Exercice 0 : Tracé alphanumérique

1) Analyser les deux fonctions ci-dessous et anticiper l'exécution de leur résultat pour $n = 4$:

```
def triangle1(n):
    if n ==0:
        return None
    else :
        print("*"*n)
        triangle1(n-1)
```

```
def triangle2(n):
    if n ==0:
        return None
    else :
        triangle2(n-1)
        print("*"*n)
```

2) Analyser les deux fonctions ci-dessous et anticiper l'exécution de leur résultat pour $n = 4$.

```
def triangles1(n):
    if n==0:
        return None
    else :
        print("*"*n)
        triangles1(n-1)
        print("*"*n)
```

```
def triangles2(n):
    if n==0:
        return None
    else :
        triangles2(n-1)
        print("*"*n)
        triangles2(n-1)
```

Dans les deux cas l'arbre des appels récursifs de limite à une branche

$triangle1(4) =>****$	$triangle2(4) => Rien$
$triangle1(3) =>***$	$triangle2(3) => Rien$
$triangle1(2) =>**$	$triangle2(2) => Rien$
$triangle1(1) =>*$	$triangle2(1) => Rien$
$triangle1(0) => Rien$	$triangle2(0) => Rien$
$triangle1(1) => Rien$	$triangle2(1) =>*$
$triangle1(2) => Rien$	$triangle2(2) =>**$
$triangle1(3) => Rien$	$triangle2(3) =>***$
$triangle1(4) => Rien$	$triangle2(4) =>****$

Pour la fonction $triangles1$, lorsque l'on dépile, il reste un print à prendre en considération

$triangles1(4) =>****$	$print(" * " * 4)$ en attente
$triangles1(3) =>***$	$print(" * " * 3)$ en attente
$triangles1(2) =>**$	$print(" * " * 2)$ en attente
$triangles1(1) =>*$	$print(" * " * 1)$ en attente
$triangles1(0) => Rien$	
$triangles1(1) =>*$	
$triangles1(2) =>**$	
$triangles1(3) =>***$	
$triangles1(4) =>****$	

Pour $triangles2$, l'arbre des appels est à plusieurs branches

$triangles2(4)$	En attente : $print(" * " * 4)$ $triangles2(3)$	
$triangles2(3)$	En attente : $print(" * " * 3)$ $triangles2(2)$	
$triangles2(2)$	En attente : $print(" * " * 2)$ $triangles2(1)$	
$triangles2(1)$	En attente : $print(" * " * 1)$ $triangles2(0)$	
$triangles2(0) => dépile$		

$print(* * 1) =>*$ $triangles2(0) => Rien$		*
$print(* * 2) =>**$ $triangles2(1) =>*$		** *
$print(* * 3) =>***$ $triangles2(2)$ Ce qui revient à faire $triangles2(1)$ $print(" * " * 2)$ $triangles2(1)$		*** * ** *
$print(* * 4) =>****$ $triangles2(3)$ Ce qui revient à faire $triangles2(2)$ $print(" * " * 3)$ $triangles2(2)$		**** * ** * *** * ** *

Exercice 1 : calcul de la factorielle

1) Proposer un algorithme itératif calculant la factorielle de n. On définira une fonction $fac\_It()$ prenant l'entier n en argument.
2) Proposer un algorithme récursif calculant la factorielle de n. On définira une fonction $fac\_Rec()$ prenant l'entier n en argument.

```
def fac_It(n):
    resultat=1
    for i in range(1,n+1,1):
        resultat=resultat*i
    return resultat
print(fac_It(4))

def fac_It2(n):
    resultat=1
    i=n
    while i>1:
        resultat=resultat*i
        i=i-1
    return resultat
"""question 2"""
def fac_Rec(n):
    if n==0 or n==1:
        return 1
    else :
        return n*fac_Rec(n-1)
print(fac_Rec(4))
```

La complexité est linéaire en temps et en espace.

