

Fonctions récursives II

Exemples élaborés d'usage de la récursivité

BCPST - Lycée Fénélon

Le problème des tours d'Hanoï

Enoncé du problème

Résolution récursive

Code

Exemples fractals

Flocon de Von Koch

Graphiques avec le module `turtle`

Triangle de Sierpinsky

Limitations

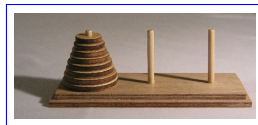
Suite de Fibonacci

Usage de la mémoire

Résolution efficace

Les tours de Hanoï

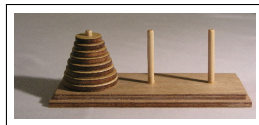
Le problème des tours de Hanoï est un jeu de réflexion inventé par le Mathématicien français Edouard Lucas en 1883 :



Des disque percés de diamètre décroissants sont empilés sur une colonne. Il y a trois colonnes.

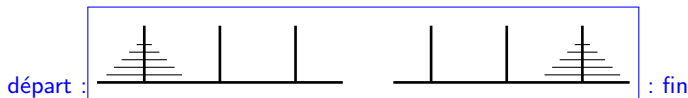
Les tours de Hanoï

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.



Les tours de Hanoï

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

Les tours de Hanoï

- 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 :

Les tours de Hanoï

- 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 :

Les tours de Hanoï

- 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.

Les tours de Hanoï

- 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.

Les tours de Hanoï

- 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.

Les tours de Hanoi

- On constitue une fonction :
`hanoi(n, Colonne_départ, Colonne_intermédiaire, Colonne_arrivée)`

Les tours de Hanoi

- 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)
```

Les tours de Hanoi

- 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, I, A, de capacités illimités (c'est à dire 3 listes en python),

Les tours de Hanoi

- 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()`

Les tours de Hanoi

- 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).

Les tours de Hanoi

- 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 = [ ]
```


Les tours de Hanoï

```
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)
```

Les tours de Hanoï

```
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
```

Les tours de Hanoï

```
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]
```

Les tours de Hanoï

- Deuxième tentative : pour afficher les états successifs.

Les tours de Hanoï

- Deuxième tentative : pour afficher les états successifs.

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

Les tours de Hanoï

- 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())`.

Les tours de Hanoï

- 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.

Les tours de Hanoï

- 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.

Les tours de Hanoï

- 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^{ère} **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)`.

Les tours de Hanoï

- 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^{ère} **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.

Les tours de Hanoï

```
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)
```

Les tours de Hanoï

2^{ème} **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()` :

Les tours de Hanoï

2^{ème} **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)
```

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]]
```

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]]
```

- **Remarque.** L'algorithme résout bien le problème des Tours d'Hanoï en un minimum de déplacements.

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]]
```

- **Remarque.** L'algorithme résout bien le problème des Tours d'Hanoï en un minimum de déplacements.

Indication : Par récurrence sur n .

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]]
```

- **Remarque.** L'algorithme résout bien le problème des Tours d'Hanoï en un minimum de déplacements.

Indication : Par récurrence sur n .

Pour résoudre le problème : tôt ou tard il faudra déplacer le disque de plus gros diamètre n .

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]]
```

- **Remarque.** L'algorithme résout bien le problème des Tours d'Hanoï en un minimum de déplacements.

Indication : 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.

Les tours de Hanoï

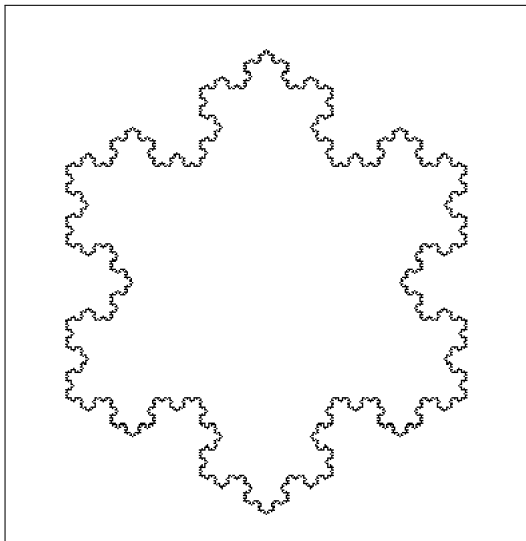
```
>>> resoudreHanoi(3)
[[3, 2, 1], [], []]
[[3, 2], [], [1]]
[[3], [2], [1]]
[[3], [2, 1], []]
[[], [2, 1], [3]]
[[1], [2], [3]]
[[1], [], [3, 2]]
[[], [], [3, 2, 1]]
```

- **Remarque.** L'algorithme résout bien le problème des Tours d'Hanoï en un minimum de déplacements.

