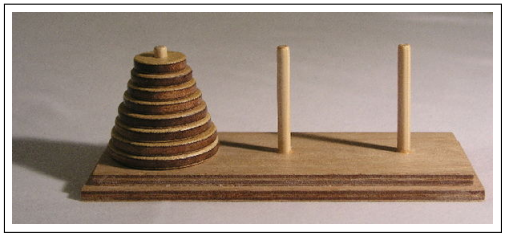
Chapitre 10

Fonctions récursives II Exemples élaborés d'usage de la récursivité

10.1 Le problème des tours d'Hanoï

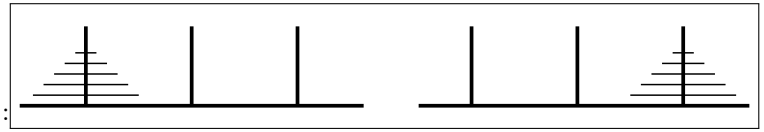
10.1.1 Enoncé du problème

Le problème des tours de Hanoï est un jeu de réflexion inventé par le Mathématicien français Edouard Lucas en 1883 :



Des disque percés de diamètre décroissants sont empilés sur une colonne. Il y a trois colonnes. Il s'agit de déplacer les disques de la colonne de gauche à la colonne de droite –en un minimum de coût– et en respectant les règles suivantes :

1. On ne peut déplacer qu'un disque à la fois, et sur la colonne de son choix.
2. On ne peut empiler un disque que sur une colonne vide ou sur un disque plus grand.

départ :  : fin

10.1.2 Résolution récursive

• Ecrivons un programme qui résout le problèmes des tours de Hanoï pour n disques.

• Pour le résoudre simplement et élégamment nous allons procéder par récursivité, en remarquant :

• Si l'on sait résoudre le problème pour $n-1$ disques alors il n'est pas difficile de le résoudre pour n disques :

1. Procéder aux mouvements permettant de résoudre avec les $n-1$ disques de plus petits diamètres pour que la position d'arrivée soit la colonne du milieu.
2. Déplacer le disque de plus gros diamètre sur la colonne de droite.
3. Procéder aux mouvements de résolution avec les $n-1$ disques de la colonne du milieu pour les déplacer sur la colonne de droite.

• On constitue une fonction :

`hanoi(n, Colonne_départ, Colonne_intermédiaire, Colonne_arrivée)`

• Schéma récursif de résolution (en pseudo-code) :

```
def hanoi(n, D, I, A):
    if n != 0:
        hanoi(n-1, D, A, I)
        déplacer le disque restant de D à A
        hanoi(n-1, I, D, A)
```

• Pour l'implémenter : on utilisera 3 piles pour `D`, `L`, `A`, de capacités illimités (c'est à dire 3 listes en `python`),

1. l'empilement s'obtient alors par `Pile.append(e)`,
2. le dépilement par `Pile.pop()`
3. le peek par `Pile[-1]`, la taille par `len(Pile)`, et "est vide ?" par `len(Pile) == 0` (que l'on n'utilisera pas ici).

• Initialement :

```
D = [ n-k for k in range(n) ]    # D = [ n, n-1, ..., 2, 1]
I = [ ]
A = [ ]
```

10.1.3 Code

```
def hanoi(n, D, I, A):     # Partie récursive
    if n!=0:
        hanoi(n-1,D,A,I)
        A.append(D.pop())    # empiler(A, depiler(D))
        hanoi(n-1,I,D,A)

def resoudreHanoi(n):    # Fonction à appeler pour initialiser
    D = [ n-i for i in range(n)]    # Initialisation Tours
    I = [ ]
    A = [ ]
    print(D,I,A)    # Affichage état au départ
    hanoi(n,D,I,A)
    print(D,I,A)    # Affichage état à l'arrivée
```

```
>>> resoudreHanoi(4)
[4, 3, 2, 1] [ ] [ ]
[ ] [ ] [4, 3, 2, 1]
```

10.1.4 Les tours de Hanoï - Complexité

• Déterminons la complexité de cet algorithme :

L'appel de `resoudreHanoi(n)` effectue :

1. n opérations élémentaire pour constituer la liste `D`,
2. n opérations élémentaires pour créer la liste `I` de capacité n,
3. n opérations élémentaires pour créer la liste `A` de capacité n,
4. $3.n$ opérations élémentaires pour écrire `D,I,A`, 2 fois
5. les opérations nécessaires à l'appel de `hanoi(n,D,I,A)`.

• Au total, si l'on appelle H_n le nombre d'opérations élémentaires nécessaires pour `hanoi(n,D,I,A)`, alors `resoudreHanoi(n)` nécessite $H_n + 9n$ opérations élémentaires.

• Calculons l'ordre de H_n.

• Calculons H_n, nombre d'opérations élémentaires pour `Hanoi(n,D,I,A)`, partie récursive de l'algorithme.

```
def hanoi(n, D, I, A):     # Partie récursive
    if n!=0:
        hanoi(n-1,D,A,I)
        A.append(D.pop())    # empiler(A, depiler(D))
        hanoi(n-1,I,D,A)
```

On obtient la relation de récurrence :

$$H_0 = 1 \qquad H_{k+1} = 1 + H_k + 2 + H_k = 2H_k + 3$$

(1 test, 2 appels de récursifs de coût H_k, et 2 opérations pour dépiler `D` et empiler `A` (l'espace ayant été réservé l'ajout en fin de liste se fait en $O(1)$).

