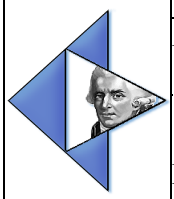


Université de Corse - Pasquale PAOLI	
Diplôme : **Licence SPI 3ème année**	2021-2022
UE : Ateliers de programmation	
Atelier 5 : Tris, expérimentation de la complexité en temps	
Enseignants : Paul-Antoine BISGAMBIGLIA, Marie-Laure NIVET, Evelyne VITTORI	

Objectifs : travailler sur le tri des valeurs dans une structure de données de type liste et expérimenter l'effet de la complexité des algorithmes sur les temps d'exécution.
Vous allez écrire différents algorithmes et tester de façon expérimentale lesquels sont les plus efficaces en fonction de la taille des listes d'éléments à trier ainsi que de l'état des listes : complètement triées, inversement triées, non triées du tout, etc.
Consignes : Nous vous demandons **d'appliquer scrupuleusement les bonnes pratiques** que nous avons vues ensemble tout au long de ces journées de codage.

PARTIE 1 - EXERCICES ESSENTIELS

Préalable, la gestion de l'aléatoire en python : Il existe en python un module permettant de générer des nombres aléatoires, c'est le module `random`. Pour l'utiliser il faut, comme pour tous les modules en python, l'importer dans votre code :

```
#Importer toutes les fonctions du module pyhton random
from random import *

#Tirage aléatoire d'un réel, entre 0 et 1, bornes incluses
random()
```

EXERCICE 1 - Génération de listes de nombres entiers aléatoires

Écrivez une fonction `gen_list_random_int` qui génère et retourne une liste de nombres aléatoires `int_nbr` compris entre `int_binf` et `int_bsup`. Si aucun paramètre n'est passé à la fonction alors `gen_list_random_int` génèrera par défaut `10` nombres compris entre `0` (inclus) et `10` (exclus).

Annexe : Spécification de valeurs par défaut pour les paramètres d'entrée des fonctions.
Dans ce cas on n'a pas besoin d'utiliser d'annotation de type puisque la valeur par défaut illustre le type attendu pour la variable.

```
#Exemple de fct avec un paramètre d'entrée avec valeur par défaut
def ma_fonction(un_parametre_int=0):
    print('valeur du paramètre',un_parametre_int)

ma_fonction(10)
ma_fonction()
```

EXERCICE 2 - Mélange des éléments d'une liste

Ecrivez une fonction `mix_list` qui prend en paramètre une liste `list_to_mix` de n'importe quoi potentiellement triée et qui retourne la liste mélangée.

Attention : la liste passée en paramètre ne doit pas subir de modification.

Remarque : vous n'avez bien sûr pas le droit d'utiliser la fonction `shuffle(lst)` proposée dans le module `random` qui prend en paramètre une liste `lst` et retourne une liste résultat ayant les mêmes éléments de base que `lst` mais dans un ordre quelconque.

Un début de code pour tester votre fonction, nécessaire mais non suffisant :

```
# Test de votre code
lst_sorted=[i for i in range(10)]
print(lst_sorted)
print('Liste triée de départ',lst_sorted)
mixed_list=mix_list(lst_sorted)
print('Liste mélangée obtenue',mixed_list)
print('Liste triée de départ après appel à mixList, elle doit être
inchangée',lst_sorted)
#assert (cf. doc en ligne) permet de vérifier qu'une condition
#est vérifiée en mode debug (désactivable)
assert lst_sorted != mixed_list,"Les deux listes doivent être différente
après l'appel à mixList..."
```

EXERCICE 3 - Choix aléatoire d'un élément dans une liste

Écrivez une fonction `choose_element_list` qui prend en paramètre une liste `list_in_which_to_choose` de n'importe quoi et qui retourne un élément de cette liste choisi au hasard. La liste de départ ne doit pas être modifiée lors de l'appel de la fonction.

Remarque : vous n'avez bien sûr pas le droit d'utiliser la fonction `choice(lst)` proposée dans le module `random` qui prend en paramètre une liste `lst` et retourne un élément de cette liste `lst` choisi au hasard.

```
# Test de votre code
print('Liste triée de départ',lst_sorted)
e1 = choose_element_list(lst_sorted)
print('A la première exécution',e1,'a été sélectionné')
e2 = choose_element_list(lst_sorted)
print('A la deuxième exécution',e2,'a été sélectionné')
assert e1 != e2,"Attention vérifiez votre code, pour deux sélections de
suite l'élément sélectionné est le même !"
```

EXERCICE 4 - Choix aléatoire de plusieurs éléments dans une liste

Écrivez une fonction `extract_elements_list` qui prend en paramètre une liste `list_in_which_to_choose` de n'importe quoi, un `int_nbr_of_element_to_extract` et qui retourne une liste composée de `int_nbr_of_element_to_extract` éléments de la liste de départ choisis au hasard. La liste de départ ne doit pas être modifiée.

