TP – CHAPITRE 9 - TRI

Sommaire

TP1 - Exemple de calcul de complexité en itératif

- Sur votre compte créer un dossier **TP-Chapitre9** dans lequel on va enregistrer les TP de ce chapitre. Récupérer le fichier **TP1 .py** et l'enregistrer dans le dossier **TP-Chapitre9**. La fonction **somme(n)** calcule la somme : $1 + 2 + 3 + \ldots + n - 1$.
- Ajouter des commentaires pour compter toutes les opérations. Puis, calculer la complexité ci-dessous.

1 + n * 2 = 2n + 1 = O(n)

- Ajouter une variable de type int nommée **compteur,** initialisée à 0 au début du programme. Ce compteur va servir à compter les opérations du programme. Afficher en fin d'exécution de la fonction la valeur de ce compteur. Exemple d'affichage dans la console :

```
>>> somme(10)
 21 opérations
45
```

- Exécuter plusieurs fois la fonction **somme( )** pour compléter le tableau suivant.

Taille *n*	10	20	50	100	500	1000
Complexité affichée	21	41	101	201	1001	2001
Formule : 2n + 1	2*10 + 1	2*20 + 1	2*50 + 1	2*100 + 1	2*500 + 1	2*1000 + 1

Lever la main pour valider ce TP.

TP2 - Exemple de calcul de complexité en récursif

- Récupérer le fichier **TP2 .py** et l'enregistrer dans le dossier **TP-Chapitre9**.
  La fonction **somme(n, c)** calcule la somme : $1 + 2 + 3 + \ldots + n - 1$ de façon récursive et le paramètre c sert à compter le nombre d'opérations.
- Compléter le **pass** dans le code pour que la fonction retourne la somme : $1 + 2 + \ldots + n - 1$ de façon récursive et affiche le nombre d'opérations.
- Exemple d'affichage dans la console :

```
>>> somme(10,0)
 31 opérations
45
```

- Exécuter plusieurs fois la fonction **somme( )** pour compléter le tableau suivant.

Taille *n*	10	20	50	100	500	1000
Complexité affichée	31	61	151	301	1501	3001
Formule : 3n + 1	3*10 + 1	3*20 + 1	3*50 + 1	3*100 + 1	3*500 + 1	3*1000 + 1

Lever la main pour valider ce TP.

Nicolas Blanc, Michel Courtieu, Lycée Anna de Noailles (Evian)

TP3 - Exemple de tri par insertion

- On considère la liste suivante : [7, 4, 8, 5, 6, 2, 1, 3].
- Compléter les cases ci-dessous en appliquant pas à pas le tri par insertion à la liste ci-dessus.

[7, 4, 8, 5 ,6, 2, 1, 3]

[4, 7, 8, 5, 6, 2, 1, 3]

[4, 7, 8, 5, 6, 2, 1, 3]

[4, 5, 7, 8, 6, 2, 1, 3]

[4, 5, 6, 7, 8, 2, 1, 3]

[2, 4, 5, 6, 7, 8, 1, 3]

[1, 2, 4, 5, 6, 7, 8, 3]

[1, 2, 3, 4? 5, 6, 7, 8]

- Ouvrir le fichier **TP3 .py** donné en ressource.
- Compléter le code, c'est à dire rajouter du code à la place de l'instruction **pass**, afin d'obtenir un tri par insertion.
- Exécuter le code et vérifier qu'on retrouve les étapes complétées ci-dessus.

Lever la main pour valider ce TP.

