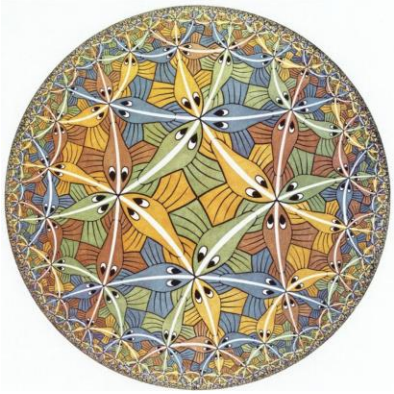
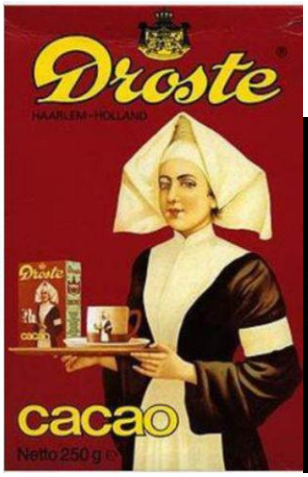
La récursivité

En mathématiques, en informatique, en biologie, mais aussi dans notre quotidien, nous faisons souvent face à des situations où un problème doit être résolu en utilisant une méthode de résolution qui est répétée plusieurs fois. Dans l'itération, cette méthode est appliquée par paliers de façon séquentielle, dans la récursivité, la méthode s'appelle elle-même. La récursivité est un principe de pensée exigeant et est souvent désigné comme « trop compliqué ». La récursivité est cependant si fondamentale qu'il n'est pas possible de l'éviter.

Dans le domaine des arts, c'est l'artiste néerlandais Maurits Cornelis Escher (1898-1972) qui fait le plus grand usage de la récursivité De son côté, la publicité a aussi utilisé la mise en abyme, rendant célèbres les fromages «La vache qui rit», le vermouth «Dubonnet», et le chocolat «Droste».





En informatique, une fonction récursive est une fonction qui s'appelle elle même.

La récursivité

1. Introduction sur un exemple : dessiner un triangle

Nous souhaitons dessiner un triangle de la façon suivante :
[ ]
[ ] [ ]
[ ] [ ] [ ]
[ ] [ ] [ ] [ ]
Pour cela nous allons définir une fonction :

```
def printTriangle(sideLenght):
```

Le triangle précédent sera affiché en appelant la fonction la fonction printTriangle(4). Pour comprendre comment la récursivité peut intervenir, il suffit de noter qu'un triangle de taille 4 peut-être obtenu à partir d'un triangle de taille 3 :
[ ]
[ ] [ ]
[ ] [ ] [ ]
[ ] [ ] [ ] [ ]
On « print » un triangle de taille 3 et on « print » une ligne avec 4 [ ]. Pour un triangle de taille quelconque, on écrira :

```
def printTriangle(sideLenght):
    printTriangle(sideLenght-1)
    print("[ ]" * sideLenght)
```

Il y a cependant un problème avec cette idée. Quand la taille du triangle est 1, on ne peut pas appeler printTriangle(0) et printTriangle(-1) etc… Il est donc nécessaire de traiter spécifiquement ce cas spécial et de ne rien « printer » quand sideLenght est inférieur à 1.

```
def printTriangle(sideLenght):
    if sideLenght < 1 : return
    printTriangle(sideLenght-1)
    print("[ ]" * sideLenght)
```

Il y a deux grandes idées à retenir pour traiter un problème par récursivité :

1) **Un calcul par récursivité résout un problème en utilisant une solution du même problème mais avec une entrée (input) plus simple.**
2) **Pour terminer une récursivité, il doit y avoir un cas spécial (dit cas de base) pour l'entrée (input) la plus simple.**

La fonction s'appelle encore et encore avec une longueur de plus en plus petite jusqu'a atteindre la valeur 0, la fonction arrête alors de s'appeler. Voici ce qu'il se passe quand on appelle la fonction avec une longueur de 4 :

- The call `printTriangle(4)` calls `printTriangle(3)`.
  - The call `printTriangle(3)` calls `printTriangle(2)`.
    - The call `printTriangle(2)` calls `printTriangle(1)`.
      - The call `printTriangle(1)` calls `printTriangle(0)`.
        - The call `printTriangle(0)` returns, doing nothing.
      - The call `printTriangle(1)` prints `[]`.
    - The call `printTriangle(2)` prints `[][]`.
  - The call `printTriangle(3)` prints `[][][]`.
- The call `printTriangle(4)` prints `[][][][]`.