Indication : Par récurrence sur n .

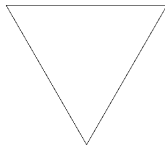
Pour résoudre le problème : tôt ou tard il faudra déplacer le disque de plus gros diamètre n . Pour cela nécessairement les $n - 1$ premiers disques devront être tous empilés dans le bon ordre sur une même colonne. **Pour que le nombre de déplacements soit minimal il faut que le disque n soit déplacé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

Flocon de Von Koch



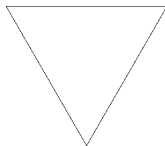
Flocon de Von Koch

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



Flocon de Von Koch

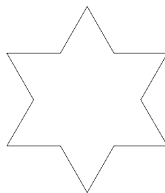
- Cette courbe est construite en partant d'un triangle équilatéral.



- 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.

Flocon de Von Koch

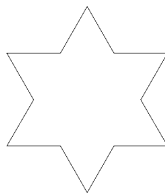
- Cette courbe est construite en partant d'un triangle équilatéral.



- 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.

Flocon de Von Koch

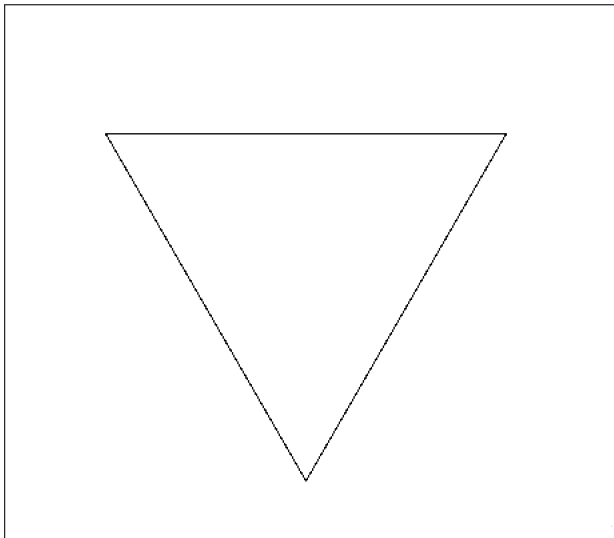
- Cette courbe est construite en partant d'un triangle équilatéral.



- 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.

