I. Mesure d'un temps d'exécution

1 - Utilisation de time()

Pour évaluer la complexité temporelle, il est possible de mesurer le temps d'exécution. La fonction`time()` permet de mesurer le temps en seconde pris par l'exécution d'un processus. Le code suivant permet de mesurer le temps passé pour effectuer la fonction proposée.

```
from time import time
def reponse():
    42**42
# main
t0 = time()
somme()
t1 = time()- t0
print(" temps d'execution : {:.2f} s".format(t1) )
```

Ex. 1 :

Q 1 - Définir une fonction `gen_liste(N)` qui renvoie une liste de N termes pris aléatoirement entre 0 et 1. On pourra utiliser la fonction `random()` de la bibliothèque random

Q 2 - Mesurer le temps d'exécution pour générer une liste de 200 éléments aléatoires

Q 3 - Le temps d'exécution étant trop court, proposer un code permettant de mesurer la génération de 1000 listes et d'afficher le temps moyen pour la création d'une liste :

```
temps de creation : 0.016 ms
```

Q 4 - En déduire le temps moyen d'affectation et de création d'une variable aléatoire.

2 - Optimisation d'algorithme

Ex. 2 :

Q 1 - Définir une fonction `moy` renvoyant la moyenne des éléments d'une liste `L`.

Q 2 - On considère les deux fonctions suivantes permettant de calculer la variance :

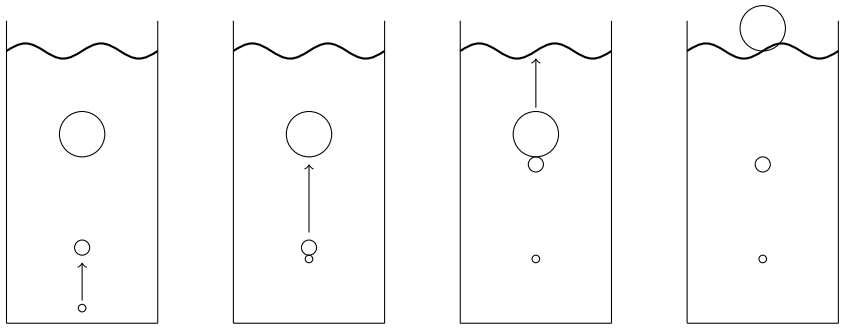
```
def var1(L):
    s = 0
    for x in L :
        s += x**2 - moy(L)**2
    return s/len(L)
```

```
def var2(L):
    s = 0
    m = moy(L)
    for x in L :
        s += x**2 - m**2
    return s/len(L)
```

Mesurer le temps d'exécution moyen pour 1000 calculs de variance dans chacun des cas. Justifier que `var2` est beaucoup plus rapide que `var1`.

3 - Application au tri à bulles

Dans un casserole d'eau bouillante on peut imaginer que les bulles de vapeur d'eau remontent à la surface en partant du fond de la façon suivante : lorsqu'une bulle rencontre une bulle plus grosse qu'elle, elle s'arrête et c'est la bulle plus grosse qui poursuit la course vers l'air libre jusqu'à ce qu'elle rencontre à son tour une bulle plus grosse. C'est exactement le principe du tri bulle.



Considérons un tableau `[T[0],...,T[n-1]]` à n éléments.

- On parcourt le tableau à partir de `T[0]`. On compare `T[0]` à `T[1]` puis `T[1]` à `T[2]` et ainsi de suite jusqu'à ce que l'on rencontre deux termes consécutifs qui sont dans le désordre. On échange alors ces deux éléments.

- On continue le parcours jusqu'à arriver à la fin du tableau. Il est clair qu'à l'issue de ce processus la valeur `T[n-1]` est la plus grande. Il reste donc à réitérer le processus sur le tableau `[T[0],...,T[n-2]]`.

Ex. 3 :

Afin de comprendre le principe, décrire au brouillon les étapes du tri bulle de la liste `[18,4,13,2,20]`.

Ex. 4 :

Q 1 - Proposer une fonction `est_trie(L)` qui renvoie un booléen selon que la liste `L` soit triée ou non. On vérifiera que :

```
est_trie([0,2,4,5]) # True
est_trie([0,2,4,5,1]) # False
```

Q 2 - Proposer une fonction `tri_bulle(T)` qui modifie la liste `T` en la triant dans l'ordre croissant par tri bulle. On vérifiera que la liste `L=[18,4,13,2,20]` devient `[2,4,13,18,20]`.

Remarque. `T` étant une liste, la fonction `tri_bulle` va bien modifier cette liste (cf. cours sur les fonctions). On parle de mutation en place.

