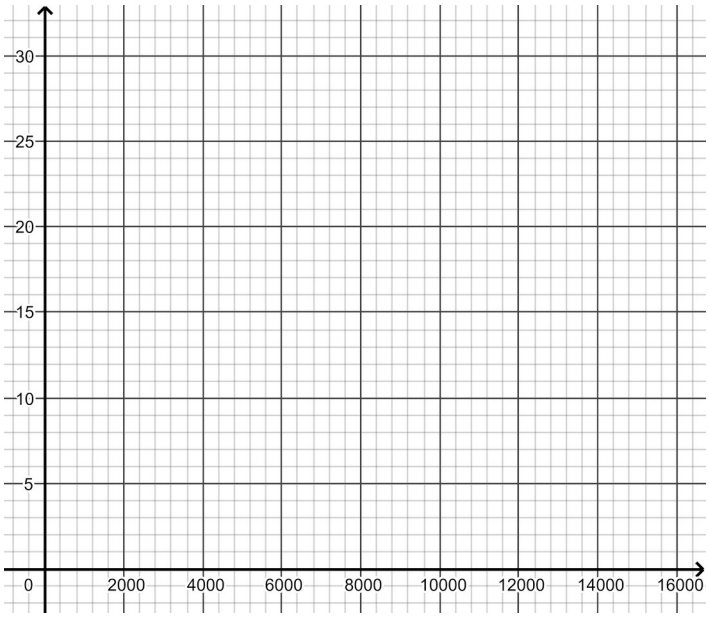
	“Meilleur des cas”					“Aléatoire”					“Pire des cas”				
	1000	2000	4000	8000	16000	1000	2000	4000	8000	16000	1000	2000	4000	8000	16000
Tri par sélection															
Tri par insertion															

Meilleur cas Aléatoire Pire des cas



Tri par sélection

Meilleur cas Aléatoire Pire des cas



Tri par insertion

Que peut-on en conclure ?

2 EXERCICES D'ENTRAÎNEMENT

a. Minimum et maximum

Écrire une fonction **valeur_min(tab)** qui renvoie la valeur minimale d’un tableau d’entiers **tab** déjà trié et une fonction **valeur_max(tab)** qui renvoie la valeur maximale d’un tableau d’entiers **tab** déjà trié. Les fonctions **min()** et **max()** sont bien sûr inutiles !

b. Ordre décroissant

Créer deux fonctions: **tri_selection_decroissant(tab)** et **tri_insertion_decroissant(tab)** qui reprennent respectivement les deux tris vues en cours mais renvoient un tableau trié par ordre décroissant. Vous devez utiliser les algorithmes vu en classe et pas la fonction **sorted()**, ni une autre méthode.

c. Sans doublons

Écrire une fonction **sans_doublon(tab)** qui vérifie si tous les éléments du tableau d'entiers **tab** sont différents. La fonction renverra un booléen.

d. Valeur fréquente

Écrire une fonction **valeur_plus_frequente(tab)** qui prend en argument un tableau d'entiers **tab** qui renvoie la valeur la plus fréquente dans ce tableau.

Aide

Commencez par trier le tableau.
Puis créer plusieurs variables :

- **current_compt** : un compteur courant, qui servira à compter le nombre d'occurrence de l'élément courant
- **current_val** : une variable qui stockera la valeur de l’élément courant
- **max_compt** : un compteur max, qui stockera le plus grand nombre d'occurrences qu’on ait rencontré
- **max_val** : une variable qui stockera la valeur de l'élément qui a le plus d'occurrences jusque là