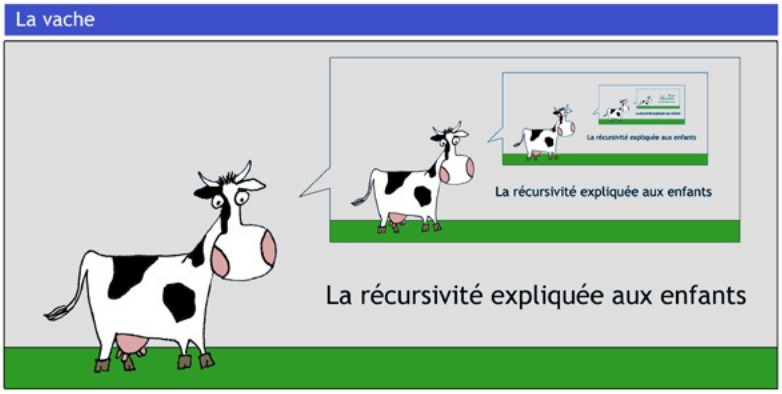
Chapitre 7

Récursivité et fractales

7.1. La récursivité

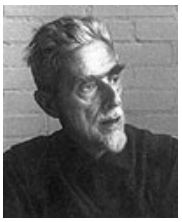


En mathématiques, en informatique, en biologie, mais aussi dans notre quotidien, nous faisons souvent face à des situations où un problème doit être résolu en utilisant une méthode de résolution qui est répétée plusieurs fois. Dans l'*itération*, cette méthode est appliquée par paliers de façon séquentielle, dans la *récursivité*, la méthode s'appelle elle-même.

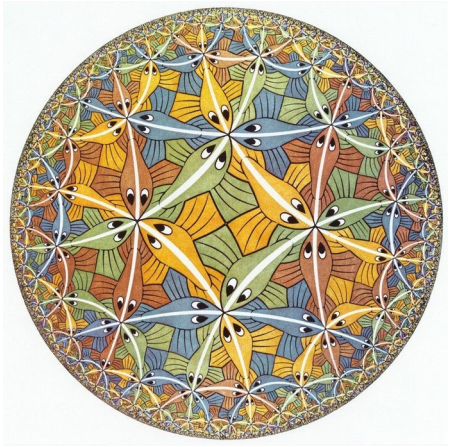
La récursivité est un principe de pensée exigeant et est souvent désigné comme « trop compliqué ». La récursivité est cependant si fondamentale qu'il n'est pas possible de l'éviter.

7.1.1. Dans les arts

Dans le domaine des arts, c'est l'artiste néerlandais Maurits Cornelis **Escher** qui fait le plus grand usage de la récursivité. Il y a d'autres exemples, notamment le Triptyque Stefaneschi, de **Giotto** (1266-1337), visible au musée du Vatican.

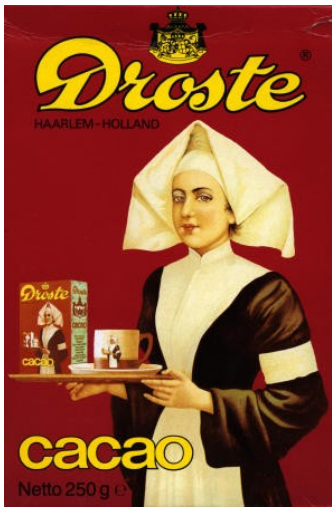


M. C. **Escher**
(1898 - 1972)



De son côté, la publicité a aussi utilisé la mise en abyme, rendant célèbres les fromages « La vache qui rit », le vermouth « Dubonnet » et le chocolat « Droste ».