4 - Validation par jeux de tests

Pour vérifier la correction de notre algorithme, on génère aléatoirement une liste à 10 éléments pris au hasard entre 1 et 99.

```
from random import *
liste=[]
for i in range(10):
    liste.append(randint(1,100))
print('une liste :', liste)
tri_bulle(liste)
print('la meme triee : ',liste)
```

Vérifiez que votre algorithme fonctionne sur quelques exemples.

Remarque. On aurait aussi pu définir la liste en compréhension en écrivant :

```
liste=[randint(1,100) for i in range(10)]
```

5 - Complexité

Ex. 5 :

Supposons que la liste `T` contient n éléments.

Q 1 - Combien de comparaisons la fonction `tri_bulle` fait-elle ?

Q 2 - Il est bien clair que le nombre d'échanges dépend de l'ordre des éléments de la liste initiale.

1. Dans le meilleur des cas, si le tableau est déjà parfaitement trié, combien d'échanges y a-t-il ?
2. Le pire des cas est celui où le tableau est trié dans l'ordre décroissant. Dans ce cas, combien y a-t-il d'échanges.

Quel est le coût minimal ? maximal ?

Pour mesurer l'efficacité du tri bulle on peut aussi produire un très grand nombre de tests (disons 10000) et mesurer le temps écouler pour trier toutes les listes.

```
from time import time
debut=time()
for j in range(10000):
    liste=[randint(1,100) for i in range(10)]
    tri_bulle(liste)
fin=time()
print("temps ecoule avec tri_bulle : ",fin-debut)

debut=time()
for j in range(10000):
    liste=[randint(1,100) for i in range(10)]
    liste.sort()
fin=time()
print("temps ecoule avec sort : ",fin-debut)
```

Que dire de l'efficacité du tri bulle par rapport à la méthode `sort` ?

Rermarque. La méthode `sort` utilise une méthode de tri appelée Timsort mise au point par Tim Peters en 2002 pour Python. C'est un tri évolué basé sur le tri fusion. Sa complexité est de l'ordre de $n\ln(n)$.

6 - Pour aller plus loin

Ex. 6 :

On considère une liste `L` de tuples représentant des points du plan définis par leurs coordoonées :

```
L=[(x0,y0),(x1,y1),...]
```

1. Écrire une fonction `distance(P)` qui à un point P associe sa distance à l'origine.
2. Écrire une fonction `tri_points(L)` qui trie la liste selon les distances de chacun des points à l'origine du plus proche au plus éloigné.

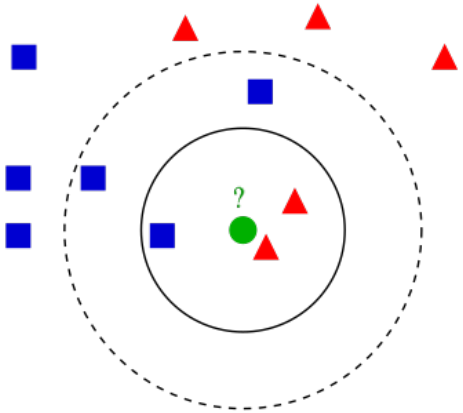
Ex. 7 :

On suppose que `L` est une liste d'entiers compris entre 0 et $n-1$. Pour trier cette liste, il suffit de compter les occurrences de chacun des entiers compris entre 0 et $n-1$ dans la liste puis de placer les entiers de 0 à $n-1$ en respectant les nombres d'occurrences. On parle de `tri par dénombrement`.

1. Écrire une fonction `occurrences(L,n)` qui associe à une liste `L` d'entiers compris entre 0 et $n-1$ une liste `occ` telle que pour tout i, `occ[i]` soit le nombre d'occurrences de i dans `L`.
2. Écrire une fonction `tri_denombrement(L,n)` qui trie `L` par dénombrement.
3. Quel est le coût de cette méthode ?

Méthode KNN partie 1

En intelligence artificielle, plus précisément en apprentissage automatique, la méthode des k plus proches voisins est une méthode d'apprentissage supervisé. En abrégé KPPV ou k-PPV en français, ou plus fréquemment k-NN ou KNN, de l'anglais k-nearest neighbors.



Le but est de classer des points du plan en fonction des voisins. Dans la figure ci-contre, l'échantillon de test (point vert) pourrait être classé soit dans la classe de carré bleu ou la seconde classe de triangles rouges. Si k = 3 (cercle en ligne pleine) il est affecté à la classe des triangles car il y a deux triangles et seulement un carré dans le cercle considéré. Si k = 5 (cercle en ligne pointillée) il est affecté à la classe des carrés (3 carrés face à deux triangles dans le cercle externe)

7 - Création d'une liste de villes d'une même famille

Dans cette partie, nous allons générer une liste `L` de points du plan définis par leurs coordonnées. `L` est donc une liste de tuples de la forme `[(x0,y0),(x1,y1),...]`.

On se place dans le plan Oxy, dans un espace de taille $L \times L$ avec `L = 10`.

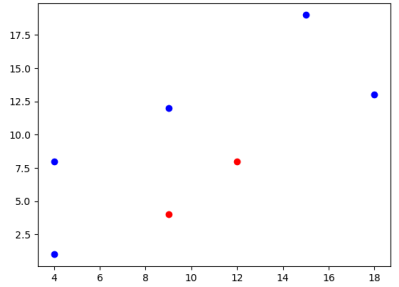
Ex. 8 :

On rappelle que la fonction `random()` du module `random` fourni un nombre entre 0 et 1. Proposer une fonction `coord_alea(n)` qui renvoie une liste de `n` tuples de coordonnées (x_i,y_i) dans l'espace de taille $L \times L$.

Pour visualiser ces points, on pourra utiliser la fonction `plot` du module `pyplot`. Les lignes suivantes permettent d'afficher les points par des ronds 'o' en bleu 'b'.

```
import matplotlib.pyplot as plt
L = coord_alea(5)
for P in L :
    plt.plot(P[0],P[1],'ob')
L2 = coord_alea(2)
for P in L2 :
    plt.plot(P[0],P[1],'or')
plt.show()
```

Le choix de la couleur ou de la forme peut être modifié. Parmi les possibilités : 'x' pour des croix, 'v' pour des triangles, 's' pour des carrés... Les couleurs simples sont : 'r' pour rouge, 'y' pour jaune... bref penser à Dora l'exploratrice pour trouver une couleur. Par exemple : 'ps' donnera des carrés violets.



Ex. 9 :

On désire vérifier que tous les points de `L` sont distincts. Proposer une fonction `sont_distincts(L)` qui renvoie un booléen selon que les différents points de `L` sont distincts ou non. On vérifiera la fonction avec les lignes suivantes :

```
>>>L = [(1,0),(0,1),0,1)]
>>> sont_distincts(L)
False
>>>L = [(1,0),(0,1),0,2)]
>>> sont_distincts(L)
True
```

8 - Recherche d'éléments proche d'un point P

On cherche à écrire des algorithmes de recherche d'éléments dans `L` satisfaisant une propriété de proximité.

Ex. 10 :

Proposer une fonction `dist` qui renvoie la distance entre deux points `P` et `Q`.

Le test suivant doit être validé :

```
>>> dist([0,1] ,[1,0] )
1.4142135
```

Parmi les points d'une liste L, pour trouver le point le plus proche d'un point `P`, il faut calculer la liste des distances possibles des points de `L` par rapport à `P`.

Ex. 11 :

Proposer une fonction `liste_distance(L,P)` qui renvoie une liste des distances de chaque point de `L` par rappor t `P`.

Le test suivant doit être validé :

```
>>>L=[ (0,1), (3,0) , (0,2) , (1,1)]
>>>P =(0,0)
>>>liste_distance(L,P)
[1,3,2,1.4414]
```

Ex. 12 :

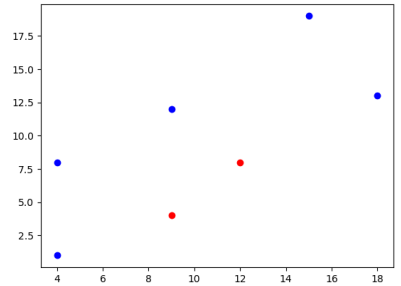
On cherche à écrire une fonction `meilleur_voisin(L,P)` qui prend comme paramètres `L` et `P` et qui renvoie le point le plus proche de `P` Il s'agit donc de trouver le point qui minimise la distance entre les éléments de `L` et `P`. Écrire cette fonction en vous inspirant de l'algorithme de recherche du minimum d'une liste.

Le test suivant doit être validé :

```
>>>L = [ (3,0), (1,0) , (0,2) , (1,1)]
>>>P=[0,0]
>>>meilleur_voisin(L,P)
(1, 0)
```

Ex. 13 :

Proposer un code permettant d'afficher en rouge le point `P` et son plus proche voisin issu de `L`.



9 - Recherche des deux éléments les plus proches

Ex. 14 :

Q 1 - Proposer une écriture de la fonction `plus_proche_voisins` qui renvoie deux points de `L` dont la distance est minimale en utilisant les fonctions précédentes pour tester toutes les paires.

Q 2 - On suppose que `L` contient n points. Combien y a-t-il de paires testées ? Quel est la complexité de l'algorithme ?