RECURSIVITE

- Dans cette feuille est abordé le concept de récursivité.
- Une fonction récursive est une fonction qui fait appel à elle même dans sa définition.
- On peut rapprocher le concept de récursivité à celui de récurrence en mathématiques.
- Ce style de programmation permet de résoudre certains problèmes qu'il n'est parfois pas facile de traiter en programmant uniquement avec des boucles.

Ci-contre : Une image tirée du film "Inception"(Christopher Nolan, 2010).

1. Une autre façon de voir les choses

Exercice 1 :

Pour définir la somme des n premiers entiers, on à l'habitude d'écrire la formule suivante : $0+1+2+...+n$. A l'aide d'une boucle `for` , compléter la fonction `somme(n)` ci-dessous.

In [8]:
```
def somme(n):
    '''Renvoie la somme des n premiers entiers
    parametre : n entier naturel
    return : la somme des n premiers entiers
    '''
    S=0




    return S

assert(somme(3)==6)
assert(somme(5)==15)
```

Remarque :

- Cette fonction fait bien ce que l'on attend d'elle. Mais on peut remarquer que ce code n'est pas directement lié à la définition. Il n'y a rien qui laisse supposer qu'une variable intermédiaire `S` est nécéssaire pour calculer cette somme.

Définition récursive

- Il existe une autre façon de faire : pour calculer la somme des n premiers entiers, il suffit d'ajouter n à la somme des $n-1$ premiers entiers !
- Il s'agit d'une définition mathématique que l'on peut rapprocher de la notion de récurrence pour les suites.
- La fonction somme peut être définie ainsi :

$$somme(n) = \begin{cases} 0 \ si \ n = 0 \\ n + somme(n-1) \ si \ n > 0 \end{cases}$$

Exemples :

- $somme(0) = 0$
- $somme(1) = 1 + somme(0) = 1 + 0 = 1$
- $somme(2) = 2 + somme(1) = 2 + 1 = 3$
- $somme(3) = 3 + somme(2) = 3 + 3 = 6$

Remarques :

- La définition de $somme(n)$ dépend donc de $somme(n-1)$.
- La fonction somme fait appel à elle-même dans sa définition.Il s'agit d'une définition récursive.
- Ainsi , pour connaître $somme(n)$, il faut connaître $somme(n-1)$, donc $somme(n-2)$ et ce jusqu'à $somme(0)$ qui vaut 0.
- On obtient $somme(n)$ en ajoutant toutes ces valeurs.

Programmation récursive

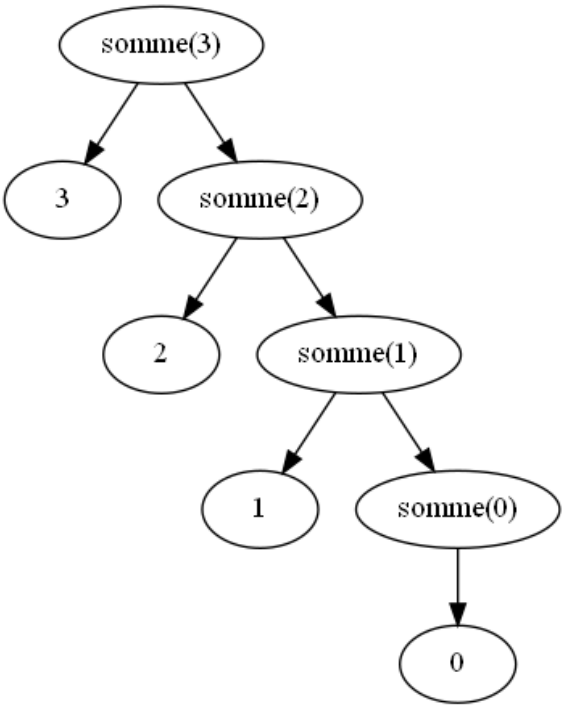
L'intérêt de cette définition récursive de la fonction somme est qu'elle est calculable, c'est à dire éxécutable par un ordinateur.

