1. Les algorithmes de tri en pratique

NOM / Prénom : *écrivez ici votre nom et votre prénom*

Objectifs :

- Observer expérimentalement le coût quadratique des algorithmes de tri par insertion et par sélection
- Constater l'inefficacité de ces deux algorithmes élémentaires dès que le tableau est trop grand
- Découvrir les fonctions de tri (efficace) offertes par Python
- (Optionnel : observer expérimentalement l'influence du tableau de départ (pire cas / meilleur cas) sur le temps d'exécution d'un tri par insertion/sélection)

2. Les tris par insertion et par sélection ont un coût quadratique

2.1 Implémentation des deux tris

On commence par rappeler l'implémentation des deux algorithmes de tri par insertion et de tri par sélection.

In [ ]:
```
# ------ TRI PAR INSERTION --------

def tri_insertion(T):
    """
    Trie le tableau T dans l'ordre croissant.
    Paramètre : T est un tableau non vide
    Sortie : aucune (car le tableau est T trié en place).
    """
    for i in range(1,len(T)):
        x = T[i]                    # x = T[i] est
```

```
l'élément à insérer

        # décalages nécessaires et recherche position
d'insertion pour T[i]
        j = i                       # j est la position
d'insertion
        while j > 0 and x < T[j-1]: # tant qu'on n'a pas
atteint le premier
                                    # et T[i] est plus
petit que ses précédents
            T[j] = T[j-1]           # on décale vers la
droite le précédent
            j = j - 1               # on passe à la
position précédente

        # insertion de T[i] en position j
        T[j] = x

# ------ TRI PAR SELECTION --------

def echange(T, i, j):
    """Echange T[i] et T[j]"""
    temp = T[i]
    T[i] = T[j]
    T[j] = temp

def tri_selection(T):
    """
    Trie le tableau T dans l'ordre croissant.
    Paramètre : T est un tableau non vide
    Sortie : aucune (car le tableau est T trié en place).
    """
    for i in range(len(T)-1):
        # recherche de l'indice du minimum dans T[i:n-1]
        ind_min = i
        for j in range(i+1, len(T)):
            if T[j] < T[ind_min]:
                ind_min = j
        # échange avec l'élément d'indice i
        echange(T, i, ind_min)
```

On peut alors tester les deux fonctions.

In [ ]:
```
tab = [4, 1, 7, 8, 1, 0, 2, 3, 5, 10]
tri_insertion(tab)
tab
```

In [ ]:
```
tab = [4, 1, 7, 8, 1, 0, 2, 3, 5, 10]
tri_selection(tab)
```

```
tab
```

Question 1 : Au moyen de `assert`, proposez un jeu de tests pour ces deux fonctions.

In [ ]:
```
# à vous de jouer !
```

2.1 Comment *mesurer* l'efficacité des tris ?

On va mesurer le temps d'exécution du tri ! De manière générale, pour *mesurer* le temps d'exécution d'une série d'instructions Python, on peut utiliser le module `time` de Python et plus précisément la fonction `time()` de ce module (documentation : https://docs.python.org/fr/3/library/time.html#time.time) qui renvoie le temps écoulé en secondes depuis le 1er janvier 1970 à 00:00:00 (UTC).

In [ ]:
```
from time import time

time()  # pour tester
```

On peut s'en servir comme un "chronomètre" : on récupère le temps `t0` juste avant les instructions à chronométrer et le temps `t1` juste après les instructions. Le temps d'exécution des instructions est alors la différence `t1 - t0`.

In [ ]:
```
from time import time

tab = [4, 1, 7, 8, 1, 0, 2, 3, 5, 10]
print(tab)

t0 = time() # on stocke dans t0 le temps de départ, avant le début du tri
tri_insertion(tab)
t1 = time() # on stocke dans t1 le temps de fin, juste après le tri

print(tab)  # pour vérifier que le tableau a été trié
print("temps :", t1 - t0)  # la différence est le temps (en SECONDES) mis par le tri
```

2.2 Efficacité des tris

Comme les deux algorithmes de tri étudiés ont un coût quadratique dans le pire cas mais aussi dans le cas moyen, nous regarderons leur efficacité en évaluant leur temps d'exécution sur des tableaux créés aléatoirement de plus en plus grand (et pas forcément dans le pire cas).

2.2.1 Tableaux de taille 100 pour démarrer

On va générer des tableaux d'entiers aléatoirement choisis entre 0 et 100. On aura besoin pour cela de la fonction `randint` du module `random`.

In [ ]:
```
from random import randint  # à exécuter pour importer la fonction randint
```

Pour construire un tableau aléatoire de taille 100, on peut procéder par compréhension comme ci-dessous :

In [ ]:
```
t100 = [randint(0, 100) for i in range(100)] # création d'un tableau de taille 100 contenant des entiers de 0 à 100.
print(t100) # tableau avant tri
tri_insertion(t100)  # tri du tableau
print(t100) # tableau après tri
```

> Si vous souhaitez vous placer dans un *pire cas*, il suffit de construire un tableau décroissant comme ci-dessous :
>
> ```
> n = 100  # taille
> t100 = [n-i for i in range(n)]
> ```

Attention : à chaque exécution, un nouveau tableau est généré. Exécutez à nouveau la cellule précédente pour vous en rendre compte.