Exercice 2 : calcul de 2^n

1) Proposer un algorithme itératif calculant 2^n avec n entier. On définira une fonction $f\_it()$ prenant l'entier n en argument.
2) Proposer un algorithme récursif calculant 2^n avec n entier. On définira une fonction $f\_rec()$ prenant l'entier n en argument.

```
"""question 1"""
def f_it(n):
    resultat = 1
    for i in range(1,n+1,1):
        resultat=resultat*2
    return resultat
print(f_it(4))

def f_it2(n):
    resultat=1
    i=n
    while i>0 :
        resultat=resultat*2
        i=i-1
    return resultat

"""question 2"""

def f_rec(n):
    if n==0:
        return 1
    else :
        return 2*f_rec(n-1)
print(f_rec(4))
#complexité T(n)=O(n)
def f_rec2(n):
    if n==0 :
        return 1
    else :
        if n%2 == 0 :
            return f_rec2(n//2)**2
        else :
            return 2*f_rec2(n//2)**2
print(f_rec2(3))
#T(100)=T(50)+1=1+(T(25)+2)=3+T(12)+2=5+T(6)+1=6+T(3)+1=7+T(1
)+2=9+T(0)+2=12
#Grossièrement T(n)=log2(n)
```

Exercice 3 : Calcul d'une suite

On considère la suite suivante :

$$\begin{cases} u_0 = 1 \\ u_{n+1} = 10 - 2u_n \end{cases}$$

1) Calculer la valeur du terme de rang n en utilisant une instruction itérative.
2) Calculer la valeur du terme de rang n en utilisant une instruction récursive.

```
def suite(n):
    u0=1
    i=n
    while i >0:
        u1=10-2*u0
        u0=u1
        i=i-1
    return u1
def suite2(n):
    if n==0:
        return 1
    else :
        return 10-2*suite2(n-1)
```

Exercice 4 : Suite de Fibonacci

Cette suite est définie par récurrence telle que :

$$u_0 = 1$$

$$u_1 = 1$$

Et pour $n \geq 2$:

$$u_n = u_{n-1} + u_{n-2}$$

On obtient alors la suite suivante : 1, 1, 2, 3, 5, 8, 13

1) Calculer la valeur du terme de rang n en utilisant une instruction itérative.
2) Calculer la valeur du terme de rang n en utilisant une instruction récursive.

```
def fibo1(n):

    u0=1
    u1=1
    i=n
    while i>1 :
        u2=u0+u1
        u0=u1
        u1=u2
        i=i-1
    return u2
def fibo2(n):
    if n==0 or n==1 :
        return 1
    else:
        return fibo2(n-1)+fibo2(n-2)
#la complexité spatiale de la fonction récursive est:
#O(10) =O(9)+O(8)=O(8)+O(7)+O(7)+O(6)=.....c'est très gourmand
d'avoir deux appels en parallèle !
```

Exercice 5 : Calcul de x^n (version récursive d'une méthode dichotomique)

1)
   a) Ecrire une fonction récursive *puissance*( ) qui prend pour argument x et n (un entier naturel) et qui calcule x^n.
   b) Estimer le nombre de multiplication nécessaire pour calculer x^n

On peut remarquer que si :

- n est paire alors $x^n = (x^p)^2$ en posant $n = 2p$
- n est impaire alors $x^n = (x^p)^2 \times x$ en posant $n = 2p + 1$

2)
   a) Ecrire une fonction récursive *puissance2*( ) qui prend pour argument x et n (un entier naturel) et qui calcule x^n.
   b) L'unité de coût de l'algorithme précédent est le nombre de multiplication. Estimer la complexité temporelle $T(n)$ associée à ce nouveau programme.

```
def puissance(x,n):
    if n==0:
        return 1
    else :
        return x*puissance(x,n-1)
print(puissance(10,2))
#T(0)=> pas de multiplication
#T(1)=>10*1 => une multiplication
#T(n)=O(n)
#de même d'un point de vue spatial : on est en O(n)
def puissance2(x,n):
    if n==0:
        return 1
    else :
        p=puissance2 (x,n//2)
        if n%2 == 0:
            return p*p
        else :
            return p*p*x
print(puissance2(10,3))
#il s'agit d'une méthode "dichotomique"
#chaque appel conduit n=>n/2=>...1
#donc T(n)=log2(n)
```

