

Bases d'algorithmique et de programmation

Sébastien Gagné, Université d'Orléans

L1 Algo - Prog - S1

Cours 07 - Introduction à la récursivité

Plan du cours

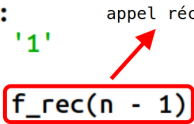
- Les fonctions récursives
- Structure récursive
- Exemples classiques
- La récursivité terminale (ou non)
- Notions d'efficacité
- Exemples graphiques

Les fonctions récursives

- Une fonction **récursive** est une fonction qui s'appelle elle-même au cours de son exécution ;
- Tous les langages de programmation modernes offrent la possibilité de définir des fonctions récursives ;

```
1 def f_rec(n) :  
2     if n == 1 :  
3         return '1'  
4     else :  
5         return f_rec(n - 1) + str(n)  
6  
7 n = int(input("n = ? "))  
8 resultat = f_rec(n)  
9 print(f"f_rec({n}) = {resultat}")
```

appel récursif



Les fonctions récursives

- Détails du calcul de `f_rec(6)`

`f_rec(6)` → `f_rec(5) + '6'`

→ `f_rec(4) + '5' + '6'`

→ `f_rec(3) + '4' + '5' + '6'`

→ `f_rec(2) + '3' + '4' + '5' + '6'`

→ `f_rec(1) + '2' + '3' + '4' + '5' + '6'`

→ `'1' + '2' + '3' + '4' + '5' + '6'`

→ `'123456'`

```
def f_rec(n) :  
    if n == 1 :  
        return '1'  
    else :  
        return f_rec(n - 1) + str(n)
```

```
n = ? 6  
f_rec(6) = 123456
```

Structure récursive

- Une fonction récursive comprend deux points essentiels :
 - ▶ Le point d'arrêt qui permet de rompre l'appel récursif ;
 - ▶ L'instruction récursive qui transmet l'appel récursif ;

```
def f_rec(n) :  
    if n == 1 :  
        return '1' } point d'arrêt  
    else :  
        return f_rec(n - 1) + str(n) } instruction récursive
```

- Ces deux points correspondent aux phases d'initialisation et de transmission du raisonnement par récurrence.

récursive ou itérative ?

- Une fonction récursive a toujours un équivalent **itératif** ;
- La fonction **itérative** ne s'appelle pas elle-même. Elle transforme l'appel récursif en une boucle répétitive ;

```
1 def f_ite(n) :  
2     chaîne = ''  
3     for i in range(1, n+1) :  
4         chaîne = chaîne + str(i)  
5     return chaîne  
6  
7 n = int(input("n = ? "))  
8 resultat = f_ite(n)  
9 print(f"f_ite({n}) = {resultat}")
```

```
n = ? 6  
f_ite(6) = 123456
```

trouvez le défaut !

```
def f_rec(n) :  
    if n == 1 :  
        return '1'  
    else :  
        return f_rec(n - 1) + str(n)
```

Cette fonction attend un entier de la part de l'utilisateur.

- Quel(s) entier(s) peut-on donner à la fonction pour la faire "planter" ? **0, -1, ou tout entier inférieur à 1**
- Comment décrire plus précisément ce "plantage" ? **la fonction s'appelle à l'infini (boucle infinie)**

Exemple classique - factorielle

- La **factorielle** d'un entier n positif est le produit de tous les entiers entre 1 et n et est notée $n!$
- On convient pour des questions de cohérence que $0! = 1$
- On définit facilement une fonction factorielle **itérative** :

```
1 def fact_ite(n) :  
2     resultat = 1  
3     for i in range(1, n+1) :  
4         resultat = resultat * i  
5     return resultat  
6  
7 n = int(input("n = ? "))  
8 resultat = fact_ite(n)  
9 print(f"fact_ite({n}) = {resultat}")
```

```
n = ? 5  
fact_ite(5) = 120
```

Exemple classique - factorielle

- $n! = n \times (n - 1) \times \dots \times 1$ donc $n! = n \times (n - 1)!$
- La définition récursive de la factorielle tient donc en deux points essentiels :
 - ▶ Le point d'arrêt pour $n = 0$: $0! = 1$
 - ▶ L'instruction récursive pour $n \geq 1$: $n! = n \times (n - 1)!$

```
1 def fact_rec(n) :  
2     if n == 0 :  
3         return 1 } point d'arrêt  
4     else :  
5         return n * fact_rec(n-1) } instruction récursive
```