Remarque : vous n'avez bien sûr pas le droit d'utiliser la fonction `sample(lst,n)` proposée dans le module `random` qui prend en paramètre une liste `lst` et retourne une liste composée de n éléments de cette liste `lst` choisis au hasard.

```
# Test de votre code
print('Liste de départ',lst_sorted)
sublist = extract_elements_list(lst_sorted,5)
print('La sous liste extraite est',sublist)
print('Liste de départ après appel de la fonction est',lst_sorted)
```

PARTIE 2.1 - EXERCICES APPROFONDIS

EXERCICE 5 - Mesure et comparaison des temps d'exécution v.1

Préalable : Vous allez ici comparer les temps d'exécutions de vos fonctions avec celles fournies par python dans le module `random`. Pour cela vous allez utiliser le module `time` qui permet de récupérer le temps système. La mesure du temps système peut s'effectuer avec la methode `perf_counter()` du module time.

Extrait de la documentation Python 3.X :

`time.perf_counter()->float` *Return the value (in fractional seconds) of a performance counter, i.e. a clock with the highest available resolution to measure a short duration. It does include time elapsed during sleep and is system-wide. The reference point of the returned value is undefined, so that only the difference between the results of consecutive calls is valid.*

```
# Exemple d'utilisation
import time

#Pour mesurer le temps d'exécution nous avons à notre disposition
#la fonction perf_counter()
n = 10000000
#Récupération du temps système et démarrage du chronomètre
start_pc = time.perf_counter()
for i in range(n):
    #on ne fait rien…
    None
end_pc = time.perf_counter()

elapsed_time_pc = end_pc-start_pc

print("Temps écoulé entre les deux mesures : ",elapsed_time_pc)
print("Temps estimé pour une instruction",elapsed_time_pc/n)
# Exécutez ce code et vérifiez par vous-même la variabilité des mesures.
```

Attention les résultats obtenus dans le cadre de cet exercice ne peuvent servir qu'à **comparer** des algorithmes - exécutés sur une même machine dans une même configuration - entre eux. En effet le temps d'exécution lui-même n'est pas une grandeur pertinente puisqu'il dépend de la machine sur laquelle l'algorithme est exécuté ainsi que de son état. Le type de votre processeur, son nombre de cœurs, sa fréquence, sa mémoire cache, la RAM, votre système d'exploitation peuvent influer sur les temps d'exécution. Les résultats ne sont donc pas comparables avec ceux de vos collègues.

Pour obtenir des résultats comparables il faut calculer la complexité (nombre d'exécution des instructions) de l'algorithme. C'est ce que vous ferez au semestre 2.

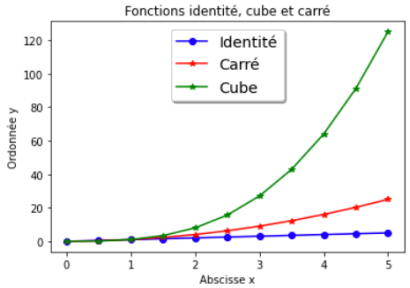
Afin de se mettre dans la meilleure configuration possible, respectez les quelques règles suivantes lorsque vous lancez les tests :

- aucune autre tâche ne doit occuper le CPU pendant le test (pas de streaming, pas de téléchargement, aucune application en tâche de fond).
- ne vous contentez jamais d'une seule mesure de temps, ce ne serait pas représentatif. Effectuez toujours au minimum une centaine de tests (plus si possible) et ne considérez que la moyenne des mesures obtenues sur ces 100 tests.
- testez toujours plusieurs configurations de même type et de même taille pour faire votre moyenne. Par exemple :
    - Configuration 1 : listes triées à `n` valeurs
    - Configuration 2 : listes en ordre inversement triées à `n` valeurs
    - Configuration 3 : listes quelconques à `n`valeurs
- variez les tailles de listes que vous considérez
    - `n` de l'ordre de 10
    - `n` de l'ordre de 100
    - `n` de l'ordre de 1000
    - ...
    - `n` de l'ordre de 100000000 ou plus