Alors H_n est une suite arithmético-géométrique, de terme général $H_n = 4 \times 2^n - 3$ (exercice).

La résolution du problème nécessite $4.2^n - 3 + 9n$ opérations élémentaires : du même ordre que 2^n. Donc la complexité est exponentielle, d'ordre $\Theta(2^n)$.

METHODE GENERALE POUR CALCULER LA COMPLEXITE D'UNE FONCTION RECURSIVE SUIVANT UNE RECURRENCE SIMPLE.

10.1.5 Les tours de Hanoï : Affichage des mouvements

• Deuxième tentative : pour afficher les états successifs.

L'affichage doit se faire au départ, puis après chaque changement.

C'est à dire avant l'appel initial (départ) et après chaque instruction `A.append(D.pop())`.

Le problème est que chaque appel récursif permute l'ordre des 3 tours.

Si l'on se contente de l'instruction `print(D,I,A)` les 3 tours ne seront pas affichées dans l'ordre exact.

1^{ere} **méthode.** Pour cela on prend en quatrième argument la liste :

`T = [D, I, A]`

qui permet de mémoriser l'ordre des tours ; c'est `T` que l'on affiche par l'instruction `print(T)`.

Chaque appel crée une copie de `T` ; mais cette copie contient chaque fois les 3 adresses mémoires où sont stockées les données des 3 tours, dans le bon ordre.

```
def hanoi(n,D,I,A,T):
    if n!=0:
        hanoi(n-1,D,A,I,T)
        A.append(D.pop())
        print(T)            # Affichage des 3 tours
        hanoi(n-1,I,D,A,T)

def resoudreHanoi(n):
    D = [ n-i for i in range(n)]
    I = []
    A = []
    T = [D, I, A]
    print(T)        # Affichage de l'état initial
    hanoi(n,D,I,A,T)
```

2^{eme} **méthode.** Déclarer la liste `T = [D, I, A]` comme une **variable globale**. On n'a plus besoin de la passer en quatrième argument à la fonction `Hanoi()` :

```
def hanoi(n,D,I,A):
    if n!=0:
        hanoi(n-1,D,A,I)
        A.append(D.pop())
        print(T)        # Affichage des trois tours
        hanoi(n-1,I,D,A)
def resoudreHanoi(n):
    D = [ n-i for i in range(n)]
    I = [ ]
    A = [ ]
    # La variable T est déclarée globale,
    # on peut accéder à son contenu de partout :
    global T             # D'abord déclarer T comme globale
    T = [D, I, A]        # Puis lui affecter une valeur
    print(T)
    hanoi(n,D,I,A)
```

10.1.6 Les tours de Hanoï

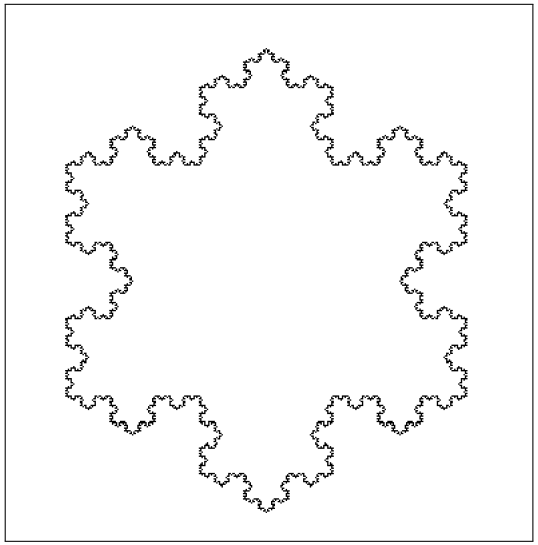
```
>>> resoudreHanoi(3)
[[3, 2, 1], [], []]
[[3, 2], [], [1]]
[[3], [2], [1]]
[[3], [2, 1], []]
[[], [2, 1], [3]]
[[1], [2], [3]]
[[1], [], [3, 2]]
[[], [], [3, 2, 1]]
```

• **Exercice * :** Démontrer que l'algorithme décrit résout bien le problème des Tours d'Hanoï en un minimum de déplacements.

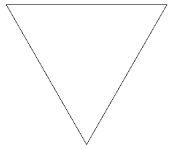
Esquisse : Par récurrence sur n.
Pour résoudre le problème : tôt ou tard il faudra déplacer le disque de plus gros diamètre n. Pour cela nécessairement les $n-1$ premiers disques devront être tous empilés dans le bon ordre sur une même colonne. Pour que le nombre de déplacements soit minimal il faut que le disque n soit déplacé d'entrée sur la colonne de droite.