Ce schéma de récursivité semble compliqué mais la clé pour utiliser avec succès les fonctions récursives est de ne pas trop penser à cela ☺
Ici la récursivité n'est pas vraiment nécessaire, nous aurions pu utiliser une boucle for :

```
def printTriangle(sideLenght):
    for i in range(1, sideLenght + 1):
        print("[ ]" * i)
```

La récursivité

Cependant beaucoup de personnes pensent que la récursivité est plus simple et efficace en général. Voici le programme complet :

2) : Cas de base, cas le plus simple.

```
1  ##
2  #  This program demonstrates how to print a triangle using recursion.
3  #
4
5  def main() :
6     printTriangle(4)
7
8  ## Prints a triangle with a given side length.
9  #  @param sideLength an integer indicating the length of the bottom row
10 #
11 def printTriangle(sideLength) :
12    if sideLength < 1 : return
13    printTriangle(sideLength - 1)
14
15    # Print the row at the bottom.
16    print("[]" * sideLength)
17
18 # Start the program.
19 main()
```

1) : La fonction s'auto appelle pour une entrée plus simple.

2. Quelques petits exemples classiques

2.1. Calcul de la factorielle

On rappelle que $n! \equiv n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1$ avec $0! \equiv 1$. Voici un exemple de calcul par récursivité :

2) : Cas de base, cas le plus simple.

```
def factorielle(n):
    if n==1 or n==0 :
        return 1
    else :
        return n*factorielle(n-1)
```

1) : La fonction s'auto appelle pour une entrée plus simple.

Il est possible de calculer la factorielle par une méthode itérative avec une boucle for :

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

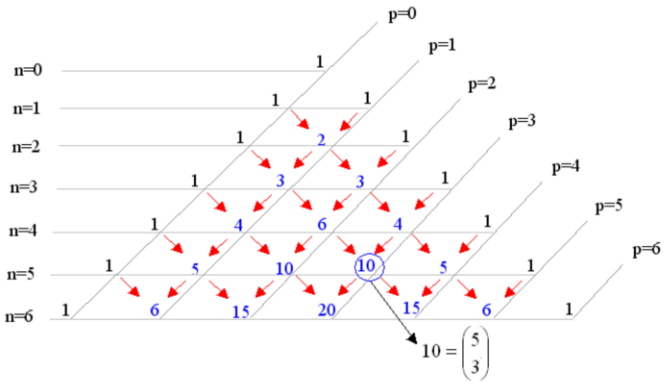
2.2. Le calcul des coefficients binomiaux

La fonction $C(n,p)$ donne le nombre de façon de choisir p (un entier) objets parmi n (un entier) avec évidemment $n < p$. Vous avez déjà dû rencontrer les relations suivantes :

$$C(n,0) = C(n,n) = 1$$

$C(n,p) = C(n-1,p-1) + C(n-1,p)$ (triangle de Pascal, cf. figure ci-dessous)

La récursivité




```
def C(n,p):
    if (n>p) and (p>0):
        return C(n-1,p-1)+C(n-1,p)
    else:
        return 1
```

2) : Cas de base, cas le plus simple.

1) : La fonction s'auto appelle (2 fois) pour deux entrées plus simple.

Le calcul de $C(n,p)$ par récursivité semble magique puisque que cela ne nécessite jamais l'utilisation de la relation bien connue $C(n,p)=\frac{n!}{(n-p)!p!}$. On utilise encore une fois le lien entre le cas à entrée plus complexe et les deux cas avec des entrées plus simples: $C(n,p)=C(n-1,p-1)+C(n-1,p)$.

3. L'efficacité de la récursivité, exemple de la suite de Fibonacci

La récursivité peut-être un outil puissant pour la mise ne œuvre d'algorithme complexe comme vous le verrez dans votre cours d'informatique. Mais d'un autre côté, la récursivité peut conduire à un algorithme de piètre performance en terme de temps de calcul. Nous allons entrevoir une partie de ce problème sur l'exemple de la fameuse suite de Fibonacci.
Cette dernière est définie de la façon suivante :

$$f_1 = f_2 = 1$$
$$f_n = f_{n-1} + f_{n-2}$$

La valeur de chaque terme est la somme des deux précédents. Les dix premiers termes de la suite sont :

$$1,1,2,3,5,8,13,21,34,55$$

Les termes de la suite sont très faciles à calculer ; l'entrée suivante est 34+55=89.