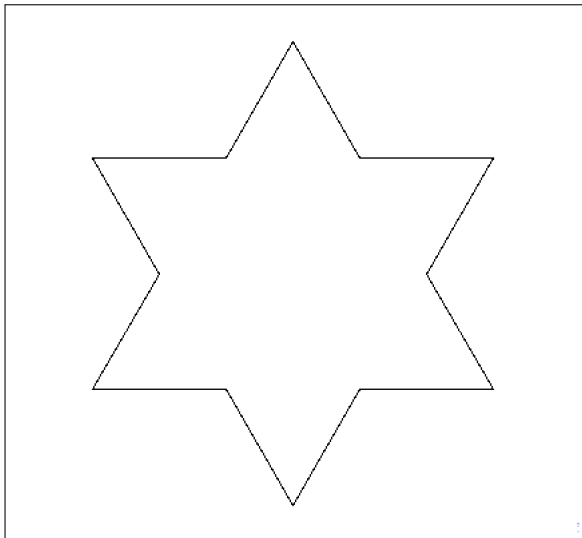
Flocon de Von Koch

$n = 1$



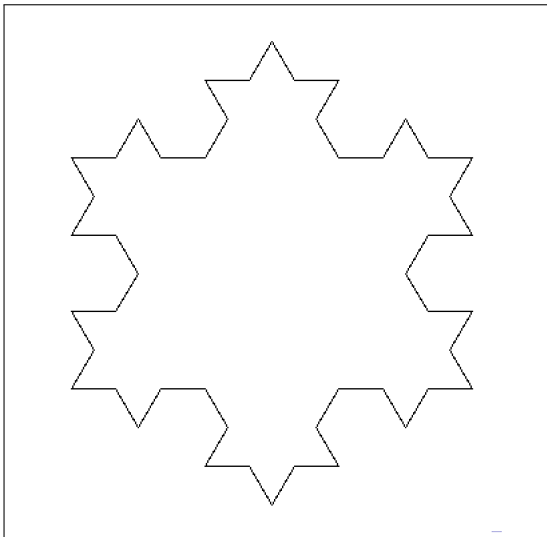
Flocon de Von Koch

$n = 2$



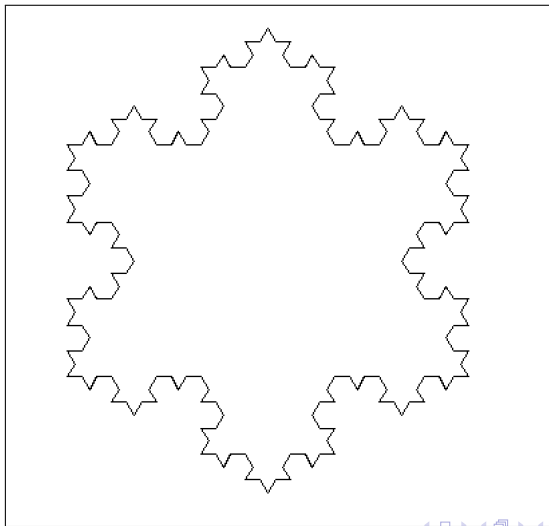
Flocon de Von Koch

$n = 3$



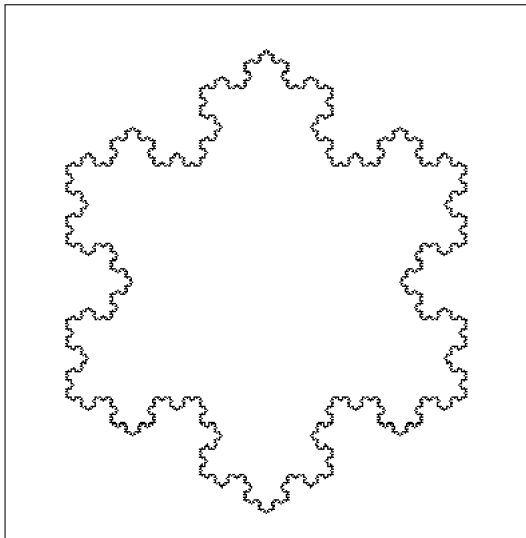
Flocon de Von Koch

$n = 4$



Flocon de Von Koch

$n = 6$



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.

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 :

<code>reset()</code>	Efface la fenêtre graphique, réinitialisation
<code>up()</code> , <code>down()</code>	Relève, abaisse le crayon
<code>forward(d)</code> , <code>backward(d)</code>	Avancer, reculer d'une distance <code>d</code>
<code>left(a)</code> , <code>right(a)</code>	Tourner à gauche, droite, d'une angle <code>a</code> en degrés

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 :

<code>reset()</code>	Efface la fenêtre graphique, réinitialisation
<code>up()</code> , <code>down()</code>	Relève, abaisse le crayon
<code>forward(d)</code> , <code>backward(d)</code>	Avancer, reculer d'une distance <code>d</code>
<code>left(a)</code> , <code>right(a)</code>	Tourner à gauche, droite, d'un angle <code>a</code> en degrés
<code>goto(x,y)</code>	Se déplace au point de coordonnées <code>(x,y)</code>
<code>position()</code>	Retourne la position courante

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 :

<code>reset()</code>	Efface la fenêtre graphique, réinitialisation
<code>up()</code> , <code>down()</code>	Relève, abaisse le crayon
<code>forward(d)</code> , <code>backward(d)</code>	Avancer, reculer d'une distance <code>d</code>
<code>left(a)</code> , <code>right(a)</code>	Tourner à gauche, droite, d'une angle <code>a</code> en degrés
<code>goto(x,y)</code>	Se déplace au point de coordonnées <code>(x,y)</code>
<code>position()</code>	Retourne la position courante
<code>color(couleur)</code>	Détermine la couleur = <code>'black'</code> , <code>'blue'</code> , <code>'red'</code> , ...
<code>width(l)</code>	Détermine l'épaisseur du trait <code>l</code>

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 :

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

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 :

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

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 :

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

- Voir l'aide en ligne :