10.2 Exemples fractals

10.2.1 Flocon de Von Koch

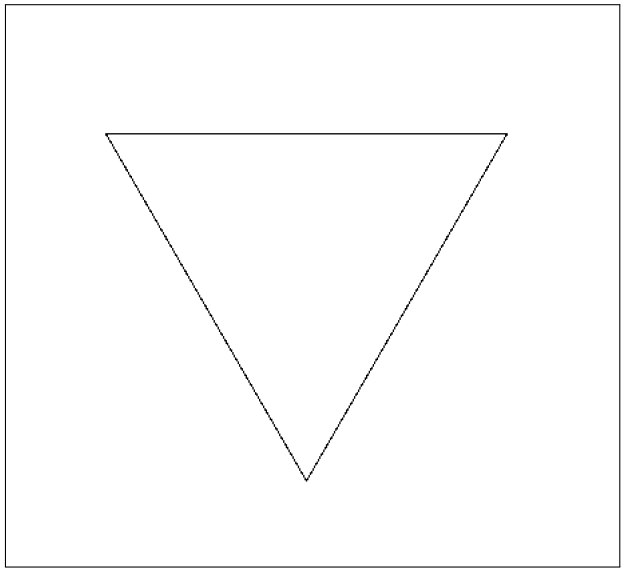


- Cette courbe est construite en partant d'un triangle équilatéral.



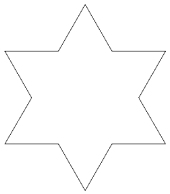
- Sur chaque segment :
  1. Diviser en 3 segments de mêmes longueurs.
  2. Considérer le triangle équilatéral extérieur construit sur le segment du milieu.
  3. Remplacer ce segment par les 2 autres côtés du triangle équilatéral.
- Recommencer avec chaque segment obtenu.

10.2.2 Flocon de Von Koch



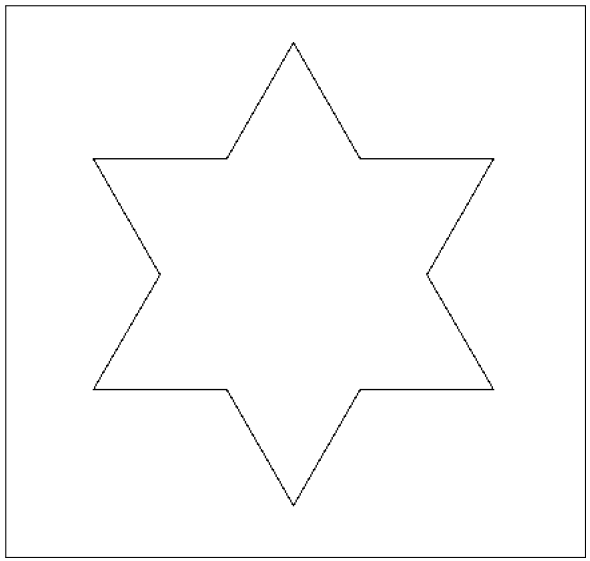
$n = 1$

- Cette courbe est construite en partant d'un triangle équilatéral.



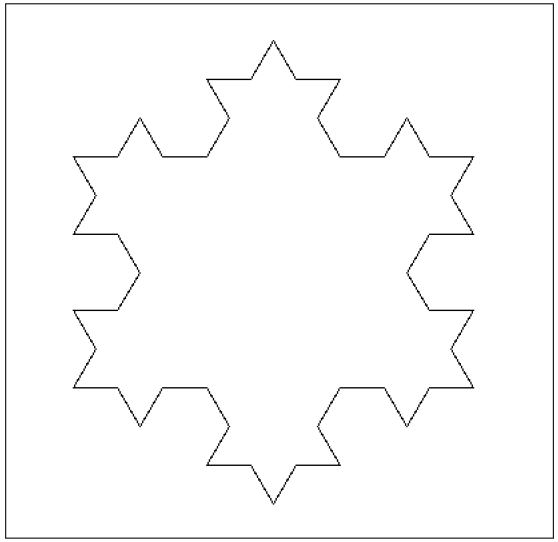
- Sur chaque segment :
  1. Diviser en 3 segments de mêmes longueurs.
  2. Considérer le triangle équilatéral extérieur construit sur le segment du milieu.
  3. Remplacer ce segment par les 2 autres côtés du triangle équilatéral.
- Recommencer avec chaque segment obtenu.