Voici un programme qui calcul les termes de la suite par récursivité et les résultats pour les 35 premiers termes :

La récursivité

```
1  ##
2  #  This program computes Fibonacci numbers using a recursive function.
3  #
4
5  def main() :
6     n = int(input("Enter n: "))
7     for i in range(1, n + 1) :
8        f = fib(i)
9        print("fib(%d) = %d" % (i, f))
10
11 ## Computes a Fibonacci number.
12 #  @param n an integer
13 #  @return the nth Fibonacci number
14 #
15 def fib(n) :
16    if n <= 2 :
17       return 1
18    else :
19       return fib(n - 1) + fib(n - 2)
20
21 # Start the program.
22 main()
```

2) : Cas de base, cas le plus simple.

1) : La fonction s'auto appelle (2 fois) pour deux entrées plus simple.

Program Run

```
Enter n: 35
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
fib(6) = 8
fib(7) = 13
. . .
fib(35) = 9227465
```

Le programme semble fonctionner correctement. Cependant vous pouvez constater qu'en exécutant le programme, le temps de calcul des termes augmente de façon dramatique avec n. Cela n'a pas de sens, il est plus rapide de calculer les termes avec un crayon et une calculatrice !!

Pour comprendre ce qu'il se passe, le programme suivant « print » chaque étape pour avoir une trace de ce que fait le programme :

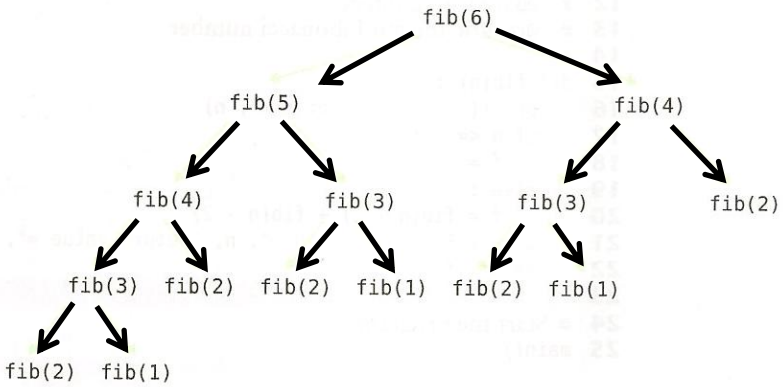
```
6  def main() :
7     n = int(input("Enter n: "))
8     f = fib(n)
9     print("fib(%d) = %d" % (n, f))
10
11 ## Computes a Fibonacci number.
12 #  @param n an integer
13 #  @return the nth Fibonacci number
14 #
15 def fib(n) :
16    print("Entering fib: n =", n)
17    if n <= 2 :
18       f = 1
19    else :
20       f = fib(n - 1) + fib(n - 2)
21    print("Exiting fib: n =", n, "return value =", f)
22    return f
23
24 # Start the program.
25 main()
```

Program Run

```
Enter n: 6
Entering fib: n = 6
Entering fib: n = 5
Entering fib: n = 4
Entering fib: n = 3
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Entering fib: n = 1
Exiting fib: n = 1 return value = 1
Exiting fib: n = 3 return value = 2
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Exiting fib: n = 4 return value = 3
Entering fib: n = 3
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Entering fib: n = 1
Exiting fib: n = 1 return value = 1
Exiting fib: n = 3 return value = 2
Exiting fib: n = 5 return value = 5
Entering fib: n = 4
Entering fib: n = 3
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Entering fib: n = 1
Exiting fib: n = 1 return value = 1
Exiting fib: n = 3 return value = 2
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Exiting fib: n = 4 return value = 3
Exiting fib: n = 6 return value = 8
fib(6) = 8
```

La récursivité

On peut aussi représenter les appels de fonction dans un arbre.