Exemple classique - factorielle

`n = fact_rec(5)`

pile d'appels de
la fonction

calcul demandé
lors de l'appel

résultat de ce
calcul

Exemple classique - factorielle

`n = fact_rec(5)`

`fact_rec(5)`

pile d'appels de
la fonction

calcul demandé
lors de l'appel

résultat de ce
calcul

Exemple classique - factorielle

$n = \text{fact_rec}(5)$

$\text{fact_rec}(5)$

$5 * \text{fact_rec}(4)$

pile d'appels de
la fonction

calcul demandé
lors de l'appel

résultat de ce
calcul

Exemple classique - factorielle

$n = \text{fact_rec}(5)$

fact_rec(4)

fact_rec(5)

$5 * \text{fact_rec}(4)$

pile d'appels de
la fonction

calcul demandé
lors de l'appel

résultat de ce
calcul

Exemple classique - factorielle

$n = \text{fact_rec}(5)$

$\text{fact_rec}(4)$

$4 * \text{fact_rec}(3)$

$\text{fact_rec}(5)$

$5 * \text{fact_rec}(4)$

pile d'appels de
la fonction

calcul demandé
lors de l'appel

résultat de ce
calcul

Exemple classique - factorielle

$n = \text{fact_rec}(5)$

fact_rec(3)

fact_rec(4)

fact_rec(5)

$4 * \text{fact_rec}(3)$

$5 * \text{fact_rec}(4)$

pile d'appels de
la fonction

calcul demandé
lors de l'appel

résultat de ce
calcul

Exemple classique - factorielle

$n = \text{fact_rec}(5)$

fact_rec(3)

$3 * \text{fact_rec}(2)$

fact_rec(4)

$4 * \text{fact_rec}(3)$

fact_rec(5)

$5 * \text{fact_rec}(4)$

pile d'appels de
la fonction

calcul demandé
lors de l'appel

résultat de ce
calcul

Exemple classique - factorielle

$n = \text{fact_rec}(5)$

fact_rec(2)

fact_rec(3)

$3 * \text{fact_rec}(2)$

fact_rec(4)

$4 * \text{fact_rec}(3)$

fact_rec(5)

$5 * \text{fact_rec}(4)$

pile d'appels de
la fonction

calcul demandé
lors de l'appel

résultat de ce
calcul

Exemple classique - factorielle

$n = \text{fact_rec}(5)$

fact_rec(2)

$2 * \text{fact_rec}(1)$

fact_rec(3)

$3 * \text{fact_rec}(2)$

fact_rec(4)

$4 * \text{fact_rec}(3)$

fact_rec(5)

$5 * \text{fact_rec}(4)$

pile d'appels de
la fonction

calcul demandé
lors de l'appel

résultat de ce
calcul

Exemple classique - factorielle

$n = \text{fact_rec}(5)$

fact_rec(1)

fact_rec(2)

$2 * \text{fact_rec}(1)$

fact_rec(3)

$3 * \text{fact_rec}(2)$

fact_rec(4)

$4 * \text{fact_rec}(3)$

fact_rec(5)

$5 * \text{fact_rec}(4)$

pile d'appels de
la fonction

calcul demandé
lors de l'appel

résultat de ce
calcul

Exemple classique - factorielle

$n = \text{fact_rec}(5)$

fact_rec(1)	$1 * \text{fact_rec}(0)$	
fact_rec(2)	$2 * \text{fact_rec}(1)$	
fact_rec(3)	$3 * \text{fact_rec}(2)$	
fact_rec(4)	$4 * \text{fact_rec}(3)$	
fact_rec(5)	$5 * \text{fact_rec}(4)$	
pile d'appels de la fonction	calcul demandé lors de l'appel	résultat de ce calcul

Exemple classique - factorielle

n = fact_rec(5)		
fact_rec(0)		
fact_rec(1)	1 * fact_rec(0)	
fact_rec(2)	2 * fact_rec(1)	
fact_rec(3)	3 * fact_rec(2)	
fact_rec(4)	4 * fact_rec(3)	
fact_rec(5)	5 * fact_rec(4)	
pile d'appels de la fonction	calcul demandé lors de l'appel	résultat de ce calcul

