Python – Algorithmes de tri

Préparatifs

Dans tous les exercices de cette feuille, et sauf mention contraire, les modifications apportées aux tableaux sont effectuées directement sur les tableaux. Il ne faut donc pas rajouter de `return tab` à la fin des fonctions. Le tableau passé en paramètre est modifié lors de l'appel de la fonction et les tableaux étant muables en Python, les modifications persistent après l'appel de la fonction.
On rappelle que la fonction **range** peut prendre plusieurs paramètres afin de modifier la valeur de départ ou encore l'écart qu'il y a entre deux valeurs successives. Cela permet, notamment de compter à rebours.

```
>>> list(range(4, 8))
[4, 5, 6, 7]
>>> list(range(4, 8))
[4, 5, 6, 7]
>>> list(range(4, 45, 3))
[4, 7, 10, 13, 16, 19, 22, 25, 28, 31, 34, 37, 40, 43]
>>> list(range(10, -1, -1))
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Dans les exemples ci-dessus, la conversion en liste sert juste à forcer la génération de toutes les valeurs.
Nous allons également utiliser le module `random`. Il faut rajouter la commande `import random` au début du fichier. En faisant une importation de la sorte, il faut rajouter `random.` avant chaque appel à une fonction du module.

```
>>> random.random()      # génération d'un nombre entre 0 et 1
0.4671064597997632
>>> random.randint(0, 100) # un entier entre 0 et 100 inclus
24
```

On peut utiliser la fonction suivante pour générer aléatoirement des tableaux d'une taille donnée :

```
def genere_tableau(n):
    return [random.random() for _ in range(n)]
```

```
>>> genere_tableau(4)
[0.34253573685847016, 0.5447961372120071, 0.16355941327056156, ...
>>> genere_tableau(100)
[0.8066534034910527, 0.8467728275829794, 0.10028352226518222, ...
```

Exercice 1 : Compléter la fonction `echange(tab, i, j)` qui prend un tableau `tab` et qui inverse les éléments en position `i` et `j`. Les valeurs sont directement échangées dans le tableau et cette fonction ne renvoie rien.

```
def echange(tab, i, j):
    tmp = tab[i]
    tab[...] = tab[...]
    tab[...] = ...
```

```
>>> tab = [4, 9, 1, 10]
>>> echange(tab, 0, 2)
>>> tab
[1, 9, 4, 10]
```

Exercice 2 : Compléter la fonction `est_trie(tab)` qui renvoie **`True`** si et seulement si le tableau `tab` est trié. On considère qu'un tableau vide est trié.

```
def est_trie(tab):
    for i in range(len(tab)-1): # On s'arrête sur l'avant dernier élément
        if ...:
            return False
    return True
```

```
>>> est_trie([4, 1, 2])
False
>>> est_trie([1, 2, 4])
True
```

Exercice 3 : Donner plusieurs tests, à l'aide de **`assert`**, pour vérifier la correction de la fonction `est_trie(tab)`. Rajouter ces tests dans le fichier après la définition de la fonction. Par exemple, on peut mettre :

```
assert not est_trie([4, 1, 2])
assert est_trie([1, 2, 4])
```

Le tri à bulles

Exercice 4 : Compléter la fonction `tri_bulles(tab)` qui trie les valeurs du tableau `tab` dans l'ordre croissant.

```
def tri_bulles(tab):
    n = len(tab)
    #nb_tests = 0  # Pour plus tard
    #nb_echanges = 0  # Pour plus tard
    for i in range(...): # On va jusqu'à l'avant dernier
        for j in range(..., i, -1): # On va du dernier à i+1
            if ...:
                echange(tab, j-1, j)
    #print(tab)  # Pour tester si on veut
    #return nb_tests, nb_echanges # Pour plus tard
```

```
>>> tab = [4, 6, 1, 3, 0]
>>> tri_bulles(tab)
>>> tab
[0, 1, 3, 4, 6]
```

Exercice 5 : Compléter la fonction `test_tri_bulles(n, k)` qui génére aléatoirement `k` tableaux de taille `n`, les trie avec le tri à bulles. La fonction renvoie `True` si tous les tableaux sont effectivement triés et `False` sinon.

```
def test_tri_bulles(n, k):
    for i in range(...):
        tab = genere_tableau(...)
        tri_bulles(...)
        if not ...:
            return False
    return True
```

```
>>> test_tri_bulles(1000, 50)
True
```

Exercice 6 : Modifier la fonction `tri_bulles(tab)` pour qu'elle renvoie un couple de valeurs `nb_tests, nb_echanges`, où `nb_tests` est le nombre de tests qui a été effectué et `nb_echanges` le nombre d'échanges.