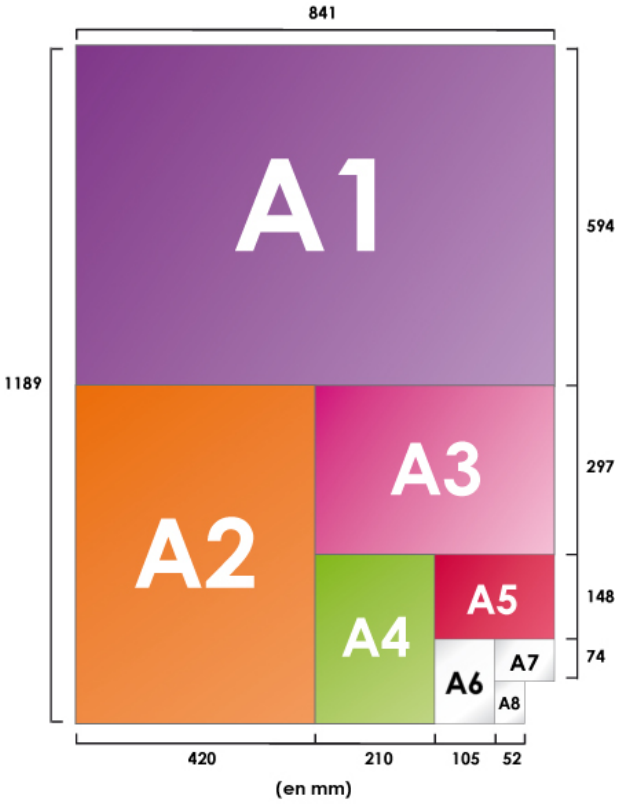
En anglais, on parle de « Droste effect » pour « mise en abyme ».



7.1.2. Vie courante : le format A4

La norme DIN-A détermine la taille du papier. Le format A0 qui est le plus grand format normalisé (1 mètre carré de surface) se décline jusqu'au format A10. La longueur du format inférieur est systématiquement égale à la largeur du format supérieur. Le format inférieur est donc obtenu en pliant le format supérieur en deux dans sa largeur.

Quel que soit le format, on trouve toujours le rapport $\sqrt{2}$ entre longueur et largeur.



A(i) est obtenu de A(i–1) en pliant dans sa longueur la feuille de papier. La relation de récurrence est donc :

- longueur de A(i) = largeur de A(i–1)
- largeur de A(i) = ½ longueur de A(i–1)

7.2. Fonctions récursives et itératives

En informatique et en mathématiques, une fonction qui s'appelle elle-même est dite récursive.

Deux fonctions peuvent s'appeler l'une l'autre, on parle alors de récursivité croisée, qui est très commune dans le style de programmation fonctionnelle et est souvent utilisée dans les langages LISP, Scheme, Prolog et autres langages similaires.

7.2.1. Élévation d'un entier *x* à une puissance *n*

La fonction puissance existe en Python. Mais si elle n'existait pas, voici une manière ingénieuse de l'implémenter.

```
def puissance(x,n):
    if n==0:
        return 1
    elif n==1:
        return x
    elif n%2==0:
        return puissance(x*x, n//2)
    else:
        return puissance(x*x, n//2)*x
```

Exercice 7.1

Ce programme calcule x^n. Expliquez comment il fonctionne.
Calculez « à la main » `puissance(2,9)`.

7.2.2. Calcul de la factorielle

Rappel : $n\,! = n\cdot(n-1)\cdot(n-2)\cdot \ldots \cdot 2\cdot 1$ pour n entier > 0. Cas particulier : 0 ! = 1.

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

Le cas n = 1 est appelé cas de base. Sans sa présence, l'algorithme ne peut pas se terminer.
On peut évidemment écrire une fonction itérative de la factorielle :

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

Le calcul de la factorielle est rarement utilisé tel quel en pratique. Pour des valeurs de n supérieures à 10, la formule de Stirling donne un résultat correct à 0.8% près :

$$n! \cong \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \text{ , avec e} = 2.718281828459...$$

Un appel de fonction est une opération plus coûteuse en soi que, par exemple, une opération

arithmétique ou un test. C'est pourquoi on préfère souvent la fonction itérative à la version récursive.

Dans le cas de la factorielle, on prendra plutôt la version itérative, mais il y a des cas où la fonction récursive est clairement préférable, par exemple pour parcourir des arbres (voir chapitre 8), ou faire des tris (voir chapitre 9).