Exemple classique - factorielle

n = fact_rec(5)		
fact_rec(0)		1
fact_rec(1)	1 * fact_rec(0)	
fact_rec(2)	2 * fact_rec(1)	
fact_rec(3)	3 * fact_rec(2)	
fact_rec(4)	4 * fact_rec(3)	
fact_rec(5)	5 * fact_rec(4)	
pile d'appels de la fonction	calcul demandé lors de l'appel	résultat de ce calcul

Exemple classique - factorielle

$n = \text{fact_rec}(5)$

fact_rec(1)	$1 * \text{fact_rec}(0)$	$1 * 1 = 1$
fact_rec(2)	$2 * \text{fact_rec}(1)$	
fact_rec(3)	$3 * \text{fact_rec}(2)$	
fact_rec(4)	$4 * \text{fact_rec}(3)$	
fact_rec(5)	$5 * \text{fact_rec}(4)$	
pile d'appels de la fonction	calcul demandé lors de l'appel	résultat de ce calcul

Exemple classique - factorielle

$n = \text{fact_rec}(5)$

fact_rec(2)	$2 * \text{fact_rec}(1)$	$2 * 1 = 2$
fact_rec(3)	$3 * \text{fact_rec}(2)$	
fact_rec(4)	$4 * \text{fact_rec}(3)$	
fact_rec(5)	$5 * \text{fact_rec}(4)$	
pile d'appels de la fonction	calcul demandé lors de l'appel	résultat de ce calcul

Exemple classique - factorielle

$n = \text{fact_rec}(5)$

$\text{fact_rec}(3)$

$3 * \text{fact_rec}(2)$

$3 * 2 = 6$

$\text{fact_rec}(4)$

$4 * \text{fact_rec}(3)$

$\text{fact_rec}(5)$

$5 * \text{fact_rec}(4)$

pile d'appels de
la fonction

calcul demandé
lors de l'appel

résultat de ce
calcul

Exemple classique - factorielle

$n = \text{fact_rec}(5)$

$\text{fact_rec}(4)$

$4 * \text{fact_rec}(3)$

$4 * 6 = 24$

$\text{fact_rec}(5)$

$5 * \text{fact_rec}(4)$

pile d'appels de
la fonction

calcul demandé
lors de l'appel

résultat de ce
calcul

Exemple classique - factorielle

$n = \text{fact_rec}(5)$

$\text{fact_rec}(5)$

pile d'appels de
la fonction

$5 * \text{fact_rec}(4)$

calcul demandé
lors de l'appel

$5 * 24 = 120$

résultat de ce
calcul

Exemple classique - factorielle

$$f(5) = 5 * f(4)$$

$$f(4) = 4 * f(3)$$

$$f(3) = 3 * f(2)$$

$$f(2) = 2 * f(1)$$

$$f(1) = 1 * f(0)$$

$$f(0) = 1$$

$$f(1) = 1 * 1 = 1$$

$$f(2) = 2 * 1 = 2$$

$$f(3) = 3 * 2 = 6$$

$$f(4) = 4 * 6 = 24$$

$$f(5) = 5 * 24 = 120$$

Autre exemple - somme de 1 à n

- On doit calculer $S_n = 1 + \dots + n$ pour $n \geq 1$
- La valeur initiale est $S_1 = 1$
- La relation de transition : $S_n = S_{n-1} + n$

```
1 def somme_rec(n) :  
2     if n == 1 :  
3         return 1  
4     else :  
5         return somme_rec(n-1) + n
```

```
1 def somme_ite(n) :  
2     resultat = 0  
3     for i in range(1, n+1) :  
4         resultat = resultat + i  
5     return resultat
```

Réversivité terminale (ou non)

- On parle de **réversivité terminale** quand l'instruction d'appel récursif ne fait intervenir que l'appel de la fonction, rien d'autre ;
- Les 3 exemples précédents ont un point commun : ce sont des exemples de **réversivité non terminale** ;
- En effet, pour le cas de la factorielle vu précédemment, l'instruction d'appel récursif contient, **en plus de l'appel de la fonction récursive**, une multiplication par n .

```
1 def fact_rec(n) :  
2     if n == 0 :  
3         return 1  
4     else :  
5         return n * fact_rec(n-1)
```

Récurtivité terminale

- Voici le code d'une fonction récursive terminale : dans l'appel récursif, il n'y a que l'appel à la fonction récursive, rien d'autre.

```
1 def f(n, acc) :  
2     if n == 0 :  
3         return acc  
4     else :  
5         return f(n - 1, acc * n)
```

- La particularité de cette fonction récursive est qu'elle prend deux arguments : n et acc ;
- Que calcule la fonction ? Que vaut par exemple $f(5, 1)$?