10.2.3 Flocon de Von Koch



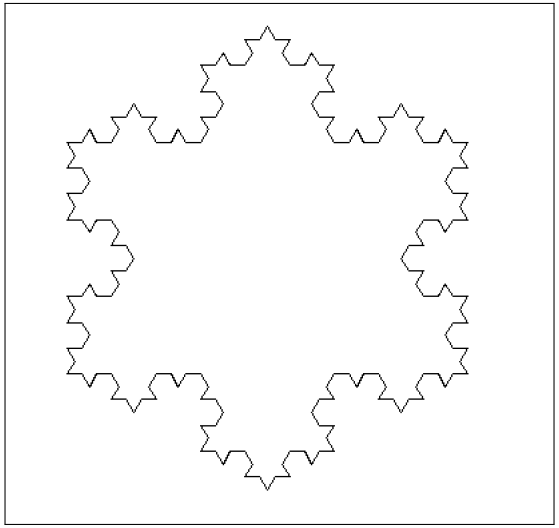
$n = 2$

10.2.4 Flocon de Von Koch



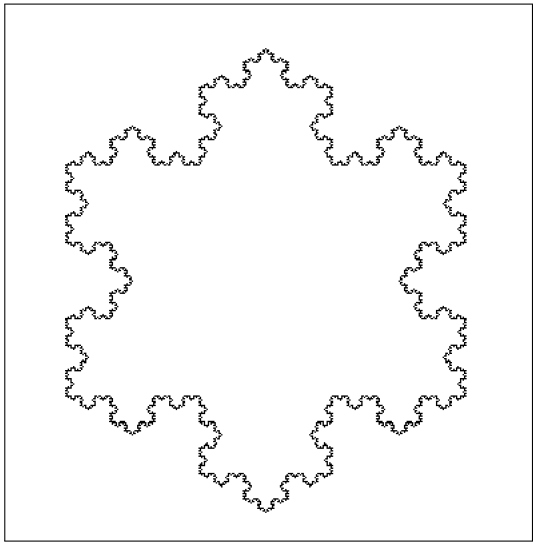
$n = 3$

10.2.5 Flocon de Von Koch



$n = 4$

10.2.6 Flocon de Von Koch



$n = 6$

10.2.7 Tracé : graphiques avec le module `turtle`

10.2.8 Le module `turtle`

• Pour le tracé on utilise le module `turtle` (inspiré du `logo`, (80's)). On déplace une tortue dans le plan (rapporté à un repère orthonormé direct), qui peut tracer sur son passage.

• Principales instructions :

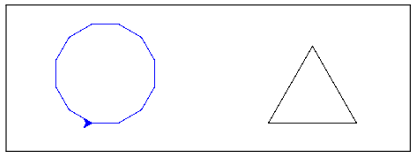
`reset()`	Efface la fenêtre graphique, réinitialisation
`up(), down()`	Relève, abaisse le crayon
`forward(d), backward(d)`	Avancer, reculer d'une distance `d`
`left(a), right(a)`	Tourner à gauche, droite, d'une angle `a` en degrés
`goto(x,y)`	Se déplace au point de coordonnées `(x,y)`
`position()`	Retourne la position courante
`color(couleur)`	Détermine la couleur = `'black'`, `'blue'`, `'red'`, ...
`width(l)`	Détermine l'épaisseur du trait `l`
`fill(p)`	Remplir un contour fermé : `p=True` et `p=False` Pour délimiter la figure à remplir
`circle(r)`	Trace un cercle de rayon `r`
`undo()`	Annule la dernière commande

• Voir l'aide en ligne : `https://docs.python.org/3.4/library/turtle.html`

• Fonction traçant un polygône :

```
import turtle as tt
def polygone(n=3, l=100, clr='black'):
    tt.color(clr)
    tt.down()
    for i in range(n):
        tt.forward(l)
        tt.left(360/n)
```

```
>>> tt.reset()
>>> polygone()
>>> tt.up()
>>> tt.goto(-200,0)
>>>
polygone(12,30,'blue')
```



10.2.9 Tracé du Flocon de Von Koch par récursivité

• Pour le tracer, on peut procéder par récursivité sur chacun des 3 segments du triangle initial :

`vonKoch(longueur,n)`

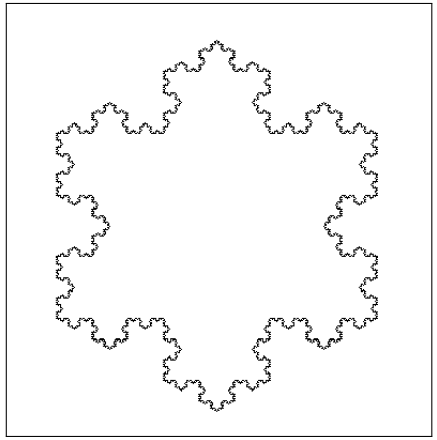
1. appeler `vonKoch(longueur /3, n-1)`
2. Tourner à gauche de 60° : `tt.left(60)`
3. appeler `vonKoch(longueur / 3, n-1)`
4. Tourner à droite de 120° : `tt.right(120)`
5. appeler `vonKoch(longueur / 3, n-1)`
6. Tourner à gauche de 60° : `tt.left(60)`
7. appeler `vonKoch(longueur / 3, n-1)`

10.2.10 Code

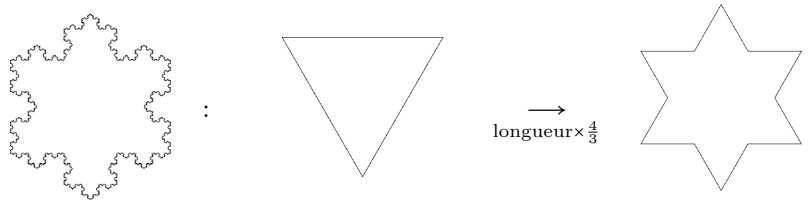
```
def vonkoch(longueur,n):
    if n == 1:
        tt.forward(longueur)
    else:
        l = longueur / 3
        vonkoch(l, n - 1); tt.left(60)
        vonkoch(l, n - 1); tt.right(120)
        vonkoch(l, n - 1); tt.left(60)
        vonkoch(l, n - 1)
```

```
def floconVonKoch(longueur, n):
    tt.pen(speed = 0)    # Accélération du mouvement
    tt.hideturtle()        # Pour ne pas tracer la tortue
    tt.up()
    tt.goto(-longueur/2, longueur/3) # Départ en haut à gauche
    tt.down()
    for i in range(3):
        vonkoch(longueur,n); tt.right(120)
```

```
>>> floconVonKoch(300,6)
```



10.2.11 Le périmètre est infini



- Le flocon de Von Koch est la 'courbe fractale' obtenue à la limite.