Pour connaître le temps d'exécution pour trier ce tableau, on utilise la fonction `time()` du module `time` comme expliqué précédemment :

In [ ]:
```
#from random import randint
#from time import time

t100 = [randint(0, 100) for i in range(100)]
print(t100)

t0 = time()
tri_insertion(t100)
t1 = time()

print(t100)
print("temps :", t1 - t0)
```

Question 2 : Exécutez plusieurs fois la cellule ci-dessus pour vérifier que le tri se fait extrêmement rapidement, en quelques millisecondes (1 ms = 0,001 s = 10^{-3} s), voire quelques microsecondes (1 µs = 10^{-6} s) ou nanosecondes (1 ns = 10^{-9} s). Pourquoi obtient-on des résultats légèrement différents d'un tableau à l'autre ?

Votre réponse : (**à compléter**)

> **Remarque** : Sans entrer dans les détails, sachez également que ces temps de calcul dépendent aussi de la puissance de la machine qui exécute le code et du nombre de processus gérés *simultanément* par le processeur (programme de Terminale) Cela donne toutefois une bonne indication !

Question 3 : Procédez de même pour évaluer le temps d'un **tri par sélection** d'un tableau de taille 100. *Vous devez constater des temps du même ordre de grandeur*.

In [ ]:
```
# à vous de jouer !
```

2.2.1 Tableaux de taille 1000

On peut créer un tableau `t1000` de taille 1000 en modifiant uniquement la valeur du `range`. Et on peut voir le temps pour trier un tel tableau.

In [ ]:
```
t1000 = [randint(0, 100) for i in range(1000)]
```

```
#print(t1000)

t0 = time()
tri_insertion(t1000)
t1 = time()

#print(t1000)
print("temps :", t1 - t0)
```

Question 4 : Exécutez 5 fois la cellule précédente pour voir le temps de tri. Notez chaque temps et faites une moyenne du temps.

Votre réponse : (**à compléter**)

Question 5 : Créez et triez maintenant un tableau `t2000` de taille 2000 (on a donc *doublé* la taille du tableau par rapport à la question 4). Exécutez 5 fois la cellule et faites la moyennes des temps de tri (par insertion) du tableau.
N'hésitez pas à enlever les `print` si cela devient trop long !.

In [ ]:
```
# à vous de jouer !
```

Votre réponse : temps moyen pour trier tableau de taille 2000 : ... (**à compléter**)

Question 6 : Par combien environ a été multiplié le temps du tri par insertion entre un tableau de taille 1000 et un tableau de taille 2000 (le double) ?

Votre réponse : si la taille du tableau double, alors le temps du tri par insertion est environ multiplié par ... (**à compléter**)

Question 7 : Qu'en est-il si la taille est triplée (de 1000 à 3000 par exemple) ? Faites les essais avant de répondre.

In [ ]:
```
# à vous de jouer !
```

Votre réponse : si la taille du tableau est multipliée par 3, alors le temps du tri par insertion est environ multiplié par ... (**à compléter**)

Question 8 : Vérifiez qu'il en est de même pour un **tri par sélection**.

```
In [ ]: # à vous de jouer !
```

2.2.2 Et pour des tailles beaucoup plus grandes ?

Question 9 : Testez le temps pour trier par insertion ou par sélection, un tableau `t10k` de taille 10 000. (Voire un tableau `t100k` de taille 100 000 si vous êtes patient 😁).

```
In [ ]: # à vous de jouer !
```

Votre réponse : pour trier par insertion/sélection un tableau de taille 10 000 il faut environ ... (**à compléter**).

Question 10 : Admettons qu'il faille 10 secondes pour trier par insertion un tableau de taille 10 000.

- Quel temps faudrait-il environ pour trier un tableau de taille 100 000 (10 fois plus grand) ?
- Et combien d'heures faudrait-il pour un tableau de 1 million de valeurs ?
- Et combien de jours faudrait-il pour un tableau de 10 millions de valeurs ? 🕛
- Et combien de vies faudrait-il pour un tableau de 1 milliard de valeurs ? ☠

Votre réponse : (**à compléter**).

- pour un tableau de taille 100 000 : ...
- pour un tableau de taille 1 000 000 : ...
- pour un tableau de taille 10 000 000 : ...
- pour un tableau de taille 1 000 000 000 : ...

> Autant vous dire qu'il est inutile d'essayer de trier par insertion/sélection des tableaux de taille supérieure ou égale à 1 million, c'est très inefficace. Il existe cependant des algorithmes de tris plus efficaces. Python vous propose deux

> façons de trier un tableau avec un algorithme beaucoup plus efficace, c'est l'objet de la dernière partie !