Préalable : Affichage de courbes.

```
import matplotlib.pyplot as plt
import numpy as np
#Ici on décrit les abscisses
#Entre 0 et 5 en 10 points
x_axis_list = np.arange(0,5.5,0.5)

fig, ax = plt.subplots()
#Dessin des courbes, le premier paramètre
#correspond aux point d'abscisse le
#deuxième correspond aux points d'ordonnées
#le troisième paramètre, optionnel permet de
#choisir éventuellement la couleur et le marqueur
ax.plot(x_axis_list,x_axis_list,'bo-',label='Identité')
ax.plot(x_axis_list,x_axis_list**2, 'r*-', label='Carré')
ax.plot(x_axis_list,x_axis_list**3,'g*-', label='Cube')
ax.set(xlabel='Abscisse x', ylabel='Ordonnée y',
       title='Fonctions identité, cube et carré')
ax.legend(loc='upper center', shadow=True, fontsize='x-large')
#fig.savefig("test.png")
plt.show()
```



Écrivez une fonction `perf_mix` qui permet de calculer le temps d'exécution moyen des deux fonctions de mélange (mix_list et random.shuffle) passées en paramètre dans une même configuration, c'est-à-dire pour une même liste.

`perf_mix` possède quatre paramètres :

- Les deux fonctions à tester de type callable
- Une liste d'entiers représentant les tailles de liste pour lesquelles on effectue la comparaison. Par exemple : [10, 500, 5000, 50000, 100000]

- Un nombre entier représentant le nombre d'exécution moyen nécessaire au calcul de la moyenne des temps.
  Par exemple : 10

Cette fonction renvoie en résultat un doublet (tuple à deux entrées : la première entrée contient les résultats pour la première fonction, la deuxième entrée contient les résultats pour la deuxième fonction). Chacun des résultats est constitué de la liste des temps d'exécution moyens pour chacune de ces tailles de liste.
Dans le cas précédent le doublet résultat sera de la forme :

```
([temps d'exécution moyen pour ces tailles pour fonction 1]),
[temps d'exécution moyen pour ces tailles pour fonction 2]))
```

Question 1. Test comparatif expérimental de mix_list et shuffle

Ecrivez le code permettant de tester expérimentalement, c'est à dire de mesurer et afficher les temps d'exécution des deux fonctions permettant de mélanger les listes : la votre `mix_list` et celle présente dans le module `random` nommée `shuffle`. Vous devez effectuer les tests pour des listes de différentes tailles et conserver ces résultats afin d'afficher les courbes des temps d'exécution en fonction du nombre d'éléments dans la liste. Vous superposerez les deux courbes pour les deux fonctions à comparer sur une même figure.

Question 2. Test comparatif expérimental de extract_elements_list et sample

Écrivez le code permettant de tester expérimentalement les deux fonctions permettant d'extraire aléatoirement `int_nbr_of_element_to_extract` éléments : la votre `extract_elements_list` et celle présente dans le module `random` nommée `sample`. Vous devez effectuer les tests pour des listes de différentes tailles et conserver ces résultats dans afin d'afficher les courbes des temps d'exécution en fonction du nombre d'éléments dans la liste. Vous superposerez les deux courbes pour les deux fonctions à comparer sur une même figure.

PARTIE 2.2 - EXERCICES APPROFONDIS

EXERCICE 6 - Tri à votre mode

Question 1. Tri

Écrivez le code d'une fonction `sort_list` prenant en paramètre une liste d'élément et retournant, sans modifier la liste de départ, une nouvelle liste constituée des éléments de la liste de départ triés par ordre croissant.

Remarque : vous n'avez bien sûr pas le droit d'utiliser la fonction `sorted(lst)` proposée par défaut, qui prend en paramètre une liste `lst` et retourne une liste composée des éléments de `lst` triés par ordre croissant.

Question 2. Comparaison expérimentale des deux fonctions de tri

Ecrivez le code permettant de tester expérimentalement les deux fonctions permettant de trier les listes : la votre `sort_list` et celle présente par défaut nommée `sorted`. Vous devez effectuer les tests pour

des listes de différentes tailles de liste mais aussi pour différentes configurations (conf1 : listes quelconques ; conf2 : listes triées par ordre croissant ; conf3 : listes inversement triées) et conserver ces résultats dans afin d'afficher les courbes des temps d'exécution en fonction du nombre d'éléments dans la liste. Vous superposerez les deux courbes pour les deux fonctions à comparer sur une même figure.

