Exercice 1 :

On donne la version itérative de la fonction « pair » qui permet de dire si un entier est pair ou non.
Écrire une version récursive de cette fonction.

```
def pair(n):
    """
    n: int --> bool
    pair renvoie True si l'entier n est pair, et False s'il est impair
    """
    if n < 0:
        n = -1*n
    while n > 0:
        n = n - 2
    return n == 0   # Retourne True si c'est le cas, False sinon
```

Exercice 2 : *Les seules opérations autorisées dans cet exercice sont : ajouter 1 ou retrancher 1.*

1) La fonction « somme » renvoie la somme des entiers naturels a et b.
On donne la version itérative de cette fonction :

```
# version itérative
def somme(a, b):
    """
    a, b : int positifs --> int positif
    renvoie a+b avec a et b entiers positifs
    """
    while b > 0:
        a = a + 1
        b = b - 1
    return a
```

Écrire une version récursive de cette fonction en tenant compte des contraintes imposées sur les opérations autorisées.

2) Écrire une fonction qui renvoie la somme de 2 entiers relatifs en tenant toujours compte des contraintes fixées.

Exercice 3 : Une fonction mathématique très connue

La fonction « factorielle » est définie pour tout entier n par :

$$factorielle(n) = \begin{cases} n \times (n-1) \times (n-2) \ldots \times 2 \times 1 = n! & si\ n > 0 \\ 1 = 0! & si\ n = 0 \end{cases}$$; On note $\boldsymbol{factorielle(n) = n!}$

Écrire une version itérative puis récursive de cette fonction.

Exercice 4 : Analyser un programme

On donne la fonction « mystere » ci-dessous :

```
def mystere(n):
    '''n est un entier positif ou nul'''
    if n < 2:
        return str(n)
    else:
        return mystere(n // 2) + str(n % 2)
```

Réaliser l'arbre d'appel de mystere(4) puis de mystere(5). (Ecrire en commentaires ou mettre une image dans le dossier avec le nom de l'exercice)
En déduire ce que fait la fonction « mystere ».

Exercice 5 :

1) Écrire une fonction récursive « somme » qui prend en paramètre une liste de nombres et renvoie la somme des éléments de cette liste.

2) Réaliser l'arbre d'appel pour somme([4, 7, 2]).

Exercice 6 : Avec du graphisme

Tester le programme ci-dessous :

```
from turtle import *

def dessin():
    for i in range(50):
        color(couleurs[i % 6])
        forward(i)
        right(59)

couleurs = ['blue', 'green', 'yellow', 'orange', 'red', 'purple']
bgcolor('black')

dessin()
exitonclick()
```

1) Expliquer pourquoi la fonction « dessin » n'est pas « pure ». La transformer en une fonction pure.

2) Écrire votre fonction de manière récursive.

Exercice 7 bonus :

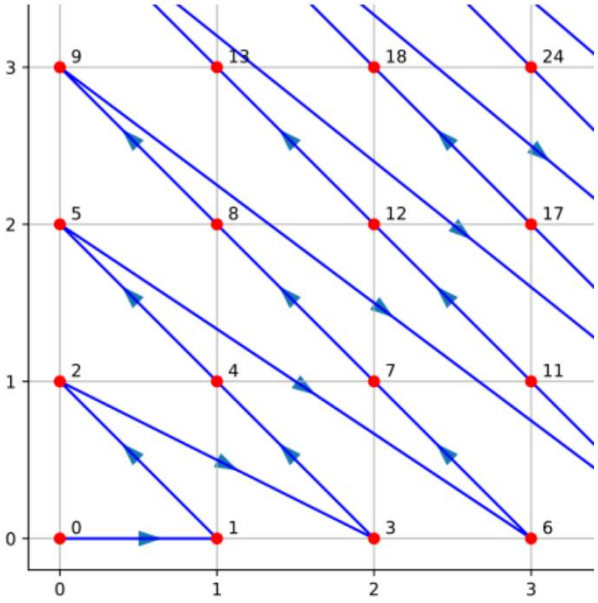
On donne une fonction itérative « nettoie » qui prend en paramètre une liste triée et élimine les éléments identiques de cette liste.
Par exemple : nettoie([1, 1, 2, 6, 6, 6, 8, 8, 9, 10]) renvoie [1, 2, 6, 8, 9,10]

```
def nettoie(L):
    k = 0
    while k < len(L) - 1:
        if L[k] != L[k+1]:
            k = k + 1
        else:
            L.pop(k)
    return L
```

Écrire une version récursive de cette fonction.

Exercice 8 bonus : la Diagonale de Cantor

Dans un repère, les points à coordonnées entières sont numérotés selon le principe suivant :



Écrire une fonction récursive « numéro » qui prend en paramètres les coordonnées *x* et *y* d'un point et qui renvoie le numéro du point selon le principe de la diagonale de Cantor.
Par exemple numero(1,3) renvoie 13.