En particulier, voici le code python de cette définition :

In [3]:
```
def somme(n):
    if n==0:
        return 0
    else:
        return n + somme(n-1)

assert(somme(3)==6)
assert(somme(0)==0)
```

Remarque : Comment se passe l'éxécution de ce programme ?

- Si `n` vaut `0`, alors la valeur `0` est renvoyée, sinon on renvoie `n + somme(n-1)`.
- Cet appel qui fait référence à la fonction que l'on est en train de définir est un appel récursif.
- Voici ci-contre une façon de représenter ce qui se passe lorsque `somme(3)` est appelé.
- On appelle cette représentation un arbre d'appels.
- Pour calculer la valeur renvoyée par `somme(3)`, il faut appeler `somme(2)` qui va appeler `somme(1)` qui va appeler `somme(0)` qui va renvoyer `0`.
- Le calcul se fait donc à rebours : le calcul de `somme(0)=0` permet celui de `somme(1)=1+0` puis celui de `somme(2)=2+1` et enfin celui de `somme(3)=3+3`.



Exercice 2 :

En mathématiques, la factorielle d'un entier naturel n est le produit des nombres entiers strictement positifs inférieurs ou égaux à n, on la note $n!$.

Par exemples, $4! = 24$ et $5! = 120$.

Compléter la fonction ci-dessous en utilisant la récursivité.

In [5]:
```
def factor(n):
    '''parametre : n entier naturel non nul
    return : la factorielle de n'''




assert(factor(4)==24)
assert(factor(5)==120)
```

A retenir

Une formulation récursive d'une fonction est constituée de plusieurs cas:

- Le ou les cas de base : Ce sont ceux pour lesquels on peut obtenir le résultat sans faire appel à la fonction définie elle-même. Dans l'exemple de la fonction `somme(n)`, le cas de base est `n=0`.Les cas de base sont habituellement les cas de valeurs particulières faciles à obtenir.
- Le ou les cas récursifs : Ce sont ceux qui renvoient à la fonction en train d'être définie. Pour la fonction `somme(n)`, le cas récursif est `n > 0`.
- Pour écrire une fonction récursive, on peut commencer par identifier les cas de base.
- Il faut faire confiance à la récursion : on ne doit pas chercher à comprendre ce qui va se passer exactement lors des appels et supposer qu'ils vont donner de bons résultats pour les valeurs sur lesquelles ils opèrent.

Exercice 3 :

1. Compléter la défintion récursive ci-dessous de la fonction $puissance(x,n)$ qui calcule la valeur de la puissance nième d'un réel x: $x^n = \underbrace{x \times x \times \ldots \times x}_{n\ fois}$

$$puissance(x,n) = \begin{cases} ?\ si\ n = 0 \\ ?\ si\ n > 0 \end{cases}$$

1. Compléter ci-dessous la fonction `puissance(x,n)` à l'aide de la définition précédente.

In [7]:
```
def puissance(x,n):
    '''Calcule la puissance nième de x
    paramètres :
    x de type int ou float
    n entier naturel

    return :
    la puissance nième de x, de type int ou float
    '''









assert(puissance(2,0)==1)
assert(puissance(3,3)==27)
```

2. Des formulations plus riches

Toute formulation récursive comporte au moins un cas de base et un cas récursif. Mais une grande variété de formes est possible.

Cas de base multiples

Il peut y avoir plusieurs cas de base identifiables. Par exemple la définition de la fonction puissance peut se faire ainsi :

$$puissance(x,n) = \begin{cases} 1 \ si \ n = 0 \\ x \ si \ n = 1 \\ x \times puissance(x, n-1) \ si \ n > 1 \end{cases}$$

Ce deuxième cas de base permet d'éviter de faire la multiplication inutile de $x \times 1$ de la précédente définition.On pourrait bien sûr continuer à ajouter des cas de base pour $n = 2$, $n = 3$, mais cela ne réduit pas le nombre de multiplication à faire.

Exercice 4 :

Modifier la fonction `puissance(n,x)` ci-dessous en insérant le cas de base $n = 1$.