EXERCICE 7 - Implémentation des algorithmes de tri classiques

Pour chacun des algorithmes il vous est demandé en premier lieu de vous approprier l'algorithme et de vous entrainer à l'expliquer à quelqu'un d'autre. Vous devez être capable de le dérouler à la main en l'expliquant sur une courte liste. Une fois que vous vous l'êtes réellement approprié, implémentez-le en python. Enfin testez-le sur le cas simple d'une liste ayant une taille raisonnable.

Question 1. Tri stupide

Ecrivez une fonction `stupid_sort` qui prend en paramètre une liste de nombres `lst_to_sort` et retourne une liste comportant les mêmes éléments mais triés. Pour effectuer ce tri on mise sur le hasard... C'est pourquoi il est appelé tri stupide (cf. description et annimation sur Wikipedia). La démarche est on ne peut plus simple : on mélange les éléments et on regarde si par chance ils sont triés dans l'ordre souhaité - décroissant ou croissant - et on réitère jusqu'à ce que ce soit le cas. L'idée générale de l'algorithme est illustrée en *pseudo-code* ci-dessous :

```
fonction tri_stupide (liste)
    tant que la liste n'est pas triée
        mélanger aléatoirement les éléments de la liste
```

Attention lors de vos tests ! Il se peut, en particulier pour des listes ayant beaucoup d'éléments, qu'il ne se termine jamais, même si théoriquement le temps d'exécution est supposé fini.

C'est le tri le moins efficace de notre sélection. Testez le mais avec prudence !

Question 2. le tri par insertion

Le tri part insertion est ce que beaucoup de personnes font lorsqu'on leur donne un paquet de cartes à trier.
Prendre la première carte non triée du paquet et l'insérer à sa place dans la partie déjà triée du paquet de cartes. Vous pouvez trouver une animation illustrant de ce tri sur la page Wikipedia dédiée à cet algorithme.
Ci-dessous la description en pseudo-code du principe de cet algorithme sur un tableau issu de la page dédiée sur Wikipedia.
Les éléments du tableau T (de taille n) sont numérotés de 0 à n-1.

```
procédure tri_insertion(tableau T, entier n)
   pour i de 1 à n - 1
      # mémoriser T[i] dans x
      x ← T[i]
      # décaler vers la droite les éléments de T[0]..T[i-1] qui sont plus
grands que x (en partant de T[i-1])
      j ← i
      tant que j > 0 et T[j - 1] > x
          T[j] ← T[j - 1]
          j ← j - 1
```

```
      fin tant que
      # placer x dans le "trou" laissé par le décalage
      T[j] ← x
   fin pour
fin procédure
```

Avant d'aller plus loin essayez de comprendre cet algorithme sur l'exemple ci-dessous

```
my_lst_to_sort = [170, 45, 75, 90, 2, 24, 802, 66]
```

Écrivez une fonction `insertion_sort` qui prend en paramètre une liste `list_to_sort` de nombres et retourne une liste comportant les mêmes éléments mais triés, le tout sans toucher à la liste de départ.

Attention : Attention l'algorithme illustré par Wikipedia effectue le tri *en place* des éléments, dans notre cas, nous souhaitons que la liste de départ ne soit pas modifiée. Cela va nécessiter quelques ajustements de votre part.

```
# Testez votre code sur la liste exemple
print('La liste avant tri',my_lst_to_sort)
print('Le tri par sélection donne ',insertion_sort(my_lst_to_sort))
```

Question 3. le tri par sélection

Pour effectuer ce tri on recherche dans la partie de la liste non triée le `min` (ou le `max`, cela dépend si on souhaite trier les éléments de la liste par ordre croissant ou décroissant) et on place ce `min` à la fin de la partie triée de la liste résultat.
Ci-dessous la description en pseudo-code du principe de cet algorithme sur un tableau issu de la page dédiée sur Wikipedia.

```
procédure tri_selection(tableau t, entier n)
    pour i de 0 à n - 2
        min ← i
        pour j de i + 1 à n - 1
            si t[j] < t[min], alors min ← j
        fin pour
        si min ≠ i, alors échanger t[i] et t[min]
    fin pour
fin procédure
```

Essayez de comprendre et dérouler à la main l'algorithme sur la liste `my_lst_to_sort`.