Récurtivité terminale

```
1 def f(n, acc) :  
2     if n == 0 :  
3         return acc  
4     else :  
5         return f(n - 1, acc * n)
```

Voici la suite d'appels

f(5, 1) =

Récurtivité terminale

```
1 def f(n, acc) :  
2     if n == 0 :  
3         return acc  
4     else :  
5         return f(n - 1, acc * n)
```

Voici la suite d'appels

$f(5, 1) = f(4, 5) =$

Récurtivité terminale

```
1 def f(n, acc) :  
2     if n == 0 :  
3         return acc  
4     else :  
5         return f(n - 1, acc * n)
```

Voici la suite d'appels

$f(5, 1) = f(4, 5) = f(3, 20) =$

Récurtivité terminale

```
1 def f(n, acc) :  
2     if n == 0 :  
3         return acc  
4     else :  
5         return f(n - 1, acc * n)
```

Voici la suite d'appels

$$f(5, 1) = f(4, 5) = f(3, 20) = f(2, 60) =$$

Récurtivité terminale

```
1 def f(n, acc) :  
2     if n == 0 :  
3         return acc  
4     else :  
5         return f(n - 1, acc * n)
```

Voici la suite d'appels

$$f(5, 1) = f(4, 5) = f(3, 20) = f(2, 60) = f(1, 120) =$$

Récurtivité terminale

```
1 def f(n, acc) :  
2     if n == 0 :  
3         return acc  
4     else :  
5         return f(n - 1, acc * n)
```

Voici la suite d'appels

$$f(5, 1) = f(4, 5) = f(3, 20) = f(2, 60) = f(1, 120) = f(0, 120) =$$

Récurtivité terminale

```
1 def f(n, acc) :  
2     if n == 0 :  
3         return acc  
4     else :  
5         return f(n - 1, acc * n)
```

Voici la suite d'appels

$$f(5, 1) = f(4, 5) = f(3, 20) = f(2, 60) = f(1, 120) = f(0, 120) = 120$$

Récurtivité terminale

- Le paramètre *acc* est un **accumulateur** ;
- Il contient, d'appel en appel, la construction du résultat final, en accumulant les multiplications successives ;
- Les valeurs successives de l'accumulateur *acc* sont donc :
 - ▶ 1 ;
 - ▶ 5×1 , donc 5 ;
 - ▶ $4 \times 5 \times 1$, donc 20 ;
 - ▶ $3 \times 4 \times 5 \times 1$, donc 60 ;
 - ▶ $2 \times 3 \times 4 \times 5 \times 1$, donc 120 ;
- la valeur finale de l'accumulateur est donc $5! = 120$

Récurtivité terminale

- Voici le code d'une autre fonction récursive terminale.
Que calcule cette fonction ? Que vaut $g(6, 0)$?

```
1 def g(n, acc) :  
2     if n == 0 :  
3         return acc  
4     else :  
5         return g(n - 1, acc + n)
```

Récurtivité terminale

- Voici le code d'une autre fonction récursive terminale.
Que calcule cette fonction ? Que vaut $g(6, 0)$?

```
1 def g(n, acc) :  
2     if n == 0 :  
3         return acc  
4     else :  
5         return g(n - 1, acc + n)
```

- Voici la suite des appels récursifs

$$g(6, 0) = g(5, 6) = g(4, 11) = g(3, 15) = g(2, 18) = g(1, 20) = g(0, 21) = 21$$

Premier bilan

- Il peut être intéressant d'utiliser le point de vue récursif car il propose une façon économique et élégante de coder ;
- Le codage récursif induit une "profondeur" d'exécution liée à la suite d'appels récursifs et de retours. Cette profondeur est l'aspect compliqué de la récursivité ;
- Cette profondeur est à maîtriser car le nombre d'appels peut devenir rapidement très grand (voir fibonacci...) et ingérable en temps limité ;
- Une fonction récursive terminale présente une suite de retours évidente : c'est le même résultat qui "remonte". On peut la réécrire en une fonction itérative équivalente ;

Notions d'efficacité

- La profondeur d'exécution peut cacher certains aspects qui influent sur l'efficacité de l'algorithme ;
- Il est important de maîtriser le nombre d'appels récursifs : un code élégant mais inefficace n'est pas à retenir ;
- Une illustration classique : la suite de Fibonacci traitée de manière intuitive en récursif.

$$f_n = \begin{cases} 0, & \text{si } n = 0 \\ 1, & \text{si } n = 1 \\ f_{n-1} + f_{n-2} & \text{si } n \geq 2 \end{cases}$$

- Cette définition donne un double appel récursif.