Prenons le calcul de x^{100} , on prévoit $\log_2 100 \approx 7$ appels

$puissance2(x, 100)$	
$p = puissance2(x, 50)$	$p * p$
$p = puissance2(x, 25)$	$p * p$
$p = puissance2(x, 12)$	$p * p * x$
$p = puissance2(x, 6)$	$p * p$
$p = puissance2(x, 3)$	$p * p$
$p = puissance2(x, 1)$	$p * p * x$
$p = puissance2(x, 0)$	x
$p = puissance2(x, 0) = x$	
$p = puissance2(x, 1) = p^3$	
$p = puissance2(x, 3) = p^6$	
$p = puissance2(x, 6) = p^{12}$	
$p = puissance2(x, 12) = p^{25}$	
$p = puissance2(x, 25) = p^{50}$	
$p = puissance2(x, 50) = p^{100}$	

Exercice 6 : Recherche dichotomique d'un élément dans une liste triée

On considère une liste L de n nombres triés par ordre croissant.

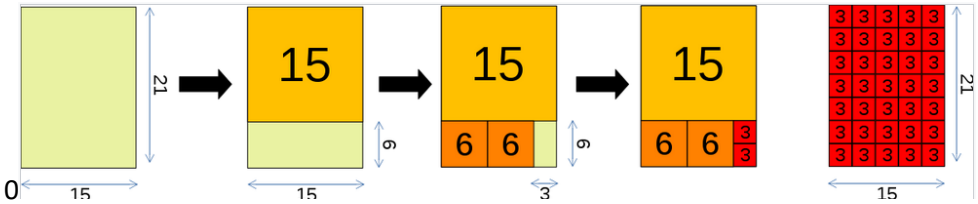
1) Proposer une instruction itérative dichotomique qui renvoie l'index de l'élément recherché (un message d'erreur est retourné si l'élément n'est pas présent dans L). Cette fonction prend une liste triée et "élément recherché en argument.
2) On souhaite écrire la fonction précédente avec un algorithme récursif dichotomique qui renvoie l'index de l'élément recherché (un message d'erreur est retourné si l'élément n'est pas présent dans L). On pourra rajouter un argument supplémentaire à la fonction en plus de la liste triée et de l'élément recherché.

```
def recherche(L,e):

    deb=0
    fin=len(L)-1
    while fin>=deb:
        m=(deb+fin)//2
        if e==L[m]:
            return True
        elif e>L[m] :
            deb=m+1
        else :
            fin =m-1
    return False
print(recherche([0,1,2,3,4],4))
    m = len(L)//2
    if len(L) == 0:
        return print("pas dans la liste")
    else :
        if L[len(L)//2]==elt:
            return print("l'élément est à la position
{}".format(index+m))
        elif L[len(L)//2]>elt:
            return dicho7(L[0:m],elt,index)
        else:
            return dicho7(L[m+1:],elt,index+m+1)
```

Si on suppose la liste de longueur n paire, alors la fin du programme aura lieu lorsque $\frac{n}{2^k} = 1$ soit $k = log_2(n)$

Exercice 7 : PGCD d'Euclide

On peut donner une interprétation graphique de ce plus grand nombre qui divise deux autres nombres (sans reste). Associons ce couple d'entiers aux dimensions d'un rectangle : leur PGCD est la longueur du côté du plus grand carré permettant de carreler entièrement ce rectangle. L'algorithme décompose ce rectangle en carrés, de plus en plus petits, par divisions euclidiennes successives, de la longueur par la largeur, puis de la largeur par le reste, jusqu'à un reste nul.



1) Proposer une fonction prenant pour argument deux nombres entiers pour lesquels on cherche le pgcd. Cette fonction utilisera une boucle while.
2) Reprendre la fonction précédente mais avec une méthode récursive.