<https://docs.python.org/3.4/library/turtle.html>

Le module turtle

- Fonction traçant un polygône :

Le module turtle

- 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)
```

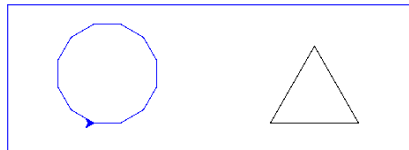
Le module turtle

- 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')
```



Flocon de Von Koch

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

`vonKoch(longueur,n)`

Flocon de Von Koch

- 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)`



Flocon de Von Koch

- 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)`



Flocon de Von Koch

- 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)`



Flocon de Von Koch

```
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)
```

Flocon de Von Koch

```
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
```

Flocon de Von Koch

```
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()
```

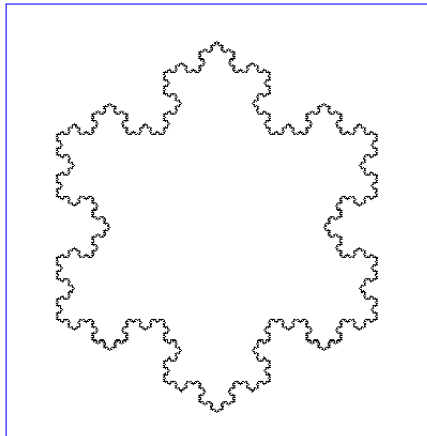
Flocon de Von Koch

```
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)
```