Exercice 7.2

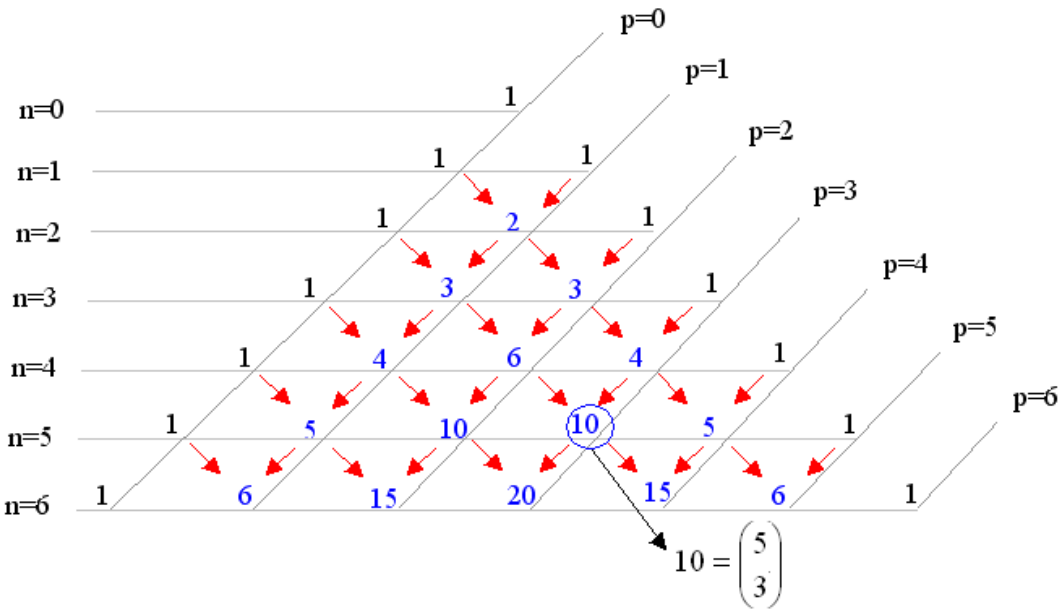
Écrivez précisément ce que ce programme va afficher à l'écran :

```
def fact(n):
    print("fact a été appelée avec n = " + str(n))
    if n == 1:
        return 1
    else:
        res = n * fact(n-1)
        print("Pour ", n, " * fact(" ,n-1, "): ",res)
        return res

print(fact(5))
```

7.2.3. Coefficients binomiaux

La fonction $C(n, p)$ donne le nombre de manières de prendre p éléments parmi n. Une importante relation que l'on peut observer dans le *triangle de Pascal* ci-dessous, lie les coefficients binomiaux :

$C(n+1, p+1) = C(n, p) + C(n, p+1)$ (formule de Pascal)




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

Exercice 7.3

La *suite de Syracuse* est définie par :

$$\begin{cases} x_1 & = & a \in \mathbb{N}^* \\ x_{n+1} & = & \begin{cases} \frac{x_n}{2} & \text{si } x_n \text{ est pair} \\ 3x_n + 1 & \text{si } x_n \text{ est impair} \end{cases} \end{cases}$$

Syracuse est ici le nom d'une ville universitaire américaine (New York).

1. Écrivez en Python une fonction itérative donnant la suite de Syracuse commençant par *a*.
2. Écrivez une version récursive.

La terminaison d'un algorithme récursif peut être un problème extrêmement difficile. Ainsi, personne n'a jusqu'à présent été capable de démontrer que la fonction *Syracuse* présentée plus haut se termine pour toute valeur de *n*.

Exercice 7.4

Écrivez un programme récursif permettant d'évaluer un nombre écrit en chiffres romains.