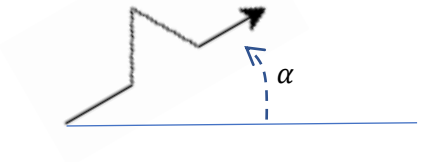
```
def pgcd1(nbr1,nbr2):
    while nbr1%nbr2 !=0:
        nbr1,nbr2=nbr2,nbr1%nbr2
    return nbr2
print(pgcd1(21,15))
def pgcd2(nbr1,nbr2):
    if nbr1%nbr2 == 0:
        return nbr2
    else :
        return pgcd2(nbr2,nbr1%nbr2)#return essentiel !
print(pgcd2(21,15))
```

Exercice 8 : Fractales : exemple du flocon de Koch

Une fractale est une figure qui présente une structure similaire à toutes les échelles. Dans cet exercice, on va utiliser le module *turtle* qui permet de réaliser des dessins. On va utiliser les fonctions suivantes :

- $turtle.pos(\ )$ qui renvoie les coordonnées x, y de la position courante du crayon.
- $turtle.goto((x,y))$ qui déplace le crayon jusqu'au point de coordonné (x,y) en traçant un trait
- On terminera le programme par $turtle.mainloop()$ afin d'afficher le dessin et $turtle.reset()$ afin de nettoyer l'écran.

1) Ecrire une fonction $trait(L, alpha)$ qui trace un segment de longueur $L/3$ dans la direction $alpha$. Faire le test avec $L = 300$ et $\alpha = 0$.
2) Dans le cas du flocon de Koch, la structure de base est représentée ci-dessous. Chaque segment est de longueur $\frac{L}{3}$ et les changements de direction par rapport à la direction α sont de $\pm\frac{\pi}{3}$. Proposer un algorithme $motif(L, alpha)$ permettant de tracer le motif ci-dessous dans la direction $alpha$ en utilisant plusieurs fois la fonction $trait$. Faire le test avec $L = 300$ $\alpha = 0$.

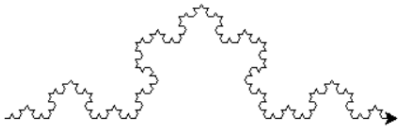
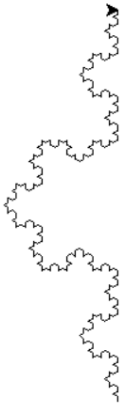


On souhaite répéter le motif à l'aide d'une méthode récursive dans une direction α donnée par rapport à l'horizontal.

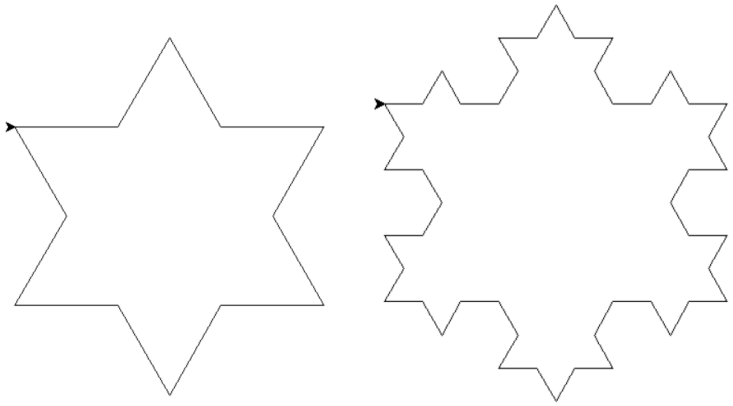
On propose ci-dessous une fonction récursive $cote(L, alpha, n)$ qui, si $n \geq 1$, effectue des appels récursifs permettant de tracer plusieurs fois les quatre segments définis par le cas de base ci-dessus (le niveau de

récursion n diminuant de 1 à chaque appel). Lorsque $n = 0$, ce cas de base conduit au tracé du motif

```
def cote(L,alpha,n):#la 1e valeur de alpha constitue la
direction de référence du coté
    if n==0:
        motif(L,alpha)
    else :
        cote(L/3,alpha,n-1)
        cote(L/3,alpha+np.pi/3,n-1)
        cote(L/3,alpha-np.pi/3,n-1)
        cote(L/3,alpha,n-1)
```

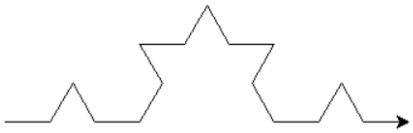
$L = 300, \alpha = 0, n = 3$	$L = 300, \alpha = \frac{\pi}{2}, n = 4$
	