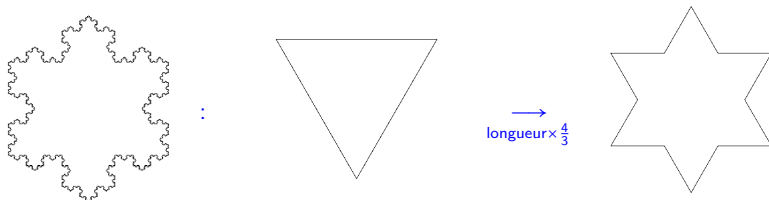
```
def floconVonKoch(longueur, n):  
    tt.pen(speed = 0)      # Accélération du mouvement  
    tt.hideturtle()        # Pour ne pas tracer la tortue  
    tt.up()  
    tt.goto(-longueur/2, longueur/3) # Départ en haut à gauche  
    tt.down()  
    for i in range(3):  
        vonkoch(longueur,n);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

Flocon de Von Koch

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



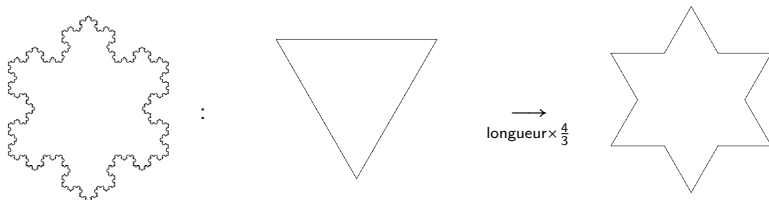
Flocon de Von Koch



- 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 :

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

Flocon de Von Koch



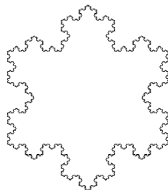
- Le flocon de Von Koch est la 'courbe fractale' obtenue à la limite.
- Toutes les courbes intermédiaires ont une longueur finie, terme de la suite géométrique :

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

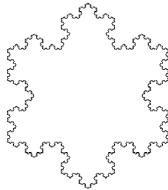
Le flocon est de longueur infinie : $\lim L_n = +\infty$.

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

Flocon de Von Koch

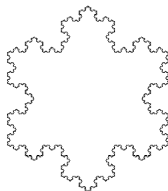


Flocon de Von Koch



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

Flocon de Von Koch

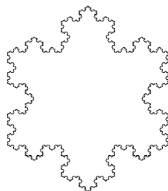


C'est une courbe de longueur infinie qui pourtant délimite un domaine d'aire finie :

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

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

Flocon de Von Koch



Longueur infinie



Aire finie.

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

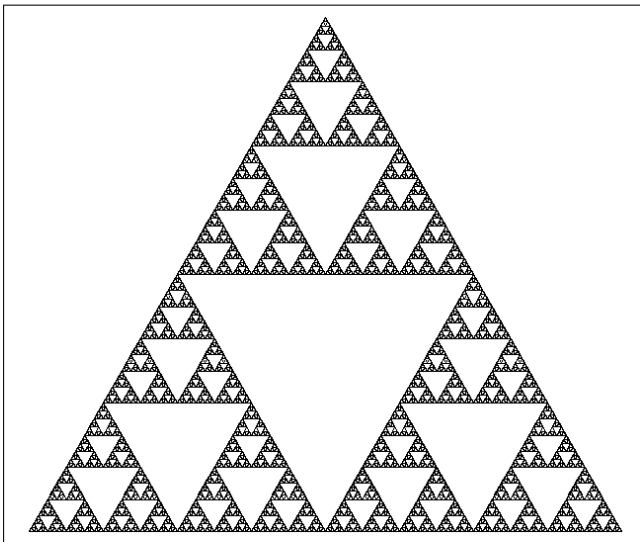
En effet : en posant L = longueur (paramètre initial) :

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

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

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

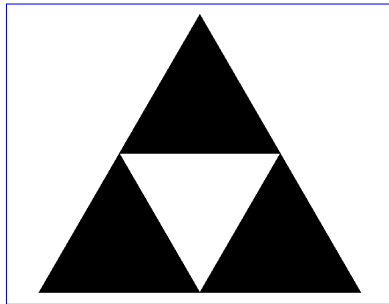
Triangle de Sierpinsky



Triangle de Sierpinsky

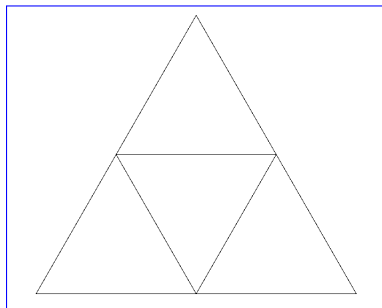
- Construction :

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.



Triangle de Sierpinsky

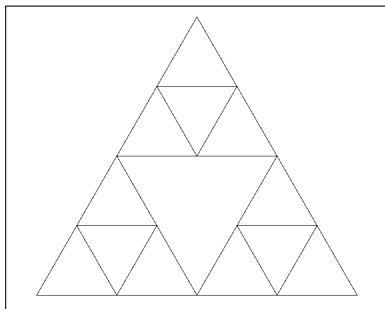
- On programmera plutôt :
 - Construire un triangle équilatéral de côtés de longueur l .
 - Tracer ses 3 médianes : elles découpent 4 triangles équilatéraux de côtés de longueur $l/2$.
 - Recommencer avec les 3 triangles extérieurs.



$n = 1$

Triangle de Sierpinsky

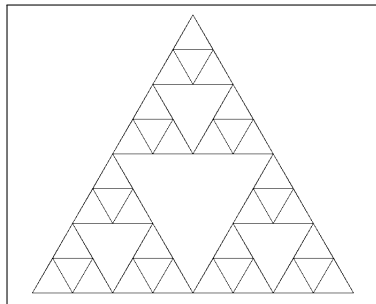
- On programmera plutôt :
 1. Construire un triangle équilatéral de côtés de longueur l .
 2. Tracer ses 3 médianes : elles découpent 4 triangles équilatéraux de côtés de longueur $l/2$.
 3. Recommencer avec les 3 triangles extérieurs.



$n = 2$

Triangle de Sierpinsky

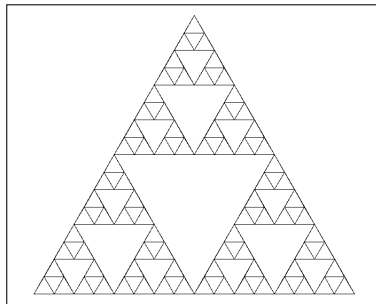
- On programmera plutôt :
 1. Construire un triangle équilatéral de côtés de longueur l .
 2. Tracer ses 3 médianes : elles découpent 4 triangles équilatéraux de côtés de longueur $l/2$.
 3. Recommencer avec les 3 triangles extérieurs.



$n = 3$

Triangle de Sierpinsky

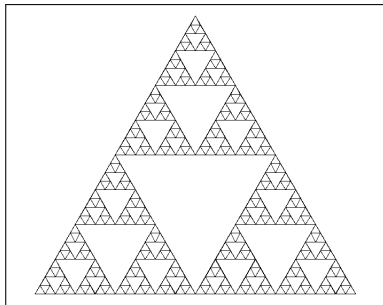
- On programmera plutôt :