Notions d'efficacité

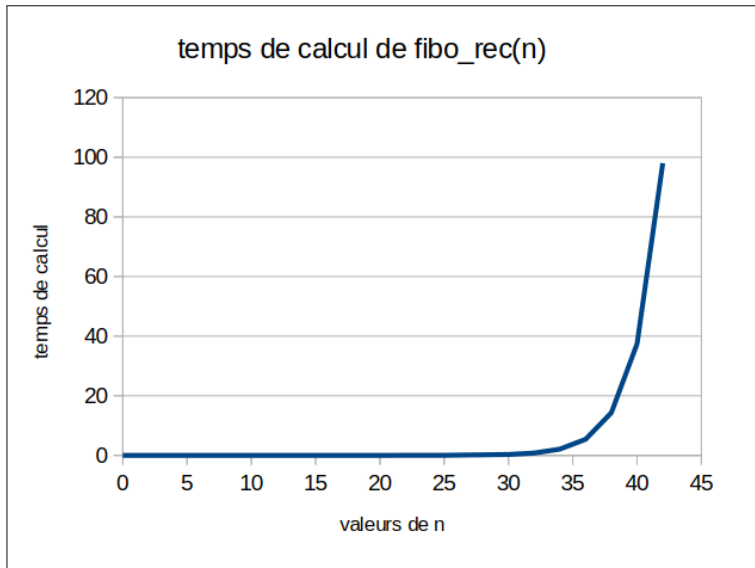
```
1 def fibo_rec(n) :  
2     if n == 0 :  
3         return 0  
4     elif n == 1 :  
5         return 1  
6     else :  
7         return fibo_rec(n-1) + fibo_rec(n-2)
```

code élégant mais pas efficace

```
1 def fibo_ite(n) :  
2     a = 0  
3     b = 1  
4     for i in range(n) :  
5         c = a + b; a = b; b = c  
6     return a
```

code moins parlant mais efficace

Notions d'efficacité



Notions d'efficacité

- La courbe a une allure exponentielle ;
- pour $fibonacci_rec(50)$: 1h 30 min de calcul ;
- pour $fibonacci_rec(60)$: 150 heures de calcul ;
- pour $fibonacci_rec(100)$: 4 millions d'années de calcul ;
- Cette croissance exponentielle vient du "gaspillage" d'appels de f_1 et de f_0 pour tous les calculs ;

Exemples graphiques

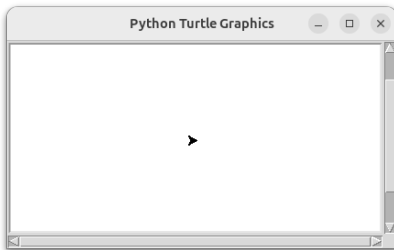
- Le module turtle de python permet d'écrire des algorithmes graphiques, dont l'exécution donne un dessin ;
- Ces algorithmes graphiques peuvent être itératifs, mais aussi récurifs ;
- Ils utilisent un objet python de type Turtle, par l'appel au constructeur Turtle() du module turtle ;
- Ces objets de type Turtle ont des méthodes spécifiques qui construisent une trace sur l'interface graphique ;

Construction d'objets Turtle

- On construit un objet de type `turtle.Turtle`

```
>>> import turtle  
>>> t = turtle.Turtle()  
>>> type(t)  
<class 'turtle.Turtle'>
```

- Cet objet `t`, **instance** de la **classe** `Turtle` du module `turtle`, est visualisé dans une fenêtre graphique.