In [10]:
```
def puissance(x,n):
    '''Calcule la puissance nième de x
    paramètres :
    x de type int ou float
    n entier naturel

    return :
    la puissance nième de x, de type int ou float
    '''
    if n==0:
        return 1




    else :
        return x*puissance(x,n-1)

assert(puissance(2,0)==1)
assert(puissance(4,1)==4)
assert(puissance(3,3)==27)
```

Cas récursifs multiples

Il est également possible de définir une fonction avec plusieurs cas récursifs.

Exercice 5 :

La suite de Syracuse est définie ainsi :

A partir d'un premier terme entier naturel plus grand que 1, on divise ce terme par 2 s'il est pair, on le multiplie par 3 et on ajoute 1 s'il est impair. On recommence le même procédé pour obtenir les termes suivants.

Pour un premier terme s_0 donné, on peut ainsi obtenir le terme de rang n en écrivant la définition suivante :

$$syracuse(n) = \begin{cases} s_0 \ si \ n = 0 \\ \frac{syracuse(n-1)}{2} \ si \ syracuse(n-1) \ est \ pair \\ 3 \times syracuse(n-1) + 1 \ si \ syracuse(n-1) \ est \ impair \end{cases}$$

Compléter la fonction c-dessous à l'aide de cette définition.

In [38]:
```
s0=27

def syracuse(n):
    '''Calcul du terme de rang n
    de la suite de Syracuse de premier terme u0

    parametres :
    u0 entier naturel > 1
    n entier naturel

    return :
    terme de rang n, entier naturel
    '''









assert(syracuse(6)==94)
assert(syracuse(0)==s0)
```

Double récursion

Les expressions qui définissent le ou les cas récursifs peuvent contenir plusieurs appels à la fonction que l'on est en train de définir.

Exercice 6 :

Voici les premiers termes de la suite de Fibonacci : $1, 1, 2, 3, 5, 8$. Elle est définie par ses deux premiers termes égaux à 1 et par le fait que chaque terme suivant s'obtient par la somme des deux précédents.

Pour obtenir le terme de rang n, sa définition récursive est donc :

$$fibonacci(n) = \begin{cases} 1 \ si \ n \leq 1 \\ fibonacci(n-1) + fibonacci(n-2) \ si \ n \geq 2 \end{cases}$$

Compléter la fonction ci-dessous à l'aide de cette définition.

In [40]:
```
def fibonacci(n):
    '''Calcule le terme de rang n de la suite de Fibonacci
    parametre : n entier naturel
    return : le terme de rang n de la suite, entier naturel
    '''









assert(fibonacci(0)==1)
assert(fibonacci(1)==1)
assert(fibonacci(5)==8)
```

Récursion mutuelle

On peut définir plusieurs fonctions récursives en même temps, quand ces fonctions font références les unes aux autres.

Exercice 7 :

Voici une façon originale de définir la parité d'un nombre : un nombre pair(respectivement impair) est un nombre qui n'est pas impair(respectivement pair).

On peut ainsi définir deux fonctions $pair(n)$ et $impair(n)$ qui s'appellent mutuellement !

$$pair(n) = \begin{cases} True \ si \ n = 0 \\ impair(n-1) \ si \ n \geq 1 \end{cases} \quad impair(n) = \begin{cases} False \ si \ n = 0 \\ pair(n-1) \ si \ n \geq 1 \end{cases}$$

Compléter les deux fonctions ci-dessous à l'aide de la définition

In [ ]:
```
def pair(n):
    '''renvoie True si n est pair,
    False sinon'''




def impair(N):
    '''renvoie True si n est impair,
    False sinon'''




assert(pair(0)==True)
assert(impair(0)==True)
assert(pair(5)==False)
assert(imparir(5)==True)
```

3. Exécution des programmes

Une fois que l'on a trouvé une définition récursive, la programmation avec Python est assez aisée. Pour éviter les valeurs absurdes, les erreurs, les boucles infinies ou le dépassement de capacité, il faut néanmoins faire attention à deux aspects:

- Etre sûr que la définition est correctement formulée.
- Veiller à ne pas saturer l'espace mémoire avec de trop nombreux appels.