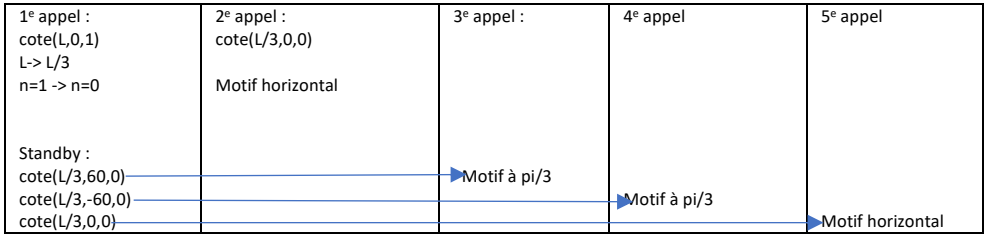
3) Tester la fonction précédente $cote(300,0,1)$ et expliquer la forme obtenue en analysant les différents appels récursifs.
4) Pour tracer un flocon de Koch, il suffit d'appeler 3 fois la fonction $cote$ afin de former un triangle équilatéral avec un nombre n d'itérations souhaitées. Donner la fonction $flocon(L, alpha, n)$ permettant d'obtenir un flocon. Obtenir les figures suivantes :



```python
import turtle
import numpy as np


def trait(L,alpha):
    """trace un trait de longueur L/3 dans la direction
alpha"""
    x,y=turtle.pos()
    turtle.goto(x+(L/3)*np.cos(alpha),y+(L/3)*np.sin(alpha))
#cas_de_base(300,np.pi/3)
def motif(L,alpha):
    """trace le motif : 4 brache de L/3 dans la direction
alpha"""
    trait(L,alpha)
    trait(L,alpha+np.pi/3)
    trait(L,alpha-np.pi/3)
    trait(L,alpha)
#motif(300,np.pi/3)
```

3)






```python
def flocon(L,alpha,n):
    cote(L,alpha,n)
    cote(L,alpha-2*np.pi/3,n)
    cote(L,alpha+2*np.pi/3,n)
#flocon(1000,0,0)=> étoile simple
```

On a ici un programme très gourmand :

$$T(n) = 3T_{cote}(n) = 3 * \left(4T_{cote}(n-1)\right) = 3 * 4^n = O(4^n)$$

Exercice 9 : Calcul de la valeur d'un polynôme

On considère un polynôme $p(x)$ de degré $n-1$ tel que :

$$p(x) = b_0 + b_1 x + b_2 x^2 + \cdots + b_{n-1} x^{n-1}$$

Ce polynôme peut donc être caractérisé par la liste de n éléments :

$$L = [b_0, b_1, b_2, \ldots b_{n-1}]$$

1) On souhaite trouver un algorithme itératif permettant d'évaluer la valeur $p(x_0)$ en utilisant la liste L.
   a) Ecrire une fonction $poly(L, x_0)$ prenant pour argument L et x_0 et retournant la valeur $p(x_0)$
   b) Commenter la complexité (dans le pire des cas et en considérant que l'unité de coût est la multiplication) de votre programme.
2) On peut remarquer que $p(x)$ peut aussi s'écrire :

$$p(x) = \Big((b_{n-1}x + b_{n-2})x + \cdots + \big((b_4 x + b_3)x + b_2\big)x + b_1\Big)x + b_0$$

   a) Proposer une fonction $poly2(L, x_0)$ prenant pour argument L et x_0 et retournant la valeur $p(x_0)$ et utilisant un schéma itératif.