```
>>> tri_bulles([1, 2, 3, 4])
(6, 0)
>>> tri_bulles([4, 3, 2, 1])
(6, 6)
>>> tri_bulles([1, 3, 2, 4])
(6, 1)
```

Le tri par sélection

Exercice 7 : Compléter la fonction `tri_selection(tab)` qui trie les valeurs du tableau `tab` dans l'ordre croissant.

```
def tri_selection(tab):
    n = len(tab)
    #nb_tests = 0
    #nb_echanges = 0
    for i in range(...):
        k = ...
        for j in range(..., n):
            if tab[j] < tab[k]:
                k = ...
        if i != k:
            echange(tab, ..., ...)
    #return nb_tests, nb_echanges
```

```
>>> tab = [4, 6, 1, 3, 0]
>>> tri_selection(tab)
>>> tab
[0, 1, 3, 4, 6]
```

Exercice 8 : Écrire une fonction `test_tri_selection(n, k)` qui génére aléatoirement `k` tableaux de taille `n`, les trie avec le tri par sélection. La fonction renvoie **`True`** si tous les tableaux sont effectivement triés et **`False`** sinon.

```
>>> test_tri_selection(1000, 50)
True
```

Exercice 9 : Modifier la fonction `tri_selection(tab)` pour qu'elle renvoie un couple de valeurs `nb_tests, nb_echanges`, où `nb_tests` est le nombre de tests qui a été effectué et `nb_echanges` le nombre d'échanges.

```
>>> tri_selection([1, 2, 3, 4])
(6, 0)
>>> tri_selection([4, 3, 2, 1])
(6, 2)
>>> tri_selection([1, 3, 2, 4])
(6, 1)
```

Le tri par insertion

Exercice 10 : Compléter la fonction `tri_insertion(tab)` qui trie les valeurs du tableau `tab` dans l'ordre croissant.

```
def tri_insertion(tab):
    n = len(tab)
    #nb_tests = 0
    #nb_echanges = 0
    for i in range(..., n):
        j = ...
        while j > ... and tab[...] > tab[...]:
            echange(tab, j-1, j)
            j = ...
    #return nb_tests, nb_echanges
```

```
>>> tab = [4, 6, 1, 3, 0]
>>> tri_insertion(tab)
>>> tab
[0, 1, 3, 4, 6]
```

Exercice 11 : Écrire une fonction `test_tri_insertion(n, k)` qui génére aléatoirement `k` tableaux de taille `n`, les trie avec le tri par insertion. La fonction renvoie **`True`** si tous les tableaux sont effectivement triés et **`False`** sinon.

```
>>> test_tri_insertion(1000, 50)
True
```

Exercice 12 : Modifier la fonction `tri_insertion(tab)` pour qu'elle renvoie un couple de valeurs `nb_tests, nb_echanges`, où `nb_tests` est le nombre de tests qui a été effectué et `nb_echanges` le nombre d'échanges. Pour la boucle **`while`**, on ne comptera que les tests effectués sur les éléments du tableau et pas sur la valeur de `j`.

```
>>> tri_insertion([1, 2, 3, 4])
(3, 0)
>>> tri_insertion([4, 3, 2, 1])
(6, 6)
>>> tri_insertion([1, 3, 2, 4])
(4, 1)
```

Comparaison des trois algorithmes de tri

Afin de comparer le fonctionnement des trois algorithmes, nous allons utiliser la fonction suivante :

```
def comparaison(tab):
    t1 = tab.copy()
    nb_t, nb_e = tri_bulles(t1)
    print(f"Tri a bulles :      {nb_t:2} tests et {nb_e:2} échanges")
    t1 = tab.copy()
    nb_t, nb_e = tri_selection(t1)
    print(f"Tri par sélection : {nb_t:2} tests et {nb_e:2} échanges")
    t1 = tab.copy()
    nb_t, nb_e = tri_insertion(t1)
    print(f"Tri pas insertion : {nb_t:2} tests et {nb_e:2} échanges")
```