Bien écrire une définition récursive

- Terminaison :Il faut être sûr que l'on va finir par arriver sur un cas de base de la définition.
- Domaine de validité : Faire attention à ce que les valeurs utilisées lors des appels soient toujours dans le domaine de validité et qu'il y ait une définition pour toutes les valeurs du domaine.

Exercice 8 :

Changeons la définition de la fonction $somme(n)$ et considérons que pour la calculer, il suffit d'ôter $n+1$ à $somme(n+1)$.

$$somme(n) = \begin{cases} 0 \ si \ n = 0 \\ somme(n+1) - (n+1) \ si \ n > 0 \end{cases}$$

Expliquer ce qui ne va pas dans cette définition.

Réponse :

Pour calculer $somme(1)$, on va faire appel à $somme(2)$ qui va faire appel à $somme(3)$, etc. On ne retombe jamais sur le cas de base $n = 0$.

Exercice 9 :

Voici la définition d'une fonction g qui s'applique aux entiers naturels:

$$g(n) = \begin{cases} 1 \ si \ n = 0 \\ n + g(n-2) \ si \ n > 0 \end{cases}$$

Expliquer ce qui ne va pas dans cette définition.

Réponse :

Exercice 10 :

Voici la définition d'une fonction h qui s'applique aux entiers naturels:

$$h(n) = \begin{cases} 1 \ si \ n = 0 \\ n + h(n-1) \ si \ n > 1 \end{cases}$$

Expliquer ce qui ne va pas dans cette définition.

Réponse :

Nombre d'appels

Exemple :

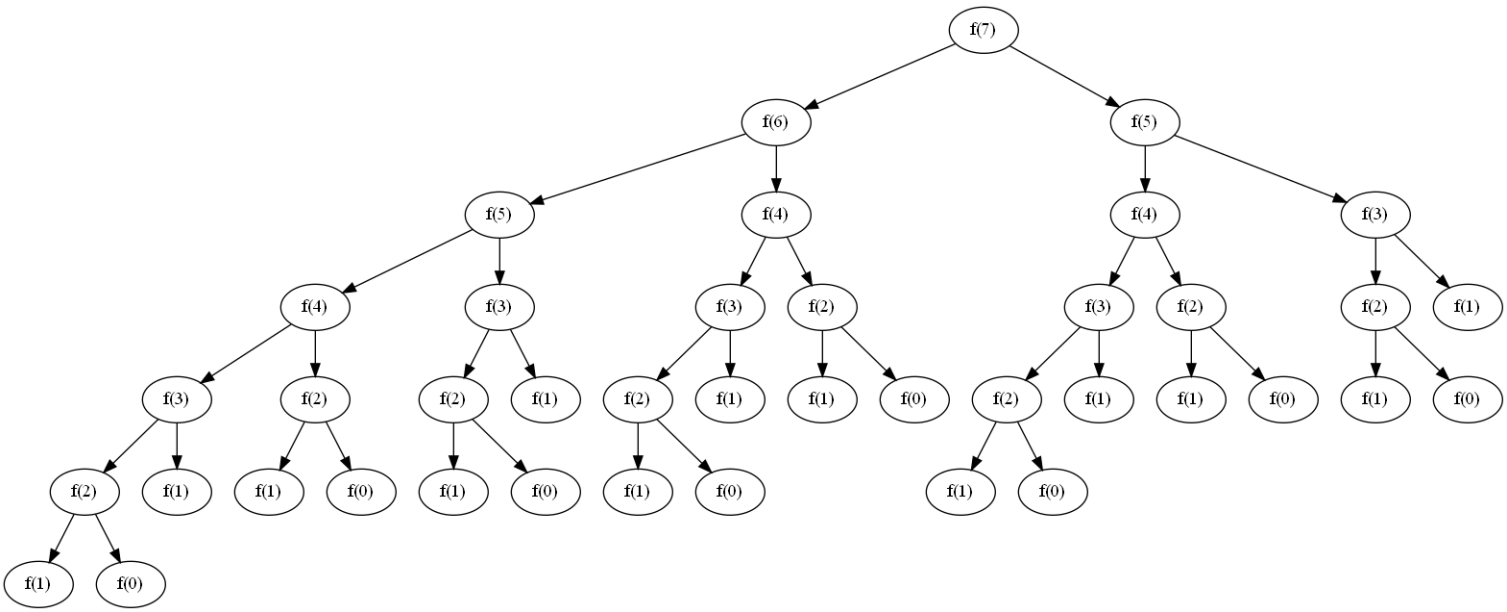
Reprenons la suite de Fibonacci, que l'on notera ici f pour plus de lisibilité pour la suite. Cette fonction est ici modifiée pour pouvoir se rendre compte des appels successifs effectués par python.