 1. Construire un triangle équilatéral de côtés de longueur l .
 2. Tracer ses 3 médianes : elles découpent 4 triangles équilatéraux de côtés de longueur $l/2$.
 3. Recommencer avec les 3 triangles extérieurs.



$n = 4$

Triangle de Sierpinsky

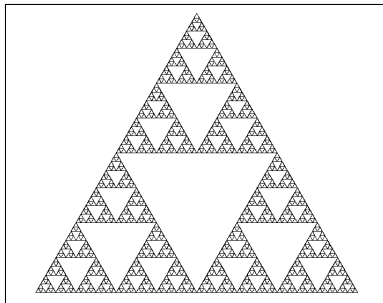
- On programmera plutôt :
 1. Construire un triangle équilatéral de côtés de longueur l .
 2. Tracer ses 3 médianes : elles découpent 4 triangles équilatéraux de côtés de longueur $l/2$.
 3. Recommencer avec les 3 triangles extérieurs.



$n = 5$

Triangle de Sierpinsky

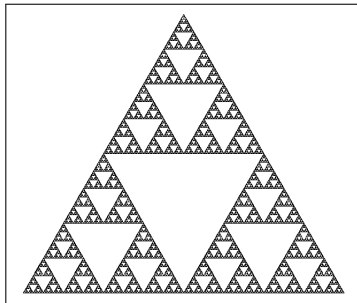
- On programmera plutôt :
 1. Construire un triangle équilatéral de côtés de longueur l .
 2. Tracer ses 3 médianes : elles découpent 4 triangles équilatéraux de côtés de longueur $l/2$.
 3. Recommencer avec les 3 triangles extérieurs.



$n = 6$

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.



$n = 7$

Triangle de Sierpinsky

- Programmation récursive : pour chacun des 3 triangles extérieurs :

Triangle de Sierpinsky

- Programmation récursive : pour chacun des 3 triangles extérieurs :

Si $n > 0$: Effectuer 3 fois :

`sierp(longueur, n)` :

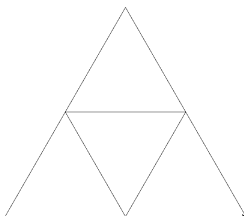
1. Tracé d'un triangle fils :

- 1.1 Appeler `sierp(longueur/2, n-1)`
- 1.2 Tourner à gauche de 120° : `tt.left(120)`
- 1.3 Appeler `sierp(longueur/2, n-1)`
- 1.4 Tourner à gauche de 120° : `tt.left(120)`
- 1.5 Appeler `sierp(longueur/2, n-1)`

2. Passer au triangle fils suivant en traçant un côté du triangle père :

- 2.1 Tourner à gauche de 120° : `tt.left(120)`
- 2.2 Avancer de longueur : `tt.forward(longueur)`

Tourner à gauche de 120° .



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

Triangle de Sierpinsky

- Programmation récursive : pour chacun des 3 triangles extérieurs :

`sierp(longueur, n) :`

Si $n > 0$:

1. Tracé d'un triangle fils :

- 1.1 Appeler `sierp(longueur/2., n-1)`
- 1.2 Tourner à gauche de 120° : `tt.left(120)`
- 1.3 Appeler `sierp(longueur/2., n-1)`
- 1.4 Tourner à gauche de 120° : `tt.left(120)`
- 1.5 Appeler `sierp(longueur/2., n-1)`

2. Passer au triangle fils suivant en traçant un côté du triangle père :

- 2.1 Tourner à gauche de 120° : `tt.left(120)`
- 2.2 Avancer de longueur : `tt.forward(longueur)`

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

1. Avancer de longueur : `tt.forward(longueur)`

Triangle de Sierpinsky

- Programmation récursive : pour chacun des 3 triangles extérieurs :

`sierp(longueur, n) :`

Si $n > 0$:

1. Tracé d'un triangle fils :

- 1.1 Appeler `sierp(longueur/2., n-1)`
- 1.2 Tourner à gauche de 120° : `tt.left(120)`
- 1.3 Appeler `sierp(longueur/2., n-1)`
- 1.4 Tourner à gauche de 120° : `tt.left(120)`
- 1.5 Appeler `sierp(longueur/2., n-1)`

2. Passer au triangle fils suivant en traçant un côté du triangle père :

- 2.1 Tourner à gauche de 120° : `tt.left(120)`
- 2.2 Avancer de longueur : `tt.forward(longueur)`

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

1. Avancer de longueur : `tt.forward(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.

Triangle de Sierpinsky