Pour implémenter un schéma récursif, on peut remarquer que :

$$p(x) = b_0 + x(b_1 + x(b_2 + x(b_3 + \cdots + x(b_{n-2} + b_{n-1}x)\ldots)$$

   b) Proposer une fonction $poly3(L, x_0)$ prenant pour argument L et x_0 et retournant la valeur $p(x_0)$ et utilisant un schéma récursif
   c) Commenter la complexité (dans le pire des cas et en considérant que l'unité de coût est la multiplication) de votre programme.

```python
L=[1,2,3]
#p(x)=1+2x+3x^2
#p(2)=1+4+3*4=17

def poly1(x0,L):
    resultat = 0
    for i in range(len(L)):
        resultat = resultat + L[i]*x0**i
    return resultat
print (poly1(2,L))
#cet algorithme nécessite 1+2+3+...n multiplications
#sa complexité est quadratique si tous les termes de la liste sont non nuls
```

```python
def poly2(x0,L):
    resultat =L[-1]
    for i in range(len(L)-2,-1,-1):
        resultat = resultat*x0+L[i]
    return resultat
print (poly2(2,L))
```

```python
def poly3(x0,L):
    if len(L)==0:
        return 0
    else :
        resultat = L[0]
        return resultat +x0*poly3(x0,L[1:])
print (poly3(2,L))
#cet algorithme nécessite effectue n appels
#sa complexité est linéaire
```

Exercice 10 : Conversion binaire

La conversion d'un mot binaire sur 8 bits en base 10 peut s'écrire par l'intermédiaire d'une liste $L = [b_7, b_6, b_5, b_4, b_3, b_2, b_1, b_0]$. La valeur associée à ce mot binaire est alors donnée par la somme S :

$$S = (b_7 2^7 + b_6 2^6 + b_5 2^5 + b_4 2^4 + b_3 2^3 + b_2 2^2 + b_1 2^1 + b_0 2^0)$$

1) On souhaite trouver un algorithme itératif permettant d'évaluer la valeur S en utilisant la liste L.
   a) Ecrire une fonction $Conv(L)$ prenant pour argument L et retournant la valeur S
   b) Commenter la complexité (dans le pire des cas et en considérant que l'unité de coût est la multiplication) de votre programme.
2) On peut remarquer que S peut aussi s'écrire :

$$p(x) = \Big((b_7 2 + b_6)2 + \cdots + \big((b_4 2 + b_3)2 + b_2\big) + b_1\Big)2 + b_0$$

   a) Proposer une fonction $Conv2(L)$ prenant pour argument L et retournant la valeur S et utilisant un schéma itératif.

Pour implémenter un schéma récursif, on peut remarquer que :

$$S = b_0 + 2(b_1 + 2(b_2 + 2(b_3 + \cdots + 2(b_6 + b_7 2) \ldots)$$

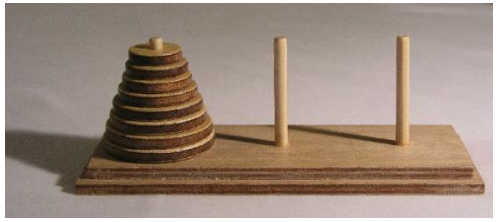
   b) Proposer une fonction $Conv3(L)$ prenant pour argument L et retournant la valeur S et utilisant un schéma récursif.
   c) Commenter la complexité (dans le pire des cas et en considérant que l'unité de coût est la multiplication) de votre programme.

```python
L=[1,1,1,1,0,0,0,0]
print(2**7+2**6+2**5+2**4)

def conv1(L):
    resultat = 0
    for i in range(len(L)):
        resultat = resultat + L[i]*2**(len(L)-1-i)
    return resultat
print (conv1(L))
#cet algorithme nécessite 1+2+3+...n multiplications
#sa complexité est quadratique si tous les termes de la liste
sont non nuls

def conv2(L):
    resultat =L[0]
    for i in range(1,len(L)):
        resultat = resultat*2+L[i]
    return resultat
print (conv2(L))

def conv3(L):
    if len(L)==0:
        return 0
    else :
        resultat = L[-1]
        return resultat +2*conv3(L[:-1])
print (conv3(L))
#cet algorithme nécessite effectue n appels
#sa complexité est linéaire
```

Exercice 11 : Tours de Hanoï

Le but : Reconstituer la pile initialement de gauche (piquet de départ) à droite (piquet d'arrivée)

La règle : Un grand disque ne peut jamais être au-dessus d'un disque plus petit que lui et on manipule un disque à la fois.