Écrivez une fonction `selection_sort` qui prend en paramètre une liste de nombres `lst_to_sort` et retourne une liste comportant les mêmes éléments mais triés, le tout sans toucher à la liste de départ.

Point technique :

```
# Point sur le swap en Python
# l'échange de valeurs en python est différent de l'échange
# classique qu'on rencontre dans les autres langages
# Echange dans un langage "classique" comme C, Java, C++, etc.
a = 5
b = 6
print("Avant échange : a = ",a,"b = ",b)

# swap
temp = a
a = b
b = temp

print("Après échange : a = ",a, "b = ",b)
```

```
# En python il est possible de se passer de variable temporaire
c = 5
d = 6
print("Avant échange : c = ",c,"d = ",d)

c,d = d,c
print("Après échange : c = ",c, "d = ",d)
```

Une fois le code de `selection_sort` écrit testez-le.

```
# Testez votre code sur la liste exemple
print('La liste avant tri',my_lst_to_sort)
print('Le tri par sélection donne ',selection_sort(my_lst_to_sort))
```

Question 4. le tri à bulle

Le principe de cet algorithme est simple : l'algorithme parcourt le tableau et compare les éléments consécutifs. Lorsque deux éléments consécutifs ne sont pas dans l'ordre convenu (croissant ou décroissant selon ce que l'on souhaite obtenir), ils sont échangés. Après un premier parcours complet du tableau, le plus grand (respectivement petit) élément est forcément en fin de tableau, à sa position définitive. On recommence ainsi de suite jusqu'à ce que le tableau soit entièrement trié.
Ci-dessous la description en pseudo-code du principe de cet algorithme sur un tableau issu de la page dédiée sur Wikipedia. Le premier algorithme est non optimisé et le deuxième l'est.

```
tri_à_bulles(Tableau T)
   pour i allant de (taille de T)-1 à 1
       pour j allant de 0 à i-1
           si T[j+1] < T[j]
               échanger(T[j+1], T[j])
```

En optimisant le parcours à l'intérieur de la boucle interne :

```
tri_à_bulles_optimisé(Tableau T)
    pour i allant de (taille de T)-1 à 1
        tableau_trié := vrai
        pour j allant de 0 à i-1
            si T[j+1] < T[j]
                échanger(T[j+1], T[j])
                tableau_trié := faux
        si tableau_trié
            fin tri_à_bulles_optimisé
```

Essayez de comprendre et dérouler à la main l'algorithme sur la liste `my_lst_to_sort`.

Ecrivez une fonction `buble_sort` qui prend en paramètre une liste de nombres `lst_to_sort` et retourne une liste comportant les mêmes éléments mais triés.

```
#Test de votre code
print('Avant tri :',my_lst_to_sort)
print('Resultat du tri :', bubble_sort(my_lst_to_sort))
print('Après le tri la liste d\'origine n\'a pas été modifiée
:',my_lst_to_sort)
```

Question 5. le tri fusion

Le principe de cet algorithme est basé sur la technique, très classique en informatique, du *diviser pour régner*. Il fait appel également à une notion que nous n'avons pas encore abordée dans ces ateliers celle de la récursivité, c'est à dire une fonction qui s'appelle elle-même.

L'idée est qu'à partir de deux listes triées on peut facilement construire, en les fusionnant, une liste triée comportant les éléments des deux listes de départ.
Ci-dessous la description en langage naturel puis en pseudo-code du principe de cet algorithme issu de la page dédiée sur Wikipedia.

L'algorithme est naturellement décrit de façon récursive (notez que vous pouvez le transformer en version itérative pour ne pas être limité par la profondeur limitée en python des appels récursifs) :

- Si le tableau n'a qu'un élément, il est déjà trié.
- Sinon, séparer le tableau en deux parties à peu près égales.
- Trier récursivement les deux parties avec l'algorithme du tri fusion.
- Fusionner les deux tableaux triés en un seul tableau trié.

En pseudo code :

```
fonction triFusion(T[1, …, n])
# entrée : un tableau T
# sortie : une permutation triée de T
      si n ≤ 1
              renvoyer T
      sinon
              renvoyer fusion(triFusion(T[1, …, n/2]), triFusion(T[n/2 + 1,
…, n]))
fonction fusion(A[1, …, a], B[1, …, b])
#entrée : deux tableaux triés A et B
#sortie : un tableau trié qui contient exactement les éléments des tableaux
A et B
      si A est le tableau vide
              renvoyer B
      si B est le tableau vide
              renvoyer A
      si A[1] ≤ B[1]
              renvoyer A[1] :: fusion(A[2, …, a], B)
      sinon
              renvoyer B[1] :: fusion(A, B[2, …, b])
```

Essayez de comprendre et dérouler à la main l'algorithme sur la liste `my_lst_to_sort`.