```
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)
```

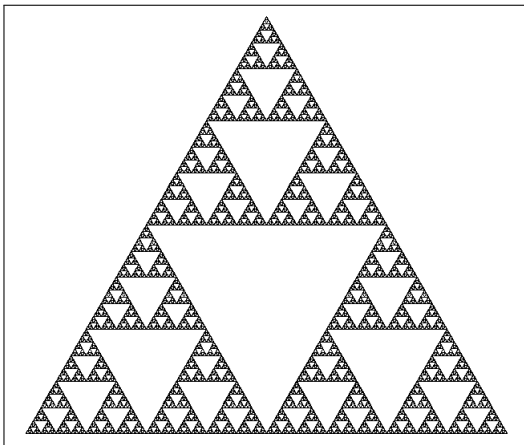
Triangle de Sierpinsky

```
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 de Sierpinsky

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



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 :

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

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 :

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

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


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 :

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

```
def fibo_iter(n):    # Version itérative
    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)
```

Limitations : Suite de Fibonacci

Comparons leurs temps d'exécution :

```
In [1]: %timeit fibo_iter(20)
```

100000 loops, best of 3: 1.98 μ s per loop

Limitations : Suite de Fibonacci

Comparons leurs temps d'exécution :

```
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 !

Limitations : Suite de Fibonacci

Comparons leurs temps d'exécution :

```
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
```

Limitations : Suite de Fibonacci

Comparons leurs temps d'exécution :

```
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 !

Limitations : Suite de Fibonacci

Comparons leurs temps d'exécution :

```
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
```

Limitations : Suite de Fibonacci

Comparons leurs temps d'exécution :

```
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 !

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
```


Explication : usage de la mémoire

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

Pile d'exécution
$\text{fib}(4) = \text{fib}(3) + \text{fib}(2)$

Appels : 4

Explication : usage de la mémoire

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

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

Appels : 4 - 3

Explication : usage de la mémoire

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

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

Appels : 4 - 3 - 2

Explication : usage de la mémoire

```
>>> 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

Explication : usage de la mémoire

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

Pile d'exécution
$\text{fib}(2) = 1 + \text{fib}(0)$
$\text{fib}(3) = \text{fib}(2) + \text{fib}(1)$
$\text{fib}(4) = \text{fib}(3) + \text{fib}(2)$

Appels : 4 - 3 - 2 - 1

Explication : usage de la mémoire

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

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

Appels : 4 - 3 - 2 - 1 - 0

Explication : usage de la mémoire

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

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

Appels : 4 - 3 - 2 - 1 - 0

Explication : usage de la mémoire

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

Pile d'exécution
$\text{fibonacci}(3) = 1 + \text{fibonacci}(1)$
$\text{fibonacci}(4) = \text{fibonacci}(3) + \text{fibonacci}(2)$

Appels : 4 - 3 - 2 - 1 - 0

Explication : usage de la mémoire

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

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

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

Explication : usage de la mémoire

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

Pile d'exécution
fib(3) = 2
fib(4) = fib(3) + fib(2)

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

Explication : usage de la mémoire

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

Pile d'exécution
$\text{fibo}(4) = 2 + \text{fibo}(2)$

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

Explication : usage de la mémoire

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

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

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

Explication : usage de la mémoire

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

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

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

Explication : usage de la mémoire

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

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

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

Explication : usage de la mémoire

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

Pile d'exécution
fib(2) = 1
fib(4) = 2 + fib(2)

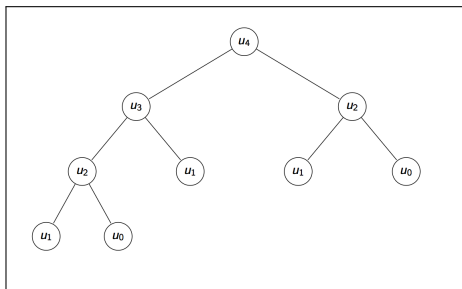
Appels : 4 - 3 - 2 - 1 - 0 - 1 - 2 - 1 - 0

Explication : usage de la mémoire

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

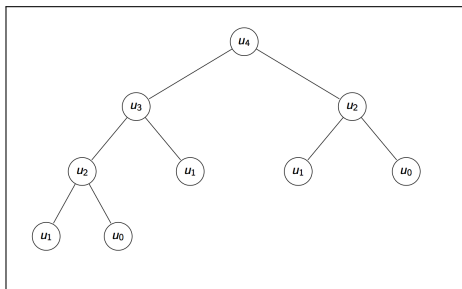
Pile d'exécution
fibonacci(4) = 3