```
>>> comparaison([1, 2, 3 ,4])
Tri a bulles :       6 tests et  0 échanges
Tri par sélection :  6 tests et  0 échanges
Tri pas insertion :  3 tests et  0 échanges
>>> comparaison([4, 3, 2 ,1])
Tri a bulles :       6 tests et  6 échanges
Tri par sélection :  6 tests et  2 échanges
Tri pas insertion :  6 tests et  6 échanges
>>> comparaison(genere_tableau(100))
Tri a bulles :      4950 tests et 2523 échanges
Tri par sélection : 4950 tests et 95 échanges
Tri pas insertion : 2616 tests et 2523 échanges
>>> comparaison(genere_tableau(100))
Tri a bulles :      4950 tests et 2338 échanges
Tri par sélection : 4950 tests et 94 échanges
Tri pas insertion : 2435 tests et 2338 échanges
>>> comparaison(genere_tableau(10000))  # attention, c'est long
Tri a bulles :      49995000 tests et 24845070 échanges
Tri par sélection : 49995000 tests et 9994 échanges
Tri pas insertion : 24855062 tests et 24845070 échanges
```

Exercice 13 : Utiliser cette fonction avec différents tableaux, choisis ou générés.

On peut remarquer que le tri à bulles est systématiquement le "pire" des tris. Le tri par sélection utilise systématiquement le moins d'échanges et le tri par insertion le moins de tests.
Nous allons maintenant mesurer le temps moyen mis par les fonctions de tri. Avant de pouvoir le faire, il faut faire une fonction pour calculer la valeur moyenne.

Exercice 14 : Écrire une fonction `moyenne(liste)` qui renvoie la moyenne des valeurs de la liste `liste`.

```
>>> moyenne([3, 1, 90, 0])
23.5
```

Nous pouvons maintenant rajouter les fonctions suivantes afin de mesurer le temps moyen d'exécution des algorithmes de tri.

```
import time    # A mettre au début du fichier

def temps_tri(tab, algo_tri):
    debut = time.time()
    algo_tri(tab)
    return time.time()-debut

def comparaison_temps(n, k):
    t_bulles = []
    t_selection = []
    t_insertion = []
    for i in range(k):
        tab = genere_tableau(n)
        t_bulles.append(temps_tri(tab.copy(), tri_bulles))
        t_selection.append(temps_tri(tab.copy(), tri_selection))
        t_insertion.append(temps_tri(tab.copy(), tri_insertion))
    print(f"Tri à bulles : {moyenne(t_bulles):.4}")
    print(f"Tri par sélection : {moyenne(t_selection):.4}")
    print(f"Tri par insertion : {moyenne(t_insertion):.4}")
```

La fonction `temps_tri(tab, algo_tri)` permet de calculer temps que met la fonction `algo_tri` pour trier le tableau `tab`. La fonction `comparaison(n, k)` affiche le temps moyen pour chaque algorithme pour trier `k` tableaux de longueur `n`.

Exercice 15 : Comparer les temps d'exécution des trois algorithmes et observez le temps d'exécution qui "explose" lorsqu'on augmente la taille de la liste. Attention à ne pas mettre des valeurs trop grandes ou à limiter le nombre de répétitions.

```
>>> comparaison_temps(100, 100)
Tri à bulles : 0.001801
Tri par sélection : 0.0007815
Tri par insertion : 0.001578
>>> comparaison_temps(1000, 100)
Tri à bulles : 0.1967
Tri par sélection : 0.08351
Tri par insertion : 0.1647
>>> comparaison_temps(10000, 1)
Tri à bulles : 20.57
Tri par sélection : 8.284
Tri par insertion : 16.88
```

Représentation graphique

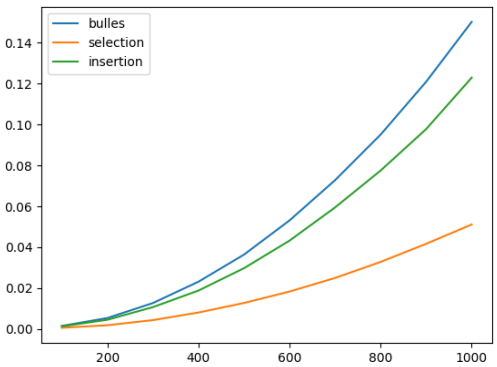
Afin de mieux comprendre l'évolution du temps de calcul en fonction de la taille du tableau, nous allons utiliser la bibliothèque `matplotlib` afin de faire un graphique. Nous allons rajouter la fonction suivante, qui permet d'afficher un graphique.