Résoudre ce problème en utilisant des boucles est extrêmement difficile. En utilisant une approche récursive, la solution est aisée. Pour le résoudre pour n disques, supposons que la solution soit connue pour $n-1$ disques, il suffit alors de réaliser les étapes suivantes :

- Déplacer les $n-1$ premiers disques (en suivant les règles du jeu) du piquet de départ au piquet intermédiaire ;
- Déplacer le disque restant (le plus grand) du piquet de départ au piquet d'arrivée ;
- Déplacer les $n-1$ disques du piquet intermédiaire au piquet d'arrivée

Si la solution pour $n-1$ disques est connues, celle pour n disques est facile à construire. Mais pour construire la solution pour $n-1$ disques, il faut néanmoins connaître la solution pour $n-2$….Finalement, pour construire la solution pour n disques, il suffit de connaître la solution pour un disque (chose aisée : il suffit de déplacer ce disque du piquet de départ au piquet d'arrivée).

L'état des trois piquets est mémorisé à l'aide de trois listes (qui joueront le rôle de piles) : $debut, inter, fin$. Pour chaque piquet, empiler consiste à mettre un anneau par-dessus (méthode *append*) et dépiler à l'enlever (méthode *pop*). En vous inspirant de la méthode de résolution proposée, écrire la fonction récursive $hanoi(n, depart, inter, fin)$ permettant d'arriver à la solution.

```python
def hanoi(n,depart,inter,fin):
    if n == 1:
        fin.append(depart.pop())
    else :
        hanoi(n-1,depart,fin,inter)
        hanoi(1,depart,inter,fin)
        hanoi(n-1,inter,depart,fin)
depart=[3,2,1]
inter=[]
fin=[]

print(depart,inter,fin)
hanoi(3,depart,inter,fin)
print(depart,inter,fin)
```

Terminaison :

L'algorithme se termine car, à chaque appel récursif, la variable n est décrémentée. Lorsque n atteint 1, la fonction hanoi déplace le dernier disque (condition d'arrêt).

Correction :

La correction de l'algorithme se fait par récurrence sur la propriété :

P(n) = « l'algorithme déplace correctement n disques ».

1 - P(1) est vraie.

2 - Supposons que P(n-1) soit vraie. Dans ce cas, le premier appel récursif déplace correctement les n-1 disques supérieurs de la colonne début vers la colonne inter. L'opération suivante déplace le n^{ième} disque de la colonne début vers la colonne fin. Enfin, le dernier appel récursif déplace les n-1 disques de la colonne inter vers la colonne fin.
Au final, les n disques ont été déplacés de la colonne D à la colonne A.

Complexité de l'algorithme : (en nombre de déplacements)

On a :

$$T(n) = 1 + 2T(n-1) = 1 + 2\big(1 + T(n-2)\big) = 1 + 2\Big(1 + 2\big(1 + T(n-3)\big)\Big)$$

$$T(n) \propto 1 + 2^1 + 2^2 + \cdots + 2^{n-1} = \frac{1-2^n}{1-2} = 2^n - 1$$

Complexité exponentielle impraticable pour des grandes valeurs de n!

Exercice 12 : Tri-rapide

L'algorithme de tri rapide d'une liste L est basé sur le principe suivant :

- Si L est vide alors l'algorithme retourne une liste vide
- Sinon :
  - → On choisit un pivot p (ici le 1^{e} élément de la liste)
  - → On construit deux sous listes : une liste L_1 constituée des éléments inférieurs à p et une liste L_2 formée des éléments plus grands que p
  - → Cette fonction retourne la concaténation de L_1 récursivement triée avec $[p]$ et L_2 triée récursivement.

Ecrire l'algorithme associé à ce type de tri.

.

```python
def tri_rapide(L):
    if len(L) == 0:
        return []
    else :
        p=L[0]
        L1=[]
        L2=[]
        for i in L[1:]:
            if i<p:
                L1.append(i)
            else :
                L2.append(i)
        return tri_rapide(L1)+[p]+tri_rapide(L2)
L=[9,8,7,6,5,4,3,2,1,0]
print(tri_rapide(L))
```