In [11]:
```
def f(n):
    '''Calcule le terme de rang n de la suite de Fibonacci
    parametre : n entier naturel
    return : le terme de rang n de la suite, entier naturel
    '''
    if n<=1:
        print('f(1)', end=' ')
        return 1
    else:
        print('f({})'.format(n-1), end=' ')
        print('f({})'.format(n-2), end=' ')
        return f(n-1)+f(n-2)

f(7)
```

```
f(6) f(5) f(5) f(4) f(4) f(3) f(3) f(2) f(2) f(1) f(1) f(0) f(1) f(1) f(1) f(1) f(0) f(1)
f(1) f(2) f(1) f(1) f(0) f(1) f(1) f(1) f(3) f(2) f(2) f(1) f(1) f(0) f(1) f(1) f(1) f(1)
f(0) f(1) f(1) f(4) f(3) f(3) f(2) f(2) f(1) f(1) f(0) f(1) f(1) f(1) f(1) f(0) f(1) f(1)
f(2) f(1) f(1) f(0) f(1) f(1) f(1)
```

Out[11]: 21

Et voici l'arbre des appels pour $f(7)$:



- Pour pouvoir effectuer ces calculs, l'espace mémoire est organisé sous forme d'une pile ou sont stockés les contextes d'éxécution de chaque appel de fonction(valeur de n, valeur de retour, opérations à effectuer...).
- Dans le cas de $f(7)$, on constate que certains calculs apparaissent plusieurs fois. Par exemple, il y a 5 appels pour $f(3)$, ce qui signifie que cette valeur est calculée 5 fois par le programme et qu'il y a donc 5 emplacements mémoire où l'on trouve le même contexte nécéssaire au calcul !
- Une définition récursive peut engendrer un grand nombre d'appels et donc occuper un espace mémoire important par la pile d'appels. Python limite par défaut le nombre d'appels autorisés à 1000.D'autres langages, plus spécialisés dans ce type de programmation permettent de mieux gérer ces empilements d'appels.
- Plus généralement, quand on écrit une fonction récursive , on cherche à choisir la bonne définition qui limite le nombre d'appels(voir exercice 14).

A retenir

- Certains problèmes difficiles à résoudre peuvent être décrits à l'aide d'une définition récursive assez intuitive.
- L'écriture d'une fonction récursive nécéssite de distinguer les cas de base, pour lesquels on donne un résultat facilement et les cas récursifs qui font appel à la fonction en cours de définition.
- Il faut veiller à bien construire sa définition (terminaison, domaine de validité).
- Il faut également avoir conscience que l'espace mémoire occupé peut rapidement devenir très important du fait des nombreux appels récursifs.

4. Exercices

Exercice 11: Puzzle algorithmique

On considère un tournoi de n équipes ($n \geq 2$) dans lequel chaque équipe à rencontré exactement une fois chacune des autres équipes. Il n'y a pas de match nul. Est-il possible dans tous les cas de lister les équipes de telle sorte que chaque équipe a gagné le match contre l'équipe qui est listée juste après ? Justifier la réponse.

Aide : La réponse est oui . On pourra répondre d'abord pour $n = 2$, puis pour $n = 3$.