TP4 - Complexité moyenne du tri par insertion en nombre d'itérations

- L'objectif de ce TP est de vérifier expérimentalement que la complexité moyenne du tri par insertion en nombre d'itérations est environ de $0{,}25\,n^2 - 0{,}25\,n$ donc en $O(n^2)$. Avec n la taille de la liste à trier.
- Dupliquer le **TP3 .py** en **TP4 .py** .
- Dans le code de la fonction **tri_insertion( )** rajouter un compteur initialisé à 0 au début de la fonction et qui augmente de 1 à chaque fois qu'on passe dans la boucle **while**.
  Ce compteur va donc compter le nombre d'itérations de cet algorithme.
  Retourner (**return)** la valeur du compteur à la fin de la fonction à la place de la ligne **print(liste).** Attention à l'indentation du **return** qui ne doit pas être dans la boucle **for**.
- Après le code de la fonction **tri_insertion( )** rajouter le code ci-dessous pour évaluer la complexité moyenne du tri par insertion en nombre d'itérations.

```
from random import randint

print('Saisir la taille de la liste : ')
taille = int(input())
somme = 0
for k in range(10):
    liste1 = [randint(0, 2 * taille) for n in range(taille)]
    somme += tri_insertion(liste1)
moyenne = somme//10
print(moyenne)
```

- Exécuter le code en modifiant la taille de la liste et compléter le tableau ci-dessous pour comparer la complexité observée avec la complexité théorique de l'ordre de $0{,}25\,n^2 - 0{,}25\,n$.

Taille n	10	20	50	100	500	1000
Complexité observée	21	97	624	2453	62889	250173
$0{,}25\,n^2 - 0{,}25\,n$	22.5	95	612.5	2475	62375	249750

Lever la main pour valider ce TP.

TP5 - Complexité dans le pire des cas du tri par insertion en nombre d'itérations

- L'objectif de ce TP est de vérifier expérimentalement que la complexité dans le pire des cas du tri par insertion est de $0{,}5\,n^2 - 0{,}5\,n$ donc en $O(n^2)$.
- Dupliquer le **TP4 .py** en **TP5 .py** .
- Modifier le code de la ligne qui définit la **liste1** en compréhension, pour obtenir le pire des cas toujours en compréhension, c'est à dire une **liste classée dans l'ordre strictement décroissant**.
- Exécuter le code en modifiant la taille de la liste et compléter le tableau ci-dessous pour comparer la complexité observée avec la complexité théorique de $0{,}5\,n^2 - 0{,}5\,n$.

Taille *n*	10	20	50	100	500	1000
Complexité observée	45	190	1225	4950	124750	499500
$0{,}5\,n^2 - 0{,}5\,n$	45	190	1225	4950	124750	499500

Lever la main pour valider ce TP.

TP6 - Exemple de tri par sélection

- On considère la liste suivante : [7, 4, 8, 5, 6, 2, 1, 3].
- Compléter les cases ci-dessous en appliquant pas à pas le tri par sélection à la liste ci-dessus.

[7, 4, 8, 5, 6, 2, 1, 3]

[1, 4, 8, 5, 6, 2, 7, 3]

[1, 2, 8, 5, 6, 4, 7, 3]

[1, 2, 3, 5, 6, 4, 7, 8]

[1, 2, 3, 4, 6, 5, 7, 8]

[1, 2, 3, 4, 5, 6, 7, 8]

[1, 2, 3, 4, 5, 6, 7, 8]

[1, 2, 3, 4, 5, 6, 7, 8]

- Ouvrir le fichier **TP6 .py** donné en ressource.
- Compléter le code, c'est à dire rajouter du code à la place de l'instruction **pass**, afin d'obtenir un tri par sélection.
- Exécuter le code et vérifier qu'on retrouve les étapes complétées ci-dessus.

Lever la main pour valider ce TP.

TP7 - Complexité moyenne du tri par sélection en nombre d'itérations

- L'objectif de ce TP est de vérifier expérimentalement que la complexité moyenne du tri par sélection est de $0{,}5\,n^2 - 0{,}5\,n$ donc en $O(n^2)$. Avec n la taille de la liste à trier.
- Dupliquer le **TP6 .py** en **TP7 .py** .
- Dans le code de la fonction **tri_selection()** rajouter un compteur initialisé à 0 au début de la fonction et qui augmente de 1 à chaque fois qu'on passe dans la deuxième boucle **for**. Ce compteur va donc compter le nombre d'itérations de cet algorithme.
  Retourner (**return)** la valeur du compteur à la fin de la fonction.
- Après le code de la fonction **tri_selection()** rajouter le code ci-dessous pour évaluer la complexité moyenne du tri par sélection.

```
from random import randint

print('Saisir la taille de la liste : ')
taille = int(input())
somme = 0
for k in range(10):
    liste1 = [randint(0, 2 * taille) for n in range(taille)]
    somme += tri_selection(liste1)
moyenne = somme//10
print(moyenne)
```

- Exécuter le code en modifiant la taille de la liste et compléter le tableau ci-dessous pour comparer la complexité observée avec la complexité théorique de l'ordre de $0{,}5\,n^2 - 0{,}5\,n$.