On voit que ce n'est pas efficace : par exemple, fib(3) est appelé trois fois, ce qui est une perte de temps. De plus on imagine bien que l'arbre va devenir très vite gigantesque, avec un très grand nombre d'appels inutiles. Cela est très différent d'un calcul avec un crayon et un papier ou chaque terme n'est calculé qu'une fois. Le programme suivant imite un calcul à la main, il s'agit d'un programme par itération.

```
1  ##
2  #  This program computes Fibonacci numbers using an iterative function.
3  #
4
5  def main() :
6     n = int(input("Enter n: "))
7     for i in range(1, n + 1) :
8        f = fib(i)
9        print("fib(%d) = %d" % (i, f))
10
11 ## Computes a Fibonacci number.
12 #  @param n an integer
13 #  @return the nth Fibonacci number
14 #
15 def fib(n) :
16    if n <= 2 :
17       return 1
18    else :
19       olderValue = 1
20       oldValue = 1
21       newValue = 1
22       for i in range(3, n + 1) :
23          newValue = oldValue + olderValue
24          olderValue = oldValue
25          oldValue = newValue
26
27       return newValue
28
29 # Start the program.
30 main()
```

Program Run

```
Enter n: 50
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
fib(6) = 8
fib(7) = 13
. . .
fib(50) = 12586269025
```

Vous pouvez constater que ce programme est beaucoup plus rapide.
Le tableau ci-après donne les ordres de grandeur du temps de calcul du programme par itération et du programme par récursivité.

La récursivité

	Fonction récursive (fib1)	Fonction itérative (fib2)
k	Temps (en secondes)	Temps (en secondes)
10	$7 \cdot 10^{-5}$	$3 \cdot 10^{-6}$
20	$3 \cdot 10^{-3}$	$3 \cdot 10^{-6}$
30	0.5	$3 \cdot 10^{-6}$
40	58	$4 \cdot 10^{-6}$
50	7264	$4 \cdot 10^{-6}$

Pour conclure, on peut retenir que de façon occasionnelle, la solution par récursivité est beaucoup plus lente que la solution par itération. Cependant, dans la plupart des cas, la récursivité est simplement un peu plus lente que l'itération. Mais, dans de très nombreux cas, la solution par récursivité est plus simple à comprendre et à mettre en œuvre qu'une solution par itération.
Selon l'informaticien (et créateur de l'interpréteur GhostScript pour le langage graphique PostScript) L. Peter Deutsh : « L'itération est humaine, la récursivité est divine ».

4. Exercices d'application

Exercice 1 :

☐ Soit une chaine de caractères, écrire un algorithme récursif permettant de déterminer sa longueur.

Exercice 2 :
Soit la suite définie par :

$$\begin{cases} U_n = 1 \text{ si } n < 2 \\ U_n = 3U_{n-1} + U_{n-2} \text{ si non} \end{cases}$$

☐ Ecrire un programme récursif permettant de calculer le nième terme de la suite.

Exercice 3 :

☐ Soit un tableau X de N entiers, écrire une fonction récursive simple permettant de déterminer le maximum du tableau.

Exercice 4 :

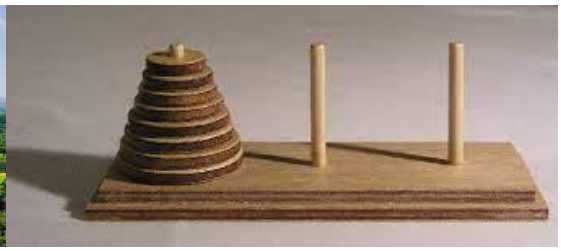
☐ Ecrire une fonction Python permettant de déterminer si une chaîne de caractères est ou non un palindrome (i.e. pouvant être lue indifféremment de la gauche vers la droite ou de la droite vers la gauche).

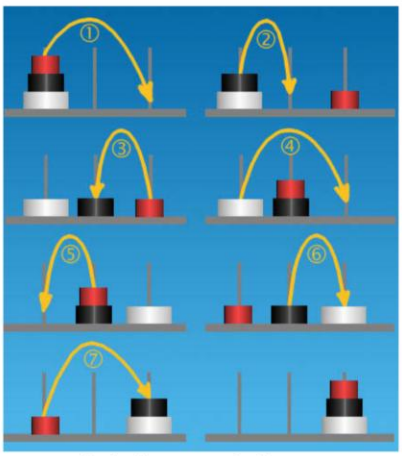
5. Projet 1: les tours de Hanoi

Le casse-tête des tours de Hanoï est un jeu de réflexion consistant à déplacer des disques de diamètres différents d'une tour de « départ » à une tour d' « arrivée » en passant par une tour « intermédiaire » et ceci en un minimum de coups, tout en respectant les règles suivante :
✔ On ne peut pas déplacer plus d'un disque à la fois .
✔ On ne peut placer un disque que sur un autre disque plus grand que lui ou sur un emplacement vide. On suppose que cette dernière règle est également respectée dans la configuration de départ.