Rappelons que :

- M = 1000
- D = 500
- C = 100
- L = 50
- X = 10
- V = 5
- I = 1

Comme exemple, évaluons le nombre MCMXCIX.

```
|On est sur le premier M.
|Son successeur est C, il est plus petit, donc notre résultat final sera la
|valeur de M (1000) plus la valeur du reste (CMXCIX).
|  La valeur du reste est la suivante :
|  C est plus petit que M (une soustraction aura lieu) donc la valeur de
|  CMXCIX est égale à la valeur de MXCIX moins la valeur de C
|  |   La valeur de MXCIX est la suivante :
|  |   |M est plus grande que X donc on a 1000+valeur(XCIX).
|  |   |   La valeur de XCIX est égale à la valeur de CIX moins la valeur de X
|  |   |   |car le premier X est plus petit que son successeur.
|  |   |   |   La valeur de CIX est égale à 100 + valeur(IX) car C est plus
|  |   |   |   |grand que I.
|  |   |   |   |    La valeur de IX est égale à la valeur de X moins la valeur
|  |   |   |   |    de I, soit 10-1 = 9.
|  |   |   |   CIX = C + 9 = 109
|  |   |   XCIX = CIX - X = 109 - 10 = 99
|  |   MXCIX = M + XCIX = 1000 + 99 = 1099
|  CMXCIX = MXCIX - C = 1099 - 100 = 999
MCMXCIX = 1000 + 999 = 1999
```

On voit la forme de l'algorithme général :

- Soit un nombre romain. Si la première lettre de ce nombre a une valeur inférieure au deuxième, alors on le soustrait de la valeur de tout le reste. Sinon, on l'additionne à la valeur de tout le reste.
- Si le nombre romain a un seul chiffre, alors on prend simplement la correspondance (M = 1000, D = 500, ...).

On constate qu'avec l'algorithme ci-dessus, le nombre MIM (qui pourtant n'existe pas chez les romains) est évalué à 1999. Le programme n'effectuera pas un test d'exactitude de la chaîne passée en paramètre, il se contentera de l'évaluer.

7.3. Les dangers de la récursivité

Utiliser une fonction récursive n'est pas toujours une bonne idée.

Prenons la suite de Fibonacci : 1, 1, 2, 3, 5, 8, 13, 21, 34, ... Un terme est la somme des deux précédents. On va écrire deux fonctions (une récursive et une itérative) qui calculent le *k*-ième terme de cette suite, puis on comparera les temps de calcul.

```
import time

def fib1(n):
    # algorithme récursif
    if n==1 or n==2 :
        return 1
    else :
        return fib1(n-1) + fib1(n-2)

def fib2(n):
    # algorithme itératif
    i=1
    j=1
    k=3
    s=2
    if n==1 or n==2 :
        return 1
    else :
        while k<=n :
            s=i+j
            i=j
            j=s
            k+=1
    return s

for k in range(5):
    print((k+1)*10)
    a=time.clock()
    fib1((k+1)*10)
    b=time.clock()
    print("récursif :", b-a)
    a=time.clock()
    fib2(k)
    b=time.clock()
    print("itératif :", b-a)
```

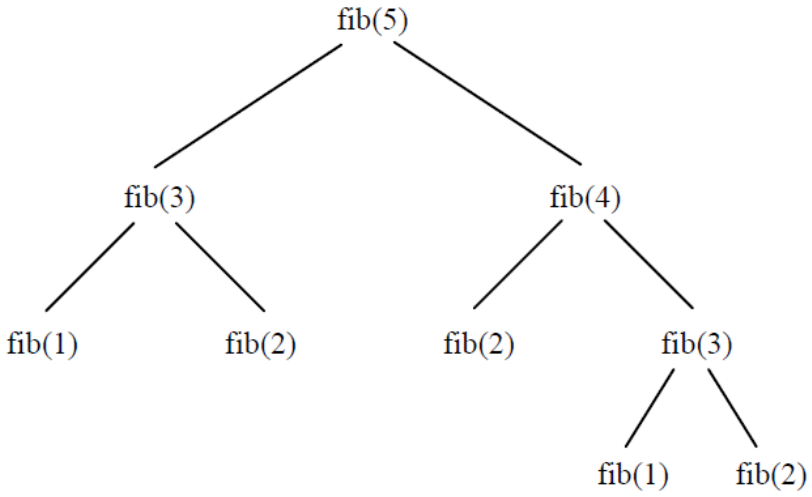
Voici une table des résultats (ces temps sont approximatifs et dépendent évidemment du processeur, mais ils mettent bien en évidence les problèmes de la récursivité) :