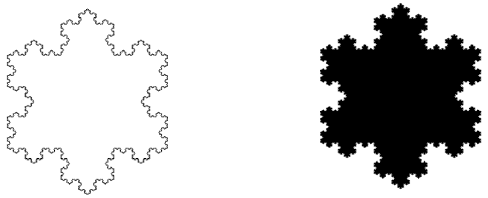
• Toutes les courbes intermédiaires ont une longueur finie, terme de la suite géométrique :

$$L_n = (3 \times \text{longueur}) \times \left(\frac{4}{3}\right)^{n-1}$$

Le flocon est de longueur infinie : $\lim L_n = +\infty$.

C'est la limite d'une suite géométrique de raison $\dfrac{4}{3} > 1$.

10.2.12 L'aire est finie



Longueur infinie Aire finie.

C'est une courbe de longueur infinie qui pourtant délimite un domaine d'aire finie :

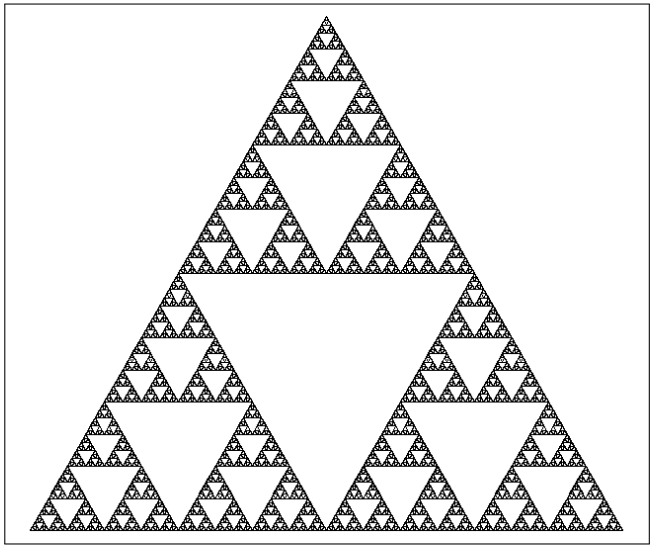
En effet : en posant L = longueur (paramètre initial) :

$$Aire_1 = \underbrace{\frac{\sqrt{3}}{2} \times L^2}_{\text{aire triangle}} \qquad Aire_{n+1} = Aire_n + \underbrace{3 \times 4^{n-1}}_{\text{Nbre petits triangle}} \times \frac{\sqrt{3}}{2} \times \Big(\underbrace{\frac{L}{3^n}}_{\text{de côté}} \Big)^2$$

$$Aire_{n+1} = Aire_n + \frac{3}{4} \times \left(\frac{4}{9}\right)^n \times \frac{\sqrt{3}}{2} \times L^2$$

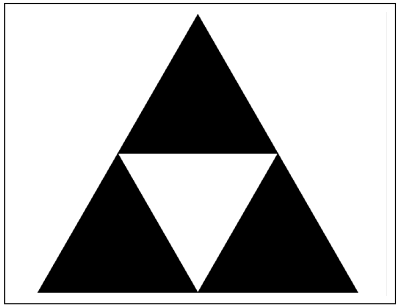
$Aire_n$: somme des termes d'une suite géométrique de raison < 1 : converge.

10.2.13 Triangle de Sierpinsky



• Construction :

1. A partir d'un triangle équilatéral plein de côtés de longueur l.
2. Tracer ses 3 médianes : elles découpent 4 triangles équilatéraux de côtés de longueur $l/2$.
3. Supprimer le triangle au centre.
4. Recommencer avec les 3 triangles extérieurs.



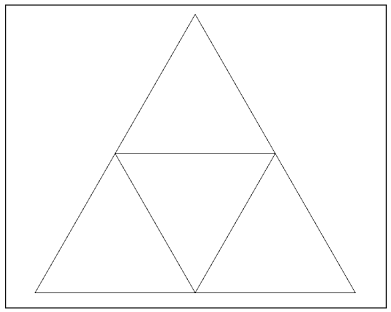
10.2.14 Tracé du Triangle de Sierpinsky

• On programmera plutôt :

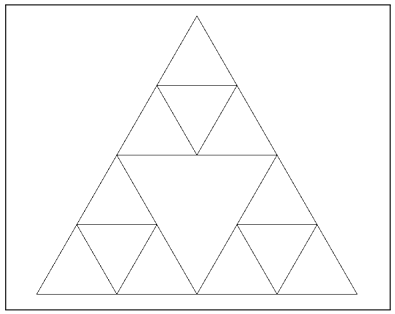
1. Construire un triangle équilatéral de côtés de longueur l.

2. Tracer ses 3 médianes : elles découpent 4 triangles équilatéraux de côtés de longueur $l/2$.

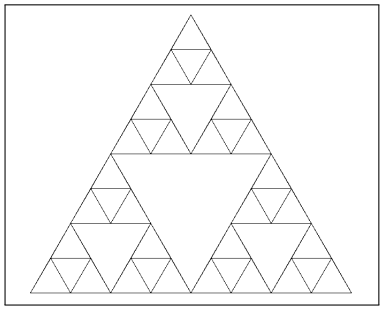
3. Recommencer avec les 3 triangles extérieurs.