Méthodes d'un objet Turtle

Les méthodes importantes d'un objet de type Turtle sont :

```
import turtle
t = Turtle()

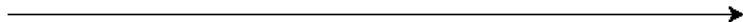
t.forward(100)    # avance de 100 px dans sa direction actuelle
t.left(50)        # tourne de 50° vers la gauche en restant sur place
t.right(60)       # tourne de 60° vers la droite en restant sur place
t.up()           # lève le crayon traceur
t.down()          # baisse le crayon traceur
t.dot()           # marque un point à l'emplacement actuel
t.goto(200, 100)  # va à l'emplacement (200, 100)
t.reset()         # retourne à l'emplacement (0, 0)
```

Courbe de *von Koch*

C'est une ligne polygonale obtenue par un processus récursif :

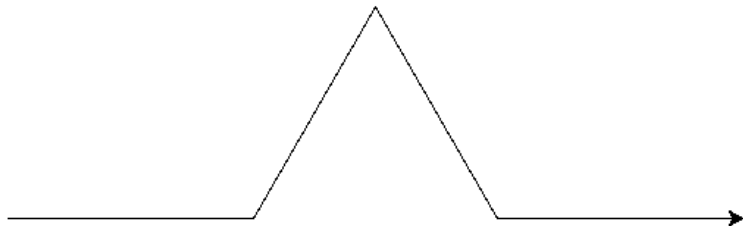
- En génération 1, on trace un segment de longueur c ;
- Pour passer d'une génération n à la suivante $n + 1$, on applique à chaque segment du dessin de la génération n l'algorithme suivant :
 - ▶ On divise le segment en 3 segments de même longueur ;
 - ▶ On construit, sur le segment du milieu qu'on efface, les deux autres côtés d'un triangle équilatéral ;

Courbe de *von Koch*



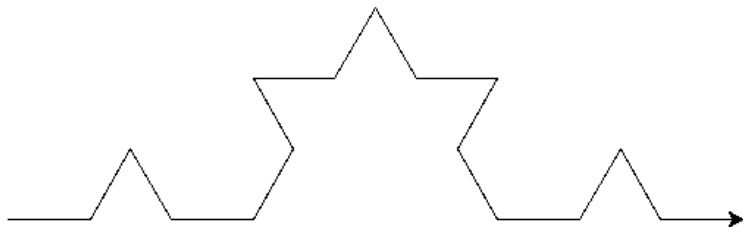
génération 1

Courbe de *von Koch*



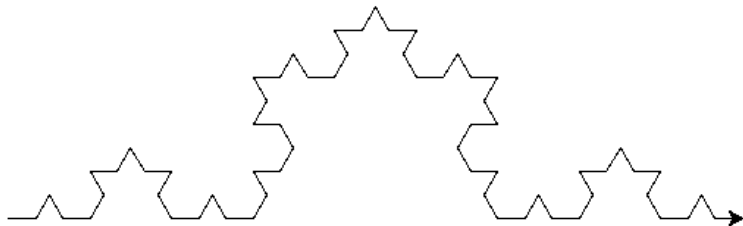
génération 2

Courbe de *von Koch*



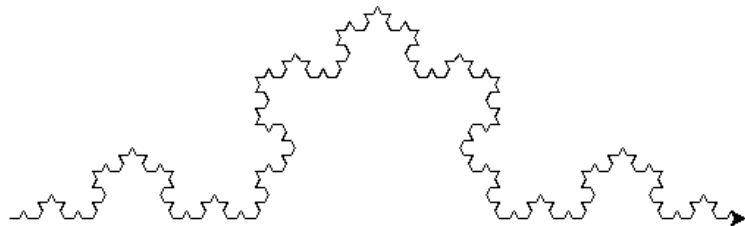
génération 3

Courbe de *von Koch*



génération 4

Courbe de *von Koch*



génération 5

flocon de von Koch

```
1 import turtle
2
3 def cote(n, c) :
4     if n == 1 :
5         t.forward(c)
6     else :
7         cote(n-1, c/3)
8         t.left(60)
9         cote(n-1, c/3)
10        t.right(120)
11        cote(n-1, c/3)
12        t.left(60)
13        cote(n-1, c/3)
14
15 def flocon(n, c) :
16     for i in range(3) :
17         cote(n, c)
18         t.right(120)
19
20 c = float(input("côté de base ? "))
21 n = int(input("quelle génération ? "))
22 t = turtle.Turtle()
23 t.up(); t.goto(-220, 100); t.down(); t.speed("fastest")
24
25 flocon(n, c)
```

} point d'arrêt

} instructions récursives

flocon de *von Koch*

```
>>> %Run von_koch.py  
côté de base ? 400  
quelle génération ? 5
```