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

Pourquoi une telle différence ? Pour le comprendre, il faut observer comment se passe le calcul récursif. Calculons fib(5) avec la méthode récursive :

```
fib(5) -> fib(4) + fib(3)
       -> (fib(3) + fib(2)) + fib(3)
       -> (fib(2) + fib(1)) + fib(2)) + fib(3)
       -> ((1 + fib(1)) + fib(2)) + fib(3)
       -> ((1 + 1) + fib(2)) + fib(3)
       -> (2 + fib(2)) + fib(3)
       -> (2 + 1) + fib(3)
       -> 3 + fib(3)
       -> 3 + (fib(2) + fib(1))
       -> 3 + (1 + fib(1))
       -> 3 + (1 + 1)
       -> 3 + 2
       -> 5
```

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



On voit que ce n'est pas efficace : par exemple, `fib(3)` est appelé deux fois, ce qui est une perte de temps. De plus on imagine bien que l'arbre va devenir très vite gigantesque, avec un très grand nombre d'appels inutiles.

La mise en œuvre des algorithmes récursifs nécessite le plus souvent une pile. C'est la difficulté d'implanter cette pile ou d'éviter son emploi qui a fait dire pendant longtemps que les programmes récursifs étaient moins efficaces que les programmes itératifs, mais la situation a changé. En fait, le débat sur le choix entre codage récursif ou itératif est aussi vieux que l'informatique et les progrès de la compilation des langages de programmation réduit encore la différence d'efficacité. Voici quelques arguments en faveur de la présentation récursive :

- La présentation récursive permet de présenter simplement des algorithmes beaucoup plus astucieux (et donc plus efficaces) et cela a été admirablement montré par Tony Hoare avec son algorithme de tri rapide (Quicksort).
- Les compilateurs d'aujourd'hui sont tellement astucieux que plus le programme leur est présenté de façon abstraite et sans effets de bord, plus ils peuvent mettre en œuvre leurs optimisations et aboutir à des codes objets efficaces.
- Des structures de données récursives ont été conçues pour leur efficacité. On ne voit pas comment on pourrait exécuter sur elles des algorithmes non récursifs.

Un **compilateur** est un programme qui transforme un code source écrit dans un langage de programmation (le langage source) en un autre langage informatique (appelé langage cible).

7.4. Les tours de Hanoï

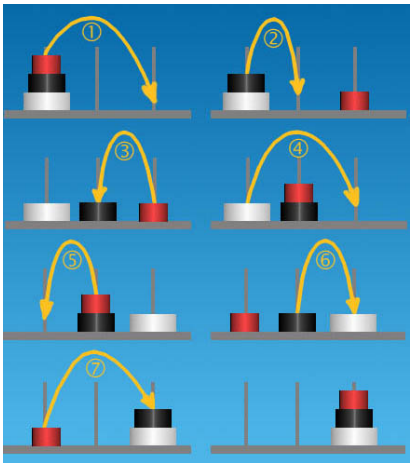
Le casse-tête des tours de Hanoï est un jeu de réflexion consistant à déplacer des disques de diamètres différents d'une tour de « départ » à une tour d' « arrivée » en passant par une tour « intermédiaire » et ceci en un minimum de coups, tout en respectant les règles suivantes :

- on ne peut pas déplacer plus d'un disque à la fois,
- on ne peut placer un disque que sur un autre disque plus grand que lui ou sur un emplacement vide.

On suppose que cette dernière règle est également respectée dans la configuration de départ.

Edouard **Lucas**
(1842 - 1891)