Réponse :

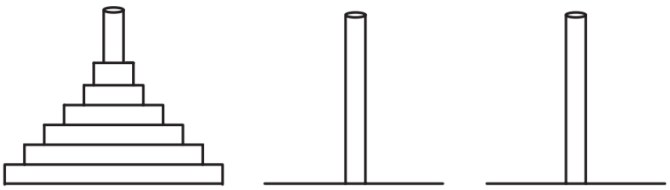
Exercice 12 : Nombre de chiffres

Compléter de façon récursive la fonction `n_chiffres(N)` ci-dessous qui renvoie le nombre de chiffres qui composent l'entier `N` .

In [ ]:
```
def n_chiffres(N):
    '''
    parametre :
    un entier naturel N

    return :
    le nombre de chiffres de N'''
```

Exercice 13 : Les tours de Hanoi

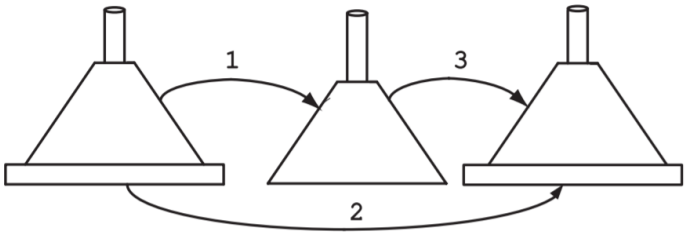
On considère n disques de différentes tailles($n \geq 1$) et trois axes. Initialement, tous les disques sont sur le premier axe, dans l'ordre croissant de leur taille, du plus grand en dessous jusqu'au plus petit sur le dessus. L'objectif est de transférer tous les disques sur un autre axe en un minimum de mouvements:



- Un seul disque peut être déplacé à la fois.
- Il est interdit de placer un disque plus grand sur un disque plus petit.

1. Tester ce jeu pour $n = 3$ (http://championmath.free.fr/tourhanoi.htm (http://championmath.free.fr/tourhanoi.htm)) . On note $M(3)$ le nombre minimum de mouvements à effectuer pour déplacer trois disques. Que vaut $M(3)$?

1. On note $M(n)$ le nombre minimum de mouvements nécéssaires pour déplacer n disques. A l'aide de l'image ci-contre, exprimer $M(n)$ en fonction de $M(n-1)$.



1. Compléter la fonction `M(n)` qui renvoie le nombre minimum de mouvements à effectuer pour déplacer `n` disques.

Réponses :

1.

1.

In [32]:
```
#3.
def M(n):
    '''Calcule le nombre minimal de mouvements
    pour déplacer n disques'''




print(M(3))
print(M(7))
```

```
7
127
```

Exercice 14 : Exponentiation rapide

Supposons que l'on dispose d'une fonction $carre(x)$ qui calcule le carré d'un nombre.

Voici alors une autre définition pour $puissance(x,n)$:

$$puissance(x,n) = \begin{cases} 1 \; si \; n = 0 \\ carre(puissance(x, \frac{n}{2})) \; si \; n \geq 1 \; et \; pair \\ x \times carre(puissance(x, \frac{n-1}{2}) \; si \; n \geq 1 \; et \; impair \end{cases}$$

1. Compléter la fonction `puissance(x,n)` à l'aide de la définition.

In [51]:
```
#1.
def carre(x):
    '''calcule le carre de x
    parametre :
    x de type int ou float

    return :
    le carre de x'''

    return x*x

def puissance(x,n):
    '''Calcule la puissance nième de x
    paramètres :
    x de type int ou float
    n entier naturel

    return :
    la puissance nième de x, de type int ou float
    '''









assert(puissance(2,0)==1)
assert(puissance(4,1)==4)
assert(puissance(3,3)==27)
```

1. Modifier la fonction ci-dessous pour qu'elle puisse compter le nombre d'appels de la fonction `puissance` .

In [53]:
```
#2.

N=1 #un compteur









print(puissance(2,8),N)
N=1 # on remet le compteur à 1
```

```
256 4
```