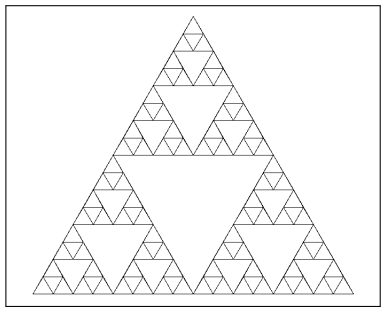
$n = 1$



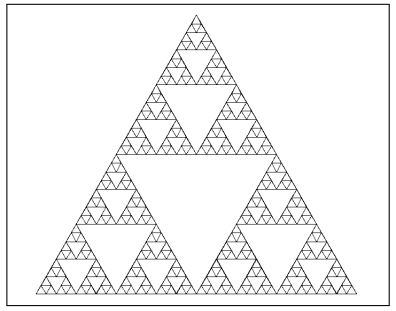
$n = 2$



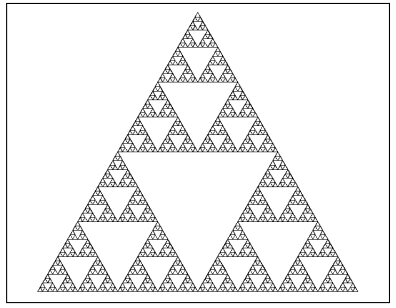
$n = 3$



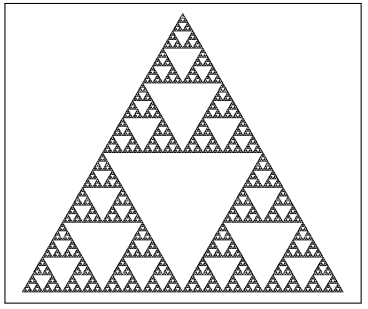
$n = 4$



$n = 5$



$n = 6$



$n = 7$

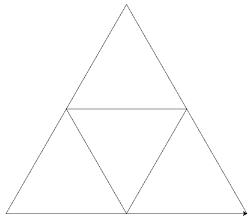
10.2.15 Tracé du Triangle de Sierpinsky par récursivité

• Programmation récursive : pour chacun des 3 triangles extérieurs :

Si `n > 0` : Effectuer 3 fois :
`sierp(longueur, n)` :

1. Tracé d'un triangle fils :
   (a) Appeler `sierp(longueur/2, n-1)`
   (b) Tourner à gauche de 120° : `tt.left(120)`
   (c) Appeler `sierp(longueur/2, n-1)`
   (d) Tourner à gauche de 120° : `tt.left(120)`
   (e) Appeler `sierp(longueur/2, n-1)`
2. Passer au triangle fils suivant en traçant un côté du triangle père :
   (a) Tourner à gauche de 120° : `tt.left(120)`
   (b) Avancer de `longueur` : `tt.forward(longueur)`

Tourner à gauche de 120°.



Un triangle "père" ses trois triangles "fils" sont les 3 "petits" triangles extérieurs.

Si `n == 0` : (cas terminal)

1. Avancer de `longueur` :

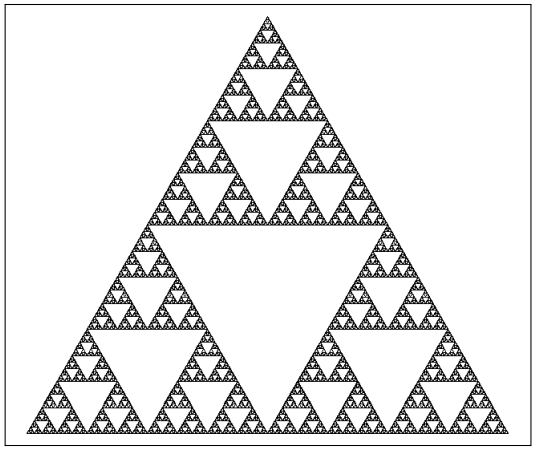
• Remarque : on trace séparément chacun des 3 premiers triangles fils pour éviter l'étape 2 (qui causerait un trait de trop : `tt.forward(longueur)`). On pourrait aussi ne pas effectuer l'étape 2 au premier appel.

10.2.16 Code

```
def sierp(longueur, n):
    if n==0:
        tt.forward(longueur)
    else:
        sierp(longueur/2, n-1); tt.left(120)
        sierp(longueur/2, n-1); tt.left(120)
        sierp(longueur/2, n-1); tt.left(120)
        tt.forward(longueur)
```

```
def triangle(longueur, n):
    tt.pen(speed=0)
    tt.hideturtle()
    tt.up()
    tt.goto(-longueur/2, -longueur/3)    # sommet bas/gauche
    tt.down()
    for i in range(3):
        sierp(longueur, n); tt.left(120)
```

```
>>> triangle(600,7)
```



10.3 Limitations

10.3.1 Suite de Fibonacci

10.3.2 Limitations de la programmation par récursivité

• Ecrivons deux versions, l'une itérative, l'autre récursive, d'une fonction retournant le terme de rang n de la suite de Fibonacci :

$$u_0 = 0, \quad u_1 = 1, \quad \forall\, n \in \mathbb{N},\ u_{n+2} = u_{n+1} + u_n$$

```
def fibo_iter(n):    # Version iterative
    if n==0:
        return 0
    u, v = 0, 1
    for i in range(n-1):
        u, v = v, u+v
    return v
```

```
def fibo_rec(n):    # Version récursive
    if n==0:
        return 0
    if n==1:
        return 1
    return fibo_rec(n-1) + fibo_rec(n-2)
```