Taille n	10	20	50	100	500	1000
Complexité observée	45	190	1225	4950	124750	499500
$0{,}5\,n^2 - 0{,}5\,n$	45	190	1225	4950	124750	499500

Lever la main pour valider ce TP.

TP8 - Temps d'exécution du tri par insertion

- L'objectif de ce TP est de calculer expérimentalement le temps d'exécution d'un tri par insertion.
- Dupliquer le **TP5 .py** en **insertion.py** . Dans ce fichier ne conserver que le code du tri par insertion : supprimer tout le reste, notamment le compteur qui fait perdre du temps d'exécution.
- Ouvrir le fichier **TP8 .py** donné en ressource. Compléter le **pass** après le code de la fonction **temps_execution( )** pour obtenir le temps d'exécution d'un tri par insertion, comme dans l'exemple ci-dessous.

```
Saisir la taille de la liste :
200
temps d'exécution : 0.003232717514038086
```

- Exécuter le code en modifiant la taille de la liste à chaque exécution et compléter le tableau ci-dessous pour donnant les temps d'exécution en fonction de la taille de la liste à trier.

Taille *n*	100	500	1000	2000	5000
Temps d'exécution en secondes à 0,0001 secondes près	0.0006	0.0111	0.0657	0.2030	1.1920

Lever la main pour valider ce TP.

TP9 - Temps d'exécution moyen pour le tri par insertion

- L'objectif de ce TP est de calculer expérimentalement le temps d'exécution d'un tri par insertion, en faisant une moyenne sur plusieurs tris pour plus de fiabilité.
- Dupliquer le **TP8 .py** en **TP9 .py** .
- En vous inspirant du code du **TP4 .py** rajouter une variable **somme** initialisée à 0 puis une boucle **for** pour calculer le temps d'exécution moyen du tri par insertion, pour 10 tris exécutés.
- Attention de ne pas faire une division entière pour le calcul de la moyenne qui est de type **float** : moyenne = somme / 10
- Exécuter le code en modifiant la taille de la liste et compléter le tableau ci-dessous donnant les temps d'exécution en fonction de la taille de la liste à trier.

Taille *n*	100	200	500	1000	2000
Temps moyen d'exécution en secondes à 0,0001 secondes près	0.002	0.0012	0.0164	0.0663	0.1823

Lever la main pour valider ce TP.

TP10 - Temps d'exécution moyen pour le tri par sélection

- L'objectif de ce TP est de calculer expérimentalement le temps d'exécution d'un tri par sélection, en faisant une moyenne sur plusieurs tris pour plus de fiabilité.
- Dupliquer le **TP7 .py** en **selection.py** . Dans ce fichier ne conserver que le code du tri par selection : supprimer tout le reste, notamment le compteur qui fait perdre du temps d'exécution.
- Dupliquer le **TP9 .py** en **TP10 .py** .
- Modifier le code pour calculer le temps moyen d'exécution pour le **tri par sélection**.
- Exécuter le code en modifiant la taille de la liste et compléter le tableau ci-dessous donnant les temps d'exécution en fonction de la taille de la liste à trier.

Taille *n*	100	200	500	1000	2000
Temps moyen d'exécution en secondes à 0,0001 secondes près	0.007	0.018	0.071	0.0290	0.1293

Lever la main pour valider ce TP.