```
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



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

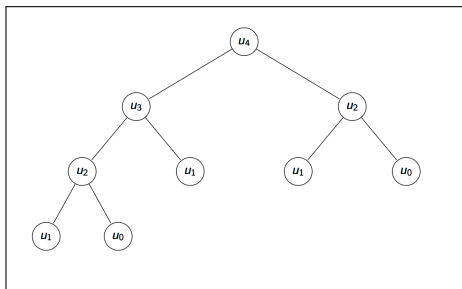
4 - 3 - 2 - 1 - 0 - 1 - 2 - 1 - 0

Complexité : $C_0, C_1 = 1, C(k+2) = C(k+1) + C(k) + O(1)$.

$$\Rightarrow C(k+2) \geq C(k+1) + C(k)$$

$$\Rightarrow C(k) \geq \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^{k+1} - \left(\frac{1-\sqrt{5}}{2} \right)^{k+1} \right) \underset{+\infty}{\sim} \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^{k+1}.$$

La complexité est exponentielle en temps.



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

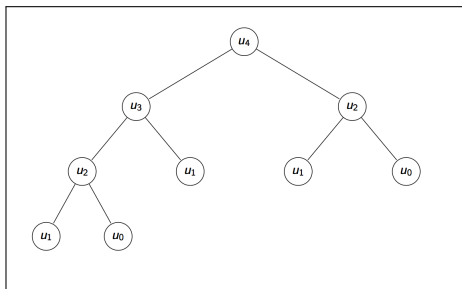
4 - 3 - 2 - 1 - 0 - 1 - 2 - 1 - 0

Complexité : $C_0, C_1 = 1, C(k+2) = C(k+1) + C(k) + O(1)$.

$$\implies C(k+2) \geq C(k+1) + C(k)$$

$$\implies C(k) \geq \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^{k+1} - \left(\frac{1-\sqrt{5}}{2} \right)^{k+1} \right) \underset{+\infty}{\sim} \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^{k+1}.$$

La complexité est exponentielle en temps.



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

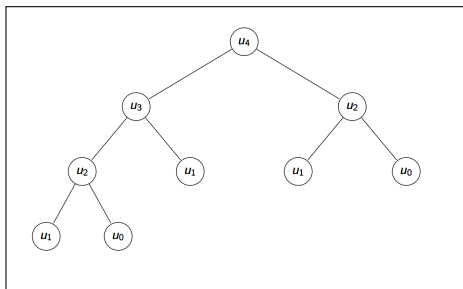
4 - 3 - 2 - 1 - 0 - 1 - 2 - 1 - 0

Complexité : $C_0, C_1 = 1, C(k+2) = C(k+1) + C(k) + O(1)$.

$$\implies C(k+2) \geq C(k+1) + C(k)$$

$$\implies C(k) \geq \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^{k+1} - \left(\frac{1-\sqrt{5}}{2} \right)^{k+1} \right) \underset{+\infty}{\sim} \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^{k+1}.$$

La complexité est exponentielle en temps.



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

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

Complexité : $C_0, C_1 = 1, C(k+2) = C(k+1) + C(k) + O(1)$.

$$\implies C(k+2) \geq C(k+1) + C(k)$$

$$\implies C(k) \geq \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^{k+1} - \left(\frac{1-\sqrt{5}}{2} \right)^{k+1} \right) \underset{+\infty}{\sim} \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^{k+1}.$$

La **complexité est exponentielle en temps**.

La **complexité est linéaire en espace** (le pile d'exécution atteint, mais ne dépasse jamais, n).

Résolution efficace

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

Résolution efficace

Pour résoudre efficacement par récursivité le calcul de la suite de Fibonacci :
Réserver un tableau L de capacité $n + 1$ pour stocker les termes $u_0, u_1, \dots, u_n$ 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 de la partie récursive.
```

Résolution efficace

```
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
```


Résolution efficace

```
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
```