Le problème mathématique des tours de Hanoï a été inventé par le mathématicien français Édouard Lucas. Il est publié dans le tome 3 de ses Récréations mathématiques, parues à titre posthume en 1892.

La récursivité

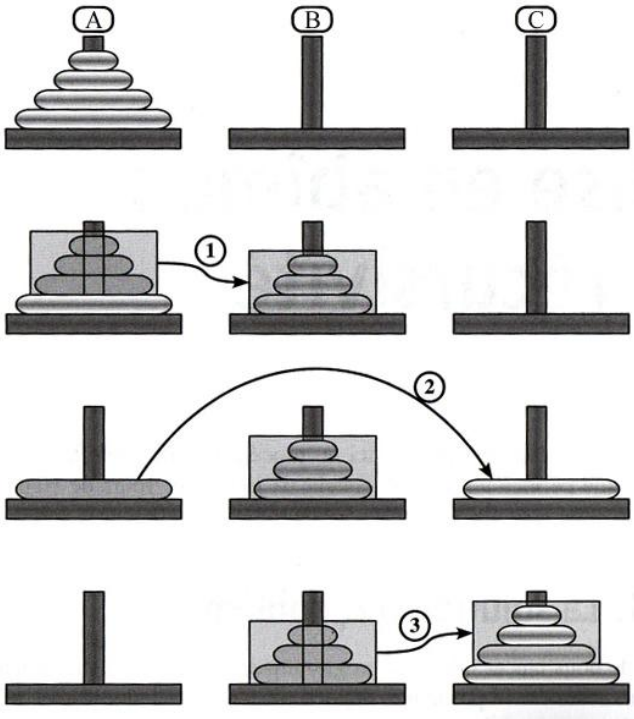


Solution pour 3 disques

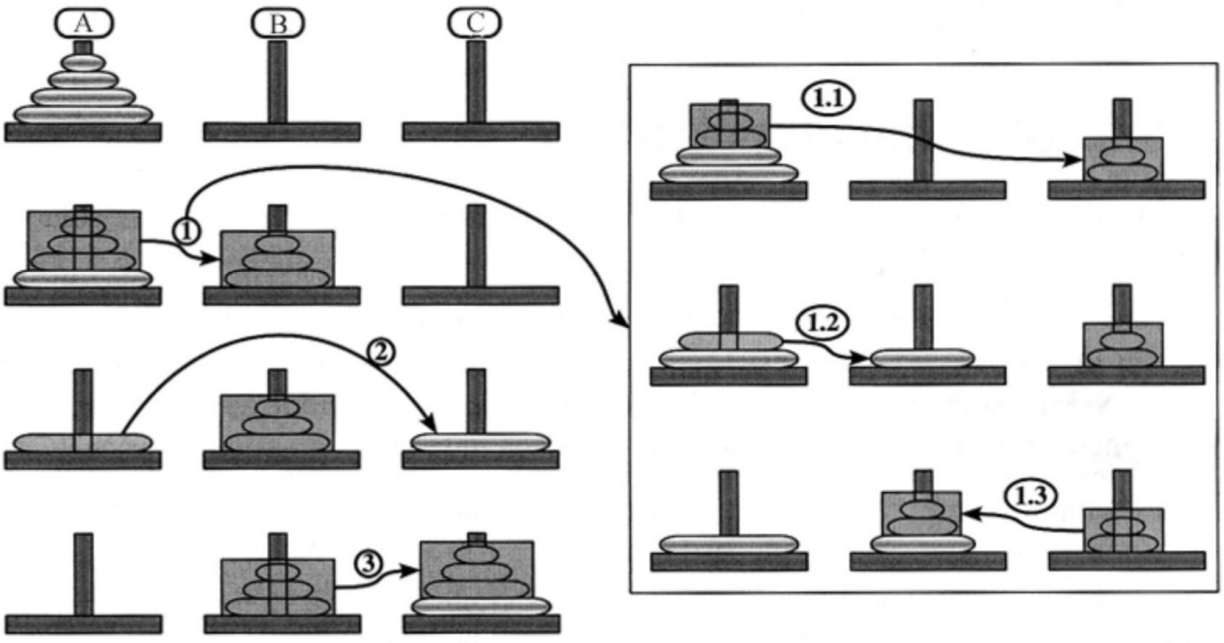
Un jeu à 64 disques requiert un minimum de $2^{64}-1$ déplacements. En admettant qu'il faille 1 seconde pour déplacer un disque, ce qui fait 86 400 déplacements par jour, la fin du jeu aurait lieu au bout d'environ 213 000 milliards de jours, ce qui équivaut à peu près à 584,5 milliards d'années, soit 43 fois l'âge estimé de l'univers (13,7 milliards d'années environ).