TP11 - Comparaison des temps d'exécution moyen pour les deux tris

- L'objectif de ce TP est de comparer graphiquement le temps d'exécution des deux tris par insertion et par sélection.
- Dupliquer le **TP8 .py** en **temps.py** . Supprimer toutes les lignes de code après le code de la fonction **temps_execution( )** .
- Ouvrir le fichier **TP11 .py** donné en ressource.
- Le code actuel du **TP11** permet de tracer un nuage de points de couleur **bleue(blue)**, donnant le temps d'exécution moyen du tri par **insertion** en fonction de la taille de la liste à trier.
- Compléter les trois **pass** avec du code, afin d'obtenir sur le graphique un deuxième nuage de points de couleur **rouge (red)** donnant le temps d'exécution moyen du tri par **sélection** en fonction de la taille de la liste à trier.

Le tri qui semble le plus efficace est le tri par selection .

Lever la main pour valider ce TP.

TP12 - Exemple de tri par arbre binaire de recherche

- Sur une feuille de brouillon, dessiner l'arbre binaire de recherche qui permet de trier la liste1, puis le parcourt infixe de cet arbre binaire.

liste1 = [13, 8, 1, 5, 9, 18, 20, 2, 14, 22, 21, 15]

- Sur une feuille de brouillon, dessiner l'arbre binaire de recherche qui permet de trier la liste2, puis le parcourt infixe de cet arbre binaire.

liste2 = [12, 11, 4, 9, 20, 30, 25, 2, 10, 19, 23, 6, 0, 16, 1]

Lever la main pour valider ce TP.

TP13 - Comparaison du tri par arbre binaire avec les autres tris

- Récupérer tous les fichiers du dossier **ABR** donné en ressources et les coller dans le dossier **TP-Chapitre9** .
- Dupliquer le **TP11 .py** en **TP13 .py** .
- Dans le fichier **TP13 .py** importer au début la fonction **tri_par_ABR** depuis le module **tri_ABR** . Puis rajouter à la fin du fichier une ligne de code pour afficher en **vert(green)** un nuage de points donnant le temps d'exécution moyen du **tri par arbre binaire de recherche** en fonction de la taille de la liste à trier.

Le tri qui semble le plus efficace est le tri par arbre binaire .

Lever la main pour valider ce TP.

TP14 - Exemple de tri fusion

- Sur une feuille de brouillon, écrire toutes les étapes du tri fusion qui permet de trier la liste suivante.

liste1 = [13, 8, 1, 5, 9, 18, 20, 2]

- Sur une feuille de brouillon, dessiner l'arbre binaire de recherche qui permet de trier la liste2, puis le parcourt infixe de cet arbre binaire.

liste2 = [12, 11, 4, 9, 20, 30, 25, 2, 10, 19, 23, 6, 0, 16, 1, 7]

Lever la main pour valider ce TP.

TP15 - Programmation du tri fusion

- Ouvrir le fichier **TP15 .py** donné en ressource.
- Compléter le code dans les fonctions **tri_fusion( )** et **fusion( )**, c'est à dire rajouter du code à la place des instructions **pass**, afin d'obtenir un tri fusion.
- Exemple d'affichage possible dans la console :

```
Liste non triée :
[50, 45, 4, 19, 87, 84, 85, 75, 20, 73, 40, 2, 77, 20, 100, 68, 94, 15, 100, 17]
Liste triée :
[2, 4, 15, 17, 19, 20, 20, 40, 45, 50, 68, 73, 75, 77, 84, 85, 87, 94, 100, 100]
```

Lever la main pour valider ce TP.

TP16 - Comparaison du tri fusion avec les autres tris

- Dupliquer le **TP15 .py** en **fusion.py** . Supprimer tout le code après la définition des deux fonctions **tri_fusion( )** et **fusion( )** .
- Dupliquer le **TP13 .py** en **TP16 .py** .
- Rajouter du code pour afficher en **orange** le nuage de points donnant les temps d'exécution moyen pour le **tri fusion**.

Le tri qui semble le plus efficace est le tri par fusion .

Lever la main pour valider ce TP.

TP17 - Comparaison du tri sorted de Python avec les autres tris

- Dupliquer le **TP16 .py** en **TP17 .py** .
- Rajouter du code pour afficher en **violet** le nuage de points donnant les temps d'exécution moyen pour le **tri sorted** de Python.

Le tri qui semble le plus efficace est le tri par python .

Lever la main pour valider ce TP.