10.3.3 Limitations : Suite de Fibonacci

Comparons leurs temps d'éxécution :

```
In [1]: %timeit fibo_iter(20)
100000 loops, best of 3: 1.98 µs per loop
In [2]: %timeit fibo_rec(20)
100 loops, best of 3: 4.6 ms per loop
```

Pour $n = 20$ La version itérative est plus de 2000 fois plus rapide !

```
In [3]: %timeit fibo_iter(30)
100000 loops, best of 3: 2.84 µs per loop
In [4]: %timeit fibo_rec(30)
1 loops, best of 3: 565 ms per loop
```

Pour $n = 30$ La version itérative est près de 200 000 fois plus rapide !

```
In [9]: %timeit fibo_iter(40)
100000 loops, best of 3: 3.68 µs per loop
In [8]: %timeit fibo_rec(40)
1 loops, best of 3: 1min 9s per loop
```

Pour $n = 40$ La version itérative est près de 20 millions de fois plus rapide !
Pour des valeurs de n plus grandes la version récursive ne répond plus !

10.3.4 Limitations. Explication.

Explication : Modifions le code pour qu'il affiche les rangs calculés :

```
def fiborec(n):
    if n==0:
        print('rang 0')
        return 0
    if n==1:
        print('rang 1')
    return 1
    print('rang',n)
    return fiborec(n-1) + fiborec(n-2)
```

```
In [19]: fiborec(4)
rang 4
rang 3
rang 2
rang 1
rang 0
rang 1
rang 2
rang 1
rang 0
```

10.3.5 Explication : usage de la mémoire

```
   >>> fiborec(4)
rangs : 4 - 3 - 2 - 1 - 0 - 1 - 2 - 1 - 0
```

Pile d'exécution
`fibo(4) = fibo(3) + fibo(2)`

```
  Appels : 4
   >>> fiborec(4)
rangs : 4 - 3 - 2 - 1 - 0 - 1 - 2 - 1 - 0
```

Pile d'exécution
`fibo(3)` `= fibo(2) + fibo(1)`
`fibo(4) =` **`fibo(3)`** `+ fibo(2)`

```
  Appels : 4 - 3
   >>> fiborec(4)
rangs : 4 - 3 - 2 - 1 - 0 - 1 - 2 - 1 - 0
```

Pile d'exécution
fibo(2) = fibo(1) + fibo(0)
fibo(3) = **fibo(2)** + fibo(1)
fibo(4) = fibo(3) + fibo(2)

```
   Appels : 4 - 3 - 2
    >>> fiborec(4)
rangs : 4 - 3 - 2 - 1 - 0 - 1 - 2 - 1 - 0
```

Pile d'exécution
fibo(1) = 1
fibo(2) = **fibo(1)** + fibo(0)
fibo(3) = fibo(2) + fibo(1)
fibo(4) = fibo(3) + fibo(2)

```
   Appels : 4 - 3 - 2 - 1
    >>> fiborec(4)
rangs : 4 - 3 - 2 - 1 - 0 - 1 - 2 - 1 - 0
```

Pile d'exécution
fibo(2) = **1** + fibo(0)
fibo(3) = fibo(2) + fibo(1)
fibo(4) = fibo(3) + fibo(2)

```
   Appels : 4 - 3 - 2 - 1
    >>> fiborec(4)
rangs : 4 - 3 - 2 - 1 - 0 - 1 - 2 - 1 - 0
```

Pile d'exécution
fibo(0) = 0
fibo(2) = 1 + **fibo(0)**
fibo(3) = fibo(2) + fibo(1)
fibo(4) = fibo(3) + fibo(2)

```
   Appels : 4 - 3 - 2 - 1 - 0
    >>> fiborec(4)
rangs : 4 - 3 - 2 - 1 - 0 - 1 - 2 - 1 - 0
```

Pile d'exécution
fibo(2) = 1
fibo(3) = **fibo(2)** + fibo(1)
fibo(4) = fibo(3) + fibo(2)

```
   Appels : 4 - 3 - 2 - 1 - 0
    >>> fiborec(4)
rangs : 4 - 3 - 2 - 1 - 0 - 1 - 2 - 1 - 0
```

Pile d'exécution
fibo(3) = **1** + fibo(1)
fibo(4) = fibo(3) + fibo(2)

```
   Appels : 4 - 3 - 2 - 1 - 0
    >>> fiborec(4)
rangs : 4 - 3 - 2 - 1 - 0 - 1 - 2 - 1 - 0
```

Pile d'exécution
fibo(1) = 1
fibo(3) = **1** + **fibo(1)**
fibo(4) = fibo(3) + fibo(2)

```
   Appels : 4 - 3 - 2 - 1 - 0 - 1
    >>> fiborec(4)
rangs : 4 - 3 - 2 - 1 - 0 - 1 - 2 - 1 - 0
```

Pile d'exécution
fibo(3) = **2**
fibo(4) = **fibo(3)** + fibo(2)

```
   Appels : 4 - 3 - 2 - 1 - 0 - 1
```

```
    >>> fiborec(4)
rangs : 4 - 3 - 2 - 1 - 0 - 1 - 2 - 1 - 0
```