Le problème mathématique des tours de Hanoï a été inventé par Édouard **Lucas** (1842-1891). Il est publié dans le tome 3 de ses *Récréations mathématiques*, parues à titre posthume en 1892. Il annonce que ce problème est dû à un de ses amis, N. Claus de Siam, prétendument professeur au collège de Li-Sou-Stian (une double anagramme de Lucas d'Amiens, sa ville de naissance, et Saint Louis, le lycée où Lucas enseignait).
Sous le titre « Les brahmes tombent », **Lucas** relate que « *N. Claus de Siam a vu, dans ses voyages pour la publication des écrits de l'illustre Fer-Fer-Tam-Tam, dans le grand temple de Bénarès, au-dessous du dôme qui marque le centre du monde, trois aiguilles de diamant, plantées dans une dalle d'airain, hautes d'une coudée et grosses comme le corps d'une abeille. Sur une de ces aiguilles, Dieu enfila au commencement des siècles, 64 disques d'or pur, le plus large reposant sur l'airain, et les autres, de plus en plus étroits, superposés jusqu'au sommet. C'est la tour sacrée du Brahmâ. Nuit et jour, les prêtres se succèdent sur les marches de l'autel, occupés à transporter la tour de la première aiguille sur la troisième, sans s'écarter des règles fixes que nous venons d'indiquer, et qui ont été imposées par Brahma. Quand tout sera fini, la tour et les brahmes tomberont, et ce sera la fin des mondes !* ».



Solution pour 3 disques

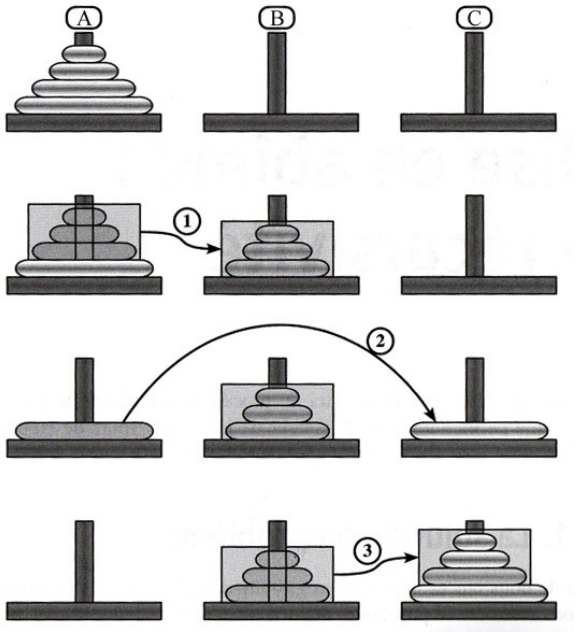
Un jeu à 64 disques requiert un minimum de $2^{64}-1$ déplacements. En admettant qu'il faille 1 seconde pour déplacer un disque, ce qui fait 86'400 déplacements par jour, la fin du jeu aurait lieu au bout d'environ 213'000 milliards de jours, ce qui équivaut à peu près à 584,5 milliards d'années, soit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

7.4.1. Résolution récursive

Texte et images de ce paragraphe tirés de [4], pp. 112-114

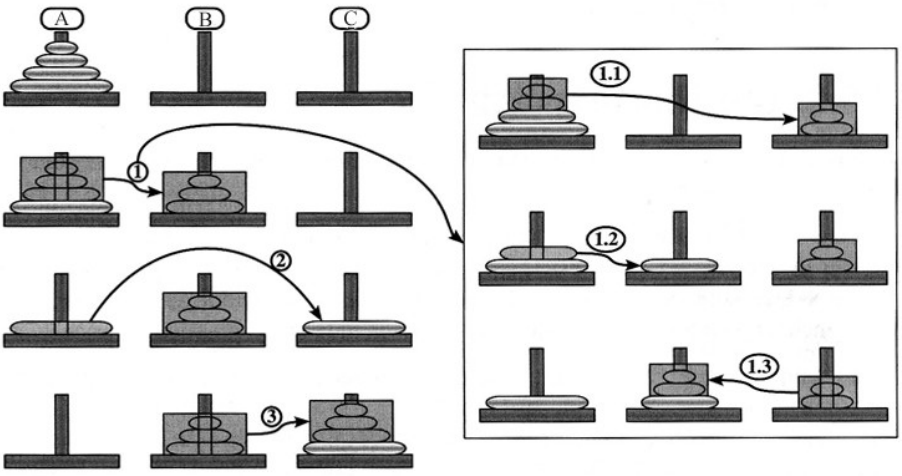
Pour résoudre le problème des tours de Hanoï, il faut raisonner récursivement. Pour fