## Exercice 1 :

Combien de fois la fonction *print* est-elle appelée dans le code suivant :

```
for i in range(5):
    for j in range(i+1,5):
        print(i+j)
```

| | | |
|---|---|---|
| $i = 0$ | $j = 1,2,3,4$ | $i + j = 1,2,3,4$ |
| $i = 1$ | $j = 2,3,4$ | $i + j = 3,4,5$ |
| $i = 2$ | $j = 3,4$ | $i + j = 5,6$ |
| $i = 3$ | $j = 4$ | $i + j = 7$ |
| $i = 4$ | | |

Ce qui conduit à 10 valeurs

## Exercice2 :

Pour chacun des 3 scripts, déterminer le nombre d'additions effectuées. Exprimer ce nombre en fonction de $n$ pour les 2 derniers.

Script 1 :

```
x=0
for i in range(2):
    x=x+1
    for j in range(3):
        x=x+j
```

Script2 :

```
x=0
for i in range(2):
    x=x+1
    for j in range(n):
        x=x+j
```

Script 3 :

```
x=0
for i in range(n):
    x=x+1
    for j in range(n):
        x=x+j
```

- script 1 :

| | | |
|---|---|---|
| $i = 0$ | $j = 0,1,2$ | 4 additions |
| $i = 1$ | $j = 0,1,2$ | 4 additions |

8 additions au total

- script 2 :

On a 2*(1+n) car la boucle externe exécute 1+n additions (et elle est exécutée 2 fois)

- script 3 :

On a $n(1 + n)$ additions car la boucle externe exécute 1+n additions (et elle est exécuté n fois)

## Exercice 3 :

Après le code python qui suit, quelles sont les valeurs finales de $x$ et $y$ ?

```
x=4
while x>0:
    y=0
    while y<x:
        y=y+1
        x=x-1
```

| | | |
|---|---|---|
| Rentré dans la boucle externe | En entrée : $x = 4$ $y = 0$ | |
| 1e passage dans la boucle interne : | En entrée : $x = 4$ $y = 0$ | En sortie : $x = 3$ $y = 1$ |
| 2e passage dans la boucle interne | En entrée : $x = 3$ $y = 1$ | En sortie : $x = 2$ $y = 2$ |
| 3e passage dans la boucle interne | En entrée : $x = 2$ $y = 2$ | Fin de boucle interne |
| 4e passage dans la boucle interne | En entrée : $x = 2$ $y = 0$ | En sortie : $x = 1$ $y = 1$ |
| 5e passage dans la boucle interne | En entrée : $x = 1$ $y = 1$ | Fin de boucle interne |
| 6e passage dans la boucle interne | En entrée : $x = 1$ $y = 0$ | En sortie : $x = 0$ $y = 1$ |
| Fin de la boucle interne puis externe | | |

## Exercice 4 :

Voici la définition d'une fonction qui prend en paramètre une liste de nombres.

```
def remonte(liste):
    for i in range(len(liste)-1):
        if liste[i]>liste[i+1]:
            liste[i],liste[i+1]=liste[i+1],liste[i]
```

On appelle la fonction *remonte* avec une liste de longueur $n$. Quelle affirmation est vraie ?

1. Dans la liste modifiée, le plus petit élément est placé au début.
2. Dans la liste modifiée, le plus grand élément est placé à la fin.
3. Le nombre de comparaisons effectuées est exactement égale à $n$.
4. Le nombre d'échanges effectués est strictement inférieur à $n - 1$.

1.Non, pas forcément, c'est le début d'un tri à bulles qui remonte la plus grande valeur à la fin de la liste.

2. Oui

3. Non, $n - 1$ comparaisons

4. Non, on peut avoir jusqu'à $n - 1$ échanges dans le cas d'une liste classée de manière décroissante

## Exercice 5 :

On reprend la fonction *remonte* définie dans l'exercice précédent. On définit une liste *nombres* = [12,5,13,8,11,6]. Si on appelle la fonction *remonte* avec en paramètre la liste *nombres*, quel est l'état final de la liste ?

[12,5,13,8,11,6] → [5,12,13,8,11,6] → [5,12,8,13,11,6]

[5,12,8,11,13,6] → [5,12,8,11,6,13]

## Exercice 6 :

En adaptant l'algorithme de tri à bulles vu en cours, écrire une fonction *ordre* qui prend en argument une liste de mots et modifie la liste en ordonnant les mots en fonction du nombre de lettres. Tester la fonction avec la liste `["toto","bonjour","a","oui","non"]`.

```
def tri_bulles(liste):
    j=len(liste)-1
    while j>0:
        modification=True
        for i in range(j):
            if len(liste[i])>len(liste[i+1]):
                liste[i],liste[i+1]=liste[i+1],liste[i]
                modification=False
        j=j-1
        if modification :
            j=0
    return liste

liste=["toto","bonjour","a","oui","non"]
print(tri_bulles(liste))
```

## Exercice 7 :

L'objectif est d'apprécier les temps d'exécution du tri à bulles sur une liste de nombres au hasard.

1) Construire une fonction qui retourne une liste de longueur $n$ comportant des entiers triés au hasard compris entre 1 et $n$ (inclus).
2) Expliquer l'évolution du temps de tri de l'algorithme de tri à bulles lorsque $n$ passe de 5000 à 10000.

```
def liste_hasard(n):
    liste=[i for i in range(1,n+1)]
    random.shuffle(liste)
    return liste
def liste_hasard2(n):
    return [random.randint(0,n) for i in range(n)]

def tri_bulles(liste):
    j=len(liste)-1
    while j>0:
        modification=True
        for i in range(j):
            if liste[i]>liste[i+1]:
                liste[i],liste[i+1]=liste[i+1],liste[i]
                modification=False
        j=j-1
        if modification :
            j=0
    return liste
liste_n=[5000,10000]
liste_temps_triee=[]
liste_temps_hasard=[]
for n in liste_n:
    liste=liste_hasard2(n)
    t0=time.time()
    tri_bulles(liste)
    liste_temps_hasard.append(time.time()-t0)


print(liste_temps_hasard)
```

Doubler la longueur de liste, revient bien à quadrupler le temps de calcul

## Exercice 8 :

On dispose de points dans le plan muni d'un repère orthonormé d'origine $O$. Chaque point possède un couple de coordonnées $(x;y)$ représenté par la liste $[x,y]$. Il s'agit de trier ces points en fonction de leur distance à $O$, de la plus petite à la plus grande.

1) Ecrire une fonction *distance* qui prend en paramètre une liste de deux nombres nommée *point* qui représente un point du plan et renvoie le carré de la distance euclidienne entre ce point et $O$.
2) Ecrire une fonction *compare* qui prend en paramètre deux listes $p1$ et $p2$ représentant deux points $P_1$ et $P_2$ et qui renvoie $-1$ si $P_1$ est plus proche de $O$ que $P_2$, 1 si $P_2$ est plus proche de $O$ que $P_1$, et 0 si les deux points sont équidistants de $O$.
3) Ecrire une fonction *tri_points* qui prend en paramètre une liste composée de listes de deux nombres représentant des points dans le plan et qui trie (avec un tri à bulles) cette liste suivant les distances entre chacun des points et $O$.

```
def distance(point):
    return (point[0]**2+point[1]**2)
def compare(p1,p2):
    d1,d2=distance(p1),distance(p2)
    if d1<d2:
        return -1
    elif d1>d2:
        return +1
    else :
        return 0
def tri(liste):
    j=len(liste)-1
    while j>0:
        permutation=False
        for i in range(j):
            if compare(liste[i],liste[i+1])==1:
                liste[i],liste[i+1]=liste[i+1],liste[i]
                permutation=True
        j=j-1
        if permutation==False:
            j=0
    return liste
```

## Exercice 9 :

On rappelle l'algorithme de recherche textuelle du cours :

```
def recherche(chaine,mot):
    n=len(chaine)
    m=len(mot)
    for i in range(n-m+1):
        j=0
        while j<m and chaine[i+j]==mot[j]:
            j=j+1
        if j==m:
            return print("mot est à l'index {}".format(i))
    print("le mot n'est pas dans la liste")
```

1) On considère le texte "*abababababab*" et le mot "*abc*".Déterminer le nombre de comparaisons effectuées par la fonction *recherche* avec ce texte et ce mot à chercher.
2) Déterminer le nombre de comparaisons si le texte "abababa..." contient 100 caractères (50 fois *ab*).
3) Déterminer le nombre de comparaison en fonction de $n$ si le texte contient $n$ caractères sur le même modèle.

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| a | b | a | b | a | b | a | b | a | b | |
| a | b | c | | | | | | | | 3 |
| | a | b | c | | | | | | | 1 |
| | | a | b | c | | | | | | 3 |
| | | | a | b | c | | | | | 1 |
| | | | | a | b | c | | | | 3 |
| | | | | | a | b | c | | | 1 |
| | | | | | | a | b | c | | 3 |
| | | | | | | | a | b | c | 1 |

La dernière valeur de $i$ est 7 (donc 8 valeurs) : (3+1+3+1+3+1+3+1) =4*3+4*1=16 comparaisons (4 cas avec 3 comparaisons, 4 cas avec une comparaison). Pour un texte de 100 caractères : la dernière valeur de $i$ est 97, soient : 49*1+49*3 = 49 + 147 = 196 comparaisons

Plus généralement, on peut remarquer que l'on fait $\frac{(n-2)*3+(n-2)}{2} = 2(n-2)$

## Exercice 10 :

On souhaite modifier l'algorithme naïf de recherche textuelle de l'exercice précédent de manière à ce que le motif soit lu de droite à gauche à partir du dernier caractère. Proposer un programme

```
def recherche(chaine,mot):
    n=len(chaine)
    m=len(mot)
    for i in range(n-m+1):
        j=m-1
        while j>=0 and chaine[i+j]==mot[j]:
            j=j-1
        if j==-1:
            return print("mot est à l'index {}".format(i))
    print("le mot n'est pas dans la liste")
```

Si on cherche "$abc$" dans "$ababc$" :

| | |
|---|---|
| $i = j = 2$ | Conditions while non vérifiées |
| $i = 3, j = 2$ | Conditions while non vérifiées |
| $i = 4, j = 2$ | $i = 3, j = 1 \rightarrow i = 2, j = 0 \rightarrow i = 1, j = -1$ |

.