Pile d'exécution
fibo(4) = **2** + fibo(2)

```
   Appels : 4 - 3 - 2 - 1 - 0 - 1 - 2
    >>> fiborec(4)
rangs : 4 - 3 - 2 - 1 - 0 - 1 - 2 - 1 - 0
```

Pile d'exécution
fibo(2) = fibo(1) + fibo(0)
fibo(4) = 2 + **fibo(2)**

```
   Appels : 4 - 3 - 2 - 1 - 0 - 1 - 2
    >>> fiborec(4)
rangs : 4 - 3 - 2 - 1 - 0 - 1 - 2 - 1 - 0
```

Pile d'exécution
fibo(1) = 1
fibo(2) = **fibo(1)** + fibo(0)
fibo(4) = 2 + fibo(2)

```
   Appels : 4 - 3 - 2 - 1 - 0 - 1 - 2 - 1
    >>> fiborec(4)
rangs : 4 - 3 - 2 - 1 - 0 - 1 - 2 - 1 - 0
```

Pile d'exécution
fibo(0) = 0
fibo(2) = **1** + **fibo(0)**
fibo(4) = 2 + fibo(2)

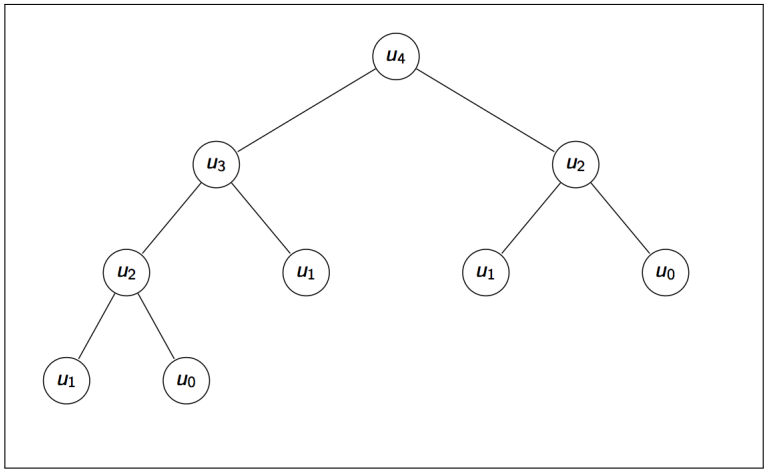
```
   Appels : 4 - 3 - 2 - 1 - 0 - 1 - 2 - 1 - 0
    >>> fiborec(4)
rangs : 4 - 3 - 2 - 1 - 0 - 1 - 2 - 1 - 0
```

Pile d'exécution
fibo(2) = 1
fibo(4) = 2 + **fibo(2)**

```
   Appels : 4 - 3 - 2 - 1 - 0 - 1 - 2 - 1 - 0
    >>> fiborec(4)
rangs : 4 - 3 - 2 - 1 - 0 - 1 - 2 - 1 - 0
```

Pile d'exécution
fibo(4) = **3**

```
Appels : 4 - 3 - 2 - 1 - 0 - 1 - 2 - 1 - 0
```



Tous les noeuds sont calculés... inutilement. L'arbre est parcouru "en profondeur d'abord" :

4 - 3 - 2 - 1 - 0 - 1 - 2 - 1 - 0

Complexité : C_0, $C_1 = 1$, $C(k+2) = C(k+1) + C(k) + O(1)$.
$\Longrightarrow C(k+2) \geqslant C(k+1) + C(k)$
$\Longrightarrow C(k) \geqslant \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^{k+1} - \left(\frac{1-\sqrt{5}}{2}\right)^{k+1}\right) \underset{+\infty}{\sim} \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{k+1}$.
La **complexité est exponentielle en temps**.
La **complexité est linéaire en espace** (le pile d'exécution atteint, mais ne dépasse jamais, n).

10.3.6 Résolution efficace par programmation dynamique

Pour résoudre efficacement par récursivité le calcul de la suite de Fibonacci :
Réserver un tableau `L` de capacité $n+1$ pour stocker les termes $u_0, u_1, \ldots, u_n$ de la suite. On le passera aussi en paramètre lors des appels récursifs :

```
def fiborecursif(n,L):    # Partie récursive
    print(n, end = ' ')    # Pour l'affichage
    if L[n] != None:    # Si déjà calculé pas d'appel récursif
        return L[n]    # et retour de sa valeur
    else :         # Sinon, appels récursifs et enregistrement
:
        L[n] = fiborecursif(n-1,L) + fiborecursif(n-2,L)
        return L[n]

def fibonacci(n):    # Partie Principale (Main)
    L = [0, 1] + [None]*(n-1)    # Création du tableau
    print('rangs : ', end = '')    # Pour l'affichage
    return fiborecursif(n,L)     # Appel récursif.
```

```
In[9]: fibonacci(4)
rangs : 4 3 2 1 0 1 2
In [10]: fibonacci(5)
rangs : 5 4 3 2 1 0 1 2 3
In[11]: fibonacci(6)
rangs : 6 5 4 3 2 1 0 1 2 3 4
```

COMPLEXITE LINEAIRE EN TEMPS (ET ESPACE) !...

```
In [12]: %timeit fibonacci(40)
10000 loops, best of 3: 24.2 µs per loop
```