1. Combien d'appels de la fonction $puissance$ sont nécéssaires pour calculer :

- x^8 :
- x^{15} :
- x^{62} :

1. Avec la définition de l'exercice 4, combien d'appels de la fonction $puissance$ sont nécéssaires pour calculer :

- x^8 :
- x^{15} :
- x^{62} :

Exercice 15: Conjecture de Syracuse

Le suite de Syracuse est la suite d'entiers s_n définie par son premier terme $s_0 > 1$ et par la relation de récurrence :

$$s_n = \begin{cases} \frac{s_n}{2} \ si \ n \ est \ pair \\ 3s_n + 1 \ si \ n \ est \ impair \end{cases}$$

La conjecture de Syracuse affirme que quelque soit la valeur de s_0, il existe un rang n pour lequel $u_n = 1$.cette conjecture n'a encore jamais été démontrée et défie les chercheurs du monde entier.

Ecrire une fonction récursive `syracuse(u_n)` qui affiche les valeurs successives de `u_n` tant que `u_n` est plus grand que 1. Tester ensuite différentes valeurs et constater cette conjecture.

In [1]:
```
def syracuse(u_n):
    '''affiche les termes successifs de la suite de Syracuse
    tant qu'ils sont supérieurs à 1.
    parametre : u_n entier, premier terme de la suite
    sortie : affichage des termes successifs'''









syracuse(27)
```

Exercice 16 : La fractale de Koch

La fractale de Koch peut s'obtenir de manière récursive:

- Le cas de base à l'ordre 0 est un segment de longueur l.
- Le cas récursif d'ordre n s'obtient en divisant ce segment en trois morceaux de même longueur $\frac{l}{3}$, puis en dessinant un triangle équilatéral dont la base est le morceau du milieu(cette base n'est pas dessinée).

Ci-dessus les courbes d'ordre 0,1,2 et 3 sont dessinées respectivement de gauche à droite.

Compléter la fonction `koch(n,l)` qui dessine la fractale de Koch de profondeur `n` à partir d'u segment de longueur `l` , à l'aide du module `turtle` .

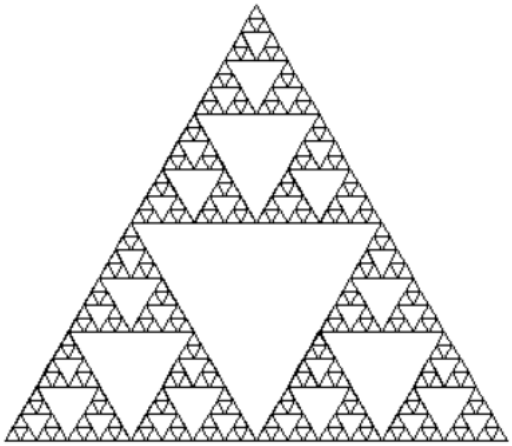
Mémo turtle :

- `forward(l)` : la tortue avance de l pixels dans le sens ou elle est orientée
- `left(60)` : la tortue tourne sur elle même vers la gauche d'un angle a en degrés

In [77]:
```
from turtle import *

def koch(n,l):
    '''dessine un flocon de koch
    de profondeur n, a partir d'un
    segment de longueur l'''









koch(3,200)
mainloop()
```

Exercice 17 : Triangle de Sierpinsky (DM ?)

- https://fr.wikipedia.org/wiki/Triangle_de_Sierpi%C5%84ski (https://fr.wikipedia.org/wiki/Triangle_de_Sierpi%C5%84ski)



Ecrire la fonction récursive `sierpinsky(n,l)` qui dessine le triangle de Sierpinsky à l'ordre n à partir d'un segment de longueur l. Ci-dessus, le résultat de `sierpinsky(5,300)` .

In [79]:
```
from turtle import *

def sierpinsky(n,l):
    '''trace un triangle de sierpinsky
    apres n itérations, à partir d'un
    triangle equilatéral de côté l
    '''









hideturtle()
speed('fastest')
sierpinsky(5,300)
```

FIN