```
import matplotlib.pyplot as plt

def graphique_temps(n_max, k, points=10):
    t_taille = [] # pour les abscisses
    t_bulles = [] # Tableaux des ordonnées pour les 3 algos
    t_selection = []
    t_insertion = []
    pas = n_max//points # distance entre les abscisses des points
    n = pas
    for _ in range(points):
        t_taille.append(n)
        temps_bulles = 0          # somme des temps de calcul
        temps_selection = 0       # pour des listes de taille n
        temps_insertion = 0       # pour les 3 algos
        for _ in range(k):
            tab = genere_tableau(n)
            temps_bulles += temps_tri(tab.copy(), tri_bulles)
            temps_selection += temps_tri(tab.copy(), tri_selection)
            temps_insertion += temps_tri(tab.copy(), tri_insertion)
        # On rajoute les temps moyens aux listes des ordonnées
        t_bulles.append(temps_bulles/k)
        t_selection.append(temps_selection/k)
        t_insertion.append(temps_insertion/k)
        n = n + pas
    # affichage des courbes
    plt.plot(t_taille, t_bulles, label="bulles")
    plt.plot(t_taille, t_selection, label="selection")
    plt.plot(t_taille, t_insertion, label="insertion")
    plt.legend()
    plt.show()
```

La valeur de `n_max` correspond à la taille maximale des tableaux qui seront utilisés, celle de `k` est le nombre de tableaux qui seront utilisés à chaque fois pour faire des temps moyens et `points` indique le nombre de points qui seront utilisés pour tracer chaque courbe. Ainsi, si `n_max` vaut 1000 et `points` vaut 10, les tableaux auront des tailles de 100, 200, 300, ..., jusqu'à 1000.

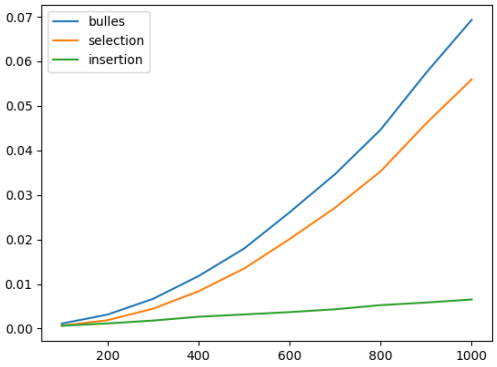
Exercice 16 : Rajouter la fonction et la tester. Attention à ne pas prendre de trop grandes valeurs. Vous pouvez aussi modifier la valeur de `k` pour aller plus vite, ou au contraire, pour avoir des courbes plus lisses.
Voici le résultat pour `graphique_temps(1000, 10)`.



On peut remarquer que sur des tableaux générés au hasard, le tri par sélection s'en sort mieux que les autres. Mais la forme des courbes montre bien l'aspect quadratique de chacun de ces algorithmes.
Par contre, si on travaille sur des tableaux "presque triés", c'est-à-dire où seulement un petit nombre de valeurs ne sont pas bien placées, les résultats sont bien différents.

```
def genere_tableau_presque_trie(n, k=20):
    tab = list(range(n))
    for _ in range(k):
        # on fait k échanges au hasard
        echange(tab, random.randint(0, n-1), random.randint(0, n-1))
    return tab
```

En mettant `k` à 20 et en modifiant la fonction `graphique_temps` pour utiliser cette fonction, on obtient la courbe ci-contre. On peut voir que le tri par insertion met nettement moins de temps que les autres et que sa courbe est presque droite, montrant un coût quasi-linéaire.
Il peut donc être utile de connaître le type de tableaux que l'on aura à trier, afin de choisir l'algorithme le plus adapté.



Et Python, il fait comment lui ?

Puisque le tri est essentiel pour de nombreux algorithme, Python possède nativement des fonctions pour trier un tableau ou une liste. Il y a deux méthodes pour les utiliser : `sorted(tab)` qui renvoie une copie de `tab` trié, sans modifier `tab` et `tab.sort()` qui trie `tab` sur place.

```
>>> tab = [54, 4, 17, 6, 90]
>>> sorted(tab)
[4, 6, 17, 54, 90]
>>> tab
[54, 4, 17, 6, 90]
>>> tab.sort()
>>> tab
[4, 6, 17, 54, 90]
```

La méthode de tri utilisé s'appelle Timsort. Elle a une complexité $O(n\log(n))$. L'efficacité est incomparable à côté de nos algorithmes de trois algorithmes de tris :

```
>>> tab = genere_tableau(10000)
>>> temps_tri(tab.copy(), tri_selection)
5.3533689975738525
>>> temps_tri(tab.copy(), sorted)
0.0024383068084716797
```

Là où le tri par sélection, le plus rapide des trois sur nos tests, met 5 secondes pour trier un tableau de longueur 10000, le tri de Python mets 2 millièmes de seconde. Cette grande différence s'explique par plusieurs facteurs. Tout d'abord, l'algorithme utilisé est bien plus efficace. Mais également, le code informatique utilisé par Python pour ses fonctions interne est compilé. Il est donc directement compréhensible par l'ordinateur. Au contraire, notre fonction est interprétée par Python. Elle est traduite en langage machine pendant l'exécution, ce qui prend plus du temps qu'avec du code compilé.