Ecrivez une fonction `merge_sort` qui prend en paramètre une liste `list_to_sort` d'entiers, qui retourne une liste composée des éléments de la liste de départ triés dans l'ordre croissant. La liste de départ ne doit pas être modifiée. Vous allez également, en accord avec l'algorithme ci-dessus être amené à écrire une fonction `merge` prenant en paramètre deux listes triées et qui retourne la liste triée résultant de la fusion des deux listes de départ.

```
print('Avant tri :',my_lst_to_sort)
print('Resultat du tri :', merge_sort(my_lst_to_sort))
print('Après le tri la liste d\'origine n\'a pas été modifiée
:',my_lst_to_sort)
```

Question 6. le tri par base

Ce tri est basé sur un tri lexicographique. On effectue en premier lieu un tri suivant le chiffre le moins significatif en conservant l'ordre de base pour les éléments ayant le même chiffre. On répète le même principe en prenant le chiffre plus significatif.

Ci-dessous un exemple tiré de la page Wikipedia dédiée au tri par base :

Trier la liste : 170, 45, 75, 90, 2, 24, 802, 66

1. tri par le chiffre le moins significatif (unités) : 170, 90, 2, 802, 24, 45, 75, 66
2. tri par le chiffre suivant (dizaines) : 2, 802, 24, 45, 66, 170, 75, 90
3. tri par le chiffre le plus significatif (centaines) : 2, 24, 45, 66, 75, 90, 170, 802

Écrivez une fonction `radix_sort` qui prend en paramètre une liste `lst_to_sort` d'entiers, qui retourne une liste composée des éléments de la liste de départ triés dans l'ordre croissant. Nous vous recommandons de créer également une fonction `radix_order_sort` qui prend en paramètre la liste à trier `lst_to_sort`, ainsi que l'ordre auquel le tri doit s'effectuer (sur l'exemple précédent, le tri en 1. s'effectue à l'ordre 0 = 10^0, le tri en 2. s'effectue à l'ordre 1 = 10^1, le tri en 3. s'effectue à l'ordre 2 = 10^2. La liste de départ ne doit pas être modifiée.

Test de votre code :

```
print('Avant tri :',my_lst_to_sort)
print('Resultat du tri :', radix_sort(my_lst_to_sort))
print('Les étapes suivantes ont été passées :')
lst_sorted_order0 = radix_order_sort(my_lst_to_sort,0)
print('Après tri à l\'ordre 0 :',lst_sorted_order0)
lst_sorted_order1 = radix_order_sort(lst_sorted_order0,1)
print('Après tri à l\'ordre 1 :',lst_sorted_order1)
print('Après tri à l\'ordre 2 :',radix_order_sort(lst_sorted_order1,2))
print('Après le tri la liste d\'origine n\'a pas été modifiée
:',my_lst_to_sort)
```

PARTIE 3 - EXERCICES BONUS

EXERCICE 8 - Découverte expérimentale de la complexité de sorted

Vous allez ici comparer les temps d'exécution des différents algorithmes de tri que vous avez implémentés en superposant leur courbe à celles de la fonction `sorted` de python. Pour cela vous allez utiliser le module `time` qui permet de récupérer le temps système (cf. Partie 2 Exercice 5). La mesure du temps système peut s'effectuer avec la methode `perf_counter()`. Vous veillerez également à tracer les courbes des temps d'exécution en fonction de la taille des listes pour la liste de valeurs [10,500,1000,5000,10000,50000,80000,100000] (si votre machine le permet !). Rappelez-vous que vous devez considérer des moyennes de temps d'exécution pour faire un travail sérieux.

A minima nous vous demandons d'afficher les courbes dans le cas moyen. C'est à dire pour des données tirées aléatoirement.

Dans l'idéal il faudrait faire trois graphiques différents celui où nous trouvons

- dans le meilleur des cas, à savoir la liste est déjà triée,
- dans le cas moyen, la liste a été aléatoirement constituée,
- dans le pire des cas, la liste est triée dans l'ordre inverse de celui souhaité.