3. Fonctions de tris offertes par Python

Python fournit deux fonctions permettant de trier de manière plus efficace un tableau. Elles se présentent de deux façons différentes, selon que l'on veuille obtenir une copie triée du tableau, sans le modifier (`sorted`), ou au contraire modifier le tableau pour le trier (`sort`).

> L'algorithme utilisé par ces fonctions est plus efficace : son coût est de l'ordre de $n \log_2(n)$, nettement meilleur que n^2.

3.3 La fonction `sorted`

Celle-ci prend en argument un tableau et renvoie un **nouveau** tableau, trié, contenant les mêmes éléments.

In [ ]:
```
t = [12, 5, 3, 6, 8, 10]
sorted(t)
```

On peut voir que le tableau de départ n'a pas été modifié :

In [ ]:
```
t
```

> **Remarque** : Cette fonction permet aussi de trier des chaînes de caractères, en utilisant l'odre alphabétique.

In [ ]:
```
sorted(["poire", "pomme", "cerise", 'kiwi'])
```

3.4 La fonction `sort`

Celle-ci s'applique à un tableau, ne renvoie rien, mais **modifie** le tableau d'origine.

In [ ]:
```
t = [5, 1, 2, 17, 9, 1, 8]
t.sort() # ne renvoie rien
```

Rien n'est renvoyé mais le tableau `t` d'origine a été modifié :

In [ ]:
```
t
```

> On peut aussi trier directement un tableau de chaînes de caractères avec `sort`.

In [ ]:
```
fruits = ["poire", "pomme", "cerise", 'kiwi']
fruits.sort()
fruits
```

Question 11 : Vérifiez qu'en utilisant l'une de ces deux méthodes, le tri d'un tableau d'un million de valeurs est instantané. *Remarque* : n'hésitez pas à créer le tableau de taille 1 million dans une autre cellule que celle du tri et de la mesure (car la construction du tableau peut être longue et fausser votre perception du temps).

In [ ]:
```
# à vous de jouer !
```

4. Pire cas du tri par insertion (OPTIONNEL)

On a vu dans le cours qu'un tableau trié dans l'ordre décroissant était le pire cas pour le tri par insertion. L'objectif de cette partie est d'en mesurer concrètement l'impact sur le temps de tri.

On peut construire un tableau de taille 1000 trié dans l'ordre décroissant de la façon suivante.

In [ ]:
```
n = 1000 # taille du tableau
t = [n-i for i in range(n)]
print(t)
```

Question 12 : Évaluer le temps pour trier par insertion ce tableau.

```
In [ ]: # à vous de jouer !
```

Question 13 : Et pour un tableau de taille 10 000 ?

> Cela doit prendre quelques secondes, voire dizaines de secondes et vous devez constater que le temps est supérieur à ceux obtenu à la question 9 (où le tableau est crée aléatoirement donc il y a peu de chances d'obtenir un pire cas).

```
In [ ]: # à vous de jouer !
```

Question 14 : Le meilleur cas de l'algorithme de **tri par insertion** est celui où le tableau est déjà trié par ordre croissant au départ. Il suffit alors de faire une comparaison par tour de boucle (et constater que chaque élément est déjà au bon endroit). Créez un tableau de taille 10 000 trié par ordre croissant (par exemple, le tableau [1, 2, 3,...,10000] puis mesurez le temps pour le trier par insertion.

> Le tri est normalement quasi instantané donc beaucoup plus rapide que pour le pire cas et que les cas générés aléatoirement.

```
In [ ]: # à vous de jouer !
```

> **Explications** : Si le tableau est déjà trié, alors il n'y a qu'une comparaison par tour de boucle. Comme il y a $n-1$ tours de boucle, cela fait donc $n-1$ comparaisons donc on trouve un coût de l'ordre de n, qui est linéaire (comme les algorithmes de recherche du nombre d'occurrences, de min/max, de calcul de moyenne...). Cela signifie que si on multiplie par k la taille du tableau (déjà trié) de départ, le temps de l'algo de tri est multiplié par k également. Ainsi, s'il ne faut que quelques ms, disons 5 ms, pour trier un tel tableau de taille 10 000, alors pour trier un tableau trié de taille 1 000 000 (multiplication par 100) il ne faudra que $5\times 100=500$ ms = $0,5$ s (contre $5\times100^2=50000$ s $\simeq 13$ min si le coût était quadratique).

Question 15 : Constatez que quel que soit le tableau de départ (trié par ordre décroissant, par ordre croissant, ou créé aléatoirement) il n'y a pas (autant) de

différences de temps pour le trier si on utilise le **tri par sélection**.

> Cela prend dans tous les cas plusieurs secondes, comme à la question 9

```
In [ ]: # à vous de jouer !
```

> **Explication** : On a vu dans le cours que pour un tableau `T`, le nombre de comparaisons pour le trier par sélection est toujours égale à $\dfrac{n(n-1)}{2}$, et ce quel que soit le tableau `T` de départ. Il est donc normal d'avoir des temps similaires.

Germain BECKER & Sébastien POINT, Lycée Mounier, ANGERS