Pour résoudre le problème des tours de Hanoï, il faut raisonner récursivement. Pour fixer les idées, prenons par exemple quatre disques. Parmi eux, un est remarquable : le plus grand. Il ne peut être posé sur aucun autre, il est donc le plus contraignant à déplacer. Une solution idéale ne devrait déplacer ce disque qu'une seule fois. Nommons A, B et C les trois piquets, la pile de disques au départ sur A. Idéalement, donc, le plus grand disque sera déplacé de A à C en une seule fois. Comme il ne peut être posé sur aucun autre disque – car il est le plus grand –, cela ne pourra se faire que si aucun disque n'est présent sur C. Autrement dit, on ne pourra effectuer ce déplacement particulier que si le grand disque est seul sur A, et que tous les autres disques sont sur B. Incidemment, nous venons de réduire le problème : pour pouvoir déplacer nos quatre disques de A vers C, il faut d'abord déplacer (seulement) trois disques de A vers B, puis déplacer le grand sur C, et enfin déplacer les trois disques de B vers C.

La récursivité



Le grand disque présente une autre particularité : il est le seul sur lequel on peut poser n'importe lequel des autres disques. Donc, dans le cadre d'un déplacement des trois disques supérieurs, il n'a aucune incidence : tout se passe comme s'il n'était pas là. Cette remarque nous amène à un constat intéressant : on peut traiter le problème du déplacement des trois disques exactement de la même manière que nous avons traité celui des quatre. La figure ci-dessous, dans sa partie droite, montre comment la première des trois étapes peut elle-même être décomposée en trois « sous-étapes » : déplacer les deux disques supérieurs, puis le disque inférieur, puis déplacer les deux disques supérieurs. La troisième étape de la partie gauche pourrait être décomposée de la même façon.



En résumé, déplacer n disques de A vers C en passant par B consiste à :
1) déplacer $(n-1)$ disques de A vers B (en passant par C);
2) déplacer le plus grand disque de A vers C ;
3) déplacer $(n-1)$ disques de B vers C (en passant par A).

Les étapes 1 et 3 peuvent elles-mêmes se décomposer selon le même principe, sauf que les rôles des paquets sont intervertis. Par exemple, dans la première, on va de A vers B, donc forcément par l'intermédiaire de C. Voici la marche à suivre, donnée sur deux niveaux :

1) déplacer $(n-1)$ disques de A vers B (en passant par C);
  1.1) déplacer $(n-2)$ disques de A vers C (en passant par B);
  1.2) déplacer un disque de A vers B ;
  1.3) déplacer $(n-2)$ disques de C vers B (en passant par A).
2) déplacer le plus grand disque de A vers C ;
3) déplacer $(n-1)$ disques de B vers C (en passant par A).
  3.1) déplacer $(n-2)$ disques de B vers A (en passant par C);
  3.2) déplacer un disque de B vers C ;
  3.3) déplacer $(n-2)$ disques de A vers C (en passant par B).

Et ainsi de suite...

☐ Ecrire un algorithme récursif permettant d'imprimer les déplacements successifs des disques.

6. Projet 2 : Tracer d'une courbe fractale par récursivité avec la bibliothèque Turtle de Python

6.1. Turtle

La bibliothèque de programmation Turtle du langage Python permet de commander les déplacements d'un objet tortue (une tortue ou un curseur) dans un plan, comme dans le langage Logo. Ce dernier a été créé dans les années 1970 et a été utilisé dans les années 1980 pour l'apprentissage de la programmation. La tortue est caractérisée par sa position (un couple de coordonnées cartésiennes) et l'angle entre sa tête (ou la flèche du curseur) et une demi-droite de base qui par défaut est orientée par le vecteur Est de coordonnées (1, 0). Par défaut, la tortue avance en ligne droite d'un certain nombre de pixels sur la demi-droite dont un vecteur directeur a pour origine sa queue et pour extrémité sa tête. Le tableau ci-dessous donne les principales commandes de Turtle.

Table 1 Turtle Functions

Function	Description
`turtle.backward(`*distance*`)`	Moves backward *distance* in the opposite direction.
`turtle.clear()`	Clears the turtle's drawing from the window, but does not move the turtle.
`turtle.forward(`*distance*`)`	Moves forward *distance* in the current direction.
`turtle.goto(`*x*`, `*y*`)`	Moves to absolute position (x, y).
`turtle.heading()`	Returns the turtle's current heading in degrees.
`turtle.hideturtle()`	Hides the turtle.

La récursivité

`turtle.home()`	Moves to the origin (0,0) and sets the heading to the east.
`turtle.left(`*angle*`)`	Turns left by *angle* degrees.
`turtle.pencolor(`*colorname*`)` `turtle.pencolor(`*red*`, `*green*`, `*blue*`)`	Sets the pen color. The color can be specified by name or by *red*, *green*, and *blue* color values in the range [0 ... 255].
`turtle.pendown()`	Puts the pen down to draw when moving.
`turtle.pensize(`*width*`)`	Sets the line thickness to *width*.
`turtle.penup()`	Picks the pen up to stop drawing when moving.
`turtle.reset()`	Clears the turtle's drawing from the window, moves the turtle to the origin (0, 0), and sets the heading to the east.
`turtle.right(`*angle*`)`	Turns right by *angle* degrees.
`turtle.showturtle()`	Shows the turtle.

Pour vous familiariser avec Turtle, voici des exemples pour tracer des figures géométriques simples :

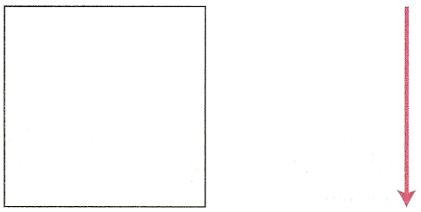
✔ Dessiner un carré et une flèche rouge :

```
##
#  This program draws a rectangle and vertical line using Python's
#  turtle graphics package.
#
import turtle

# Draw a square in the default color and pen size.
turtle.pendown()
turtle.forward(100)
turtle.right(90)
turtle.forward(100)
turtle.right(90)
turtle.forward(100)
turtle.right(90)
turtle.forward(100)

# Draw a larger red vertical line to the right of the box.
turtle.pensize(3)
turtle.pencolor("red")
turtle.penup()
turtle.right(90)
turtle.forward(200)
turtle.right(90)
turtle.pendown()
turtle.forward(100)

# Wait for user input to quit the program.
response = input("Press ENTER to quit.")
```



✔ Dessiner un carré avec une boucle for:

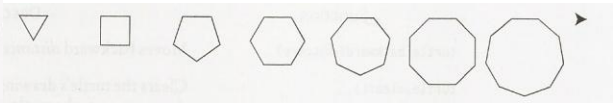
```
def square(width) :
   turtle.pendown()
   for in in range(0, 4) :
      turtle.forward(width)
      turtle.right(90)
   turtle.penup()
```

La récursivité

✔ Dessiner des polynômes réguliers

```
def regularPolygon(n, width) :
    turtle.pendown()
    for in in range(0, n) :
        turtle.forward(width)
        turtle.right(360 / n)
    turtle.penup()
```

```
def square(width) :
    turtle.pendown()
    for in in range(0, 4) :
        turtle.forward(width)
        turtle.right(90)
    turtle.penup()
```



```
for n in range(3, 10) :
    regularPolygon(n, 20)
    turtle.forward(60)
```

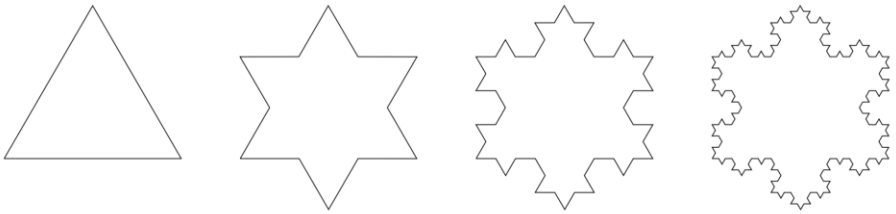
6.2. Tracés d'une courbe fractale : le flocon de Von Koch

Une fractale est une sorte de courbe mathématique un peu complexe extrêmement riche en détails, et qui possède une propriété intéressante visuellement : lorsque l'on regarde des détails de petite taille, on retrouve des formes correspondant aux détails de plus grande taille, il s'agit de l'auto-similarité. Ces objets sont omniprésents dans la nature. Le chou de Romanesco en est un exemple bien connu. Pour plus d'information : https://fr.wikipedia.org/wiki/Fractale.

La première courbe à tracer a été imaginée par le mathématicien suédois Niels Fabian Helge von Koch, afin de montrer que l'on pouvait tracer des courbes continues en tout point, mais dérivables en aucun.

Le principe est simple : on divise un segment initial en trois morceaux, et on construit un triangle équilatéral sans base au-dessus du morceau central. On réitère le processus n fois, n étant appelé l'ordre. Dans la figure suivante on voit les ordres 0, 1, 2 et 3 de cette fractale.

Si on trace trois fois cette figure, on obtient successivement un triangle, une étoile, puis un flocon de plus en plus complexe :



La récursivité

Le dessin de la fractale à l'ordre 0 est trivial, il suffit d'appeler forward() une fois.

À l'ordre 1, on va appeler quatre fois forward(), en appelant left() et right() pour changer la direction de la tortue entre chaque segments.

À l'ordre $n \geq 1$, on va appliquer le même principe qu'à l'ordre 1, mais en remplaçant les appels à forward() par des dessins de segments à l'ordre $n - 1$.

1) ☐ Que fait le programmes suivant ? Essayez de deviner avant d'exécuter le programme.

```
1
2   from turtle import *
3
4   for i in range (360):
5      forward (1)
6      right (1)
7
8
9   for i in range (5):
10      forward (200)
11      right (360 * 2/5)
12
13
14  mainloop ()
```

2) ☐ Koch à l'ordre 1. Ecrivez une fonction koch_1(longueur) qui dessine un segment de Koch à l'ordre 1, de la longueur spécifiée. Les sous-segments de la figure sont de longueur $\frac{\text{longueur}}{3}$ et les angles sont de 60 ou 120 degrés. Appelez cette fonction depuis votre programme pour vérifier son fonctionnement.

3) ☐ Début de la généralisation : ordre 0 ou 1. Modifiez votre fonction pour lui faire prendre en paramètre l'ordre de la fractale. La fonction est maintenant koch(n,longueur), ou n est l'ordre. Modifiez le corps de la fonction pour qu'elle gère correctement les cas $n = 0$ et $n = 1$ (le cas $n \geq 1$ viendra plus tard). Le code va ressembler à :

```
if n==0:
    # Cas n == 0 (trivial en utilisant forward()).
 else:
    # cas n == 1, comme ci-dessus.
```

Vérifiez que la fonction marche correctement pour ces deux valeurs.

Étonnement, il n'y a presque rien à changer pour généraliser notre fonction à une valeur quelconque de n. Pour l'instant, le cas $n = 1$ appelle 4 fois la fonction forward, qui correspond au cas $n = 0$. Nous avons vu que la fractale à l'ordre n devait utiliser la fractale à l'ordre $n - 1$ (c'est le cas puisque $0 = 1 - 1$). En remplaçant les appels à forward par des appels à koch(n - 1, ...), on ne changera pas le comportement de notre fonction koch à l'ordre 1, mais on lui permettra de gérer correctement les ordres $n \geq 2$.

4) ☐ Koch à l'ordre n. Modifiez la fonction Koch comme expliqué ci-dessus pour gérer les ordres $n \geq 2$. Testez votre fonction avec différentes valeurs de n (en pratique, on ne voit plus grand chose avec un ordre supérieur à 5 ou 6). Une manière de tester est d'utiliser le morceau de code suivant :

```
for i in range (10):
    penup ()
    goto(0, 70 * i)
    pendown ()
    koch(i, 300)
```

L'exécution peut être assez lente. Deux astuces pour l'accélérer :

– Appeler speed(0) avant de dessiner : ceci va régler la vitesse de la tortue au maximum.

– Appeler hideturtle() avant de dessiner, et showturtle() après. Le triangle représentant la tortue ne sera pas redessiné à chaque étape, on gagne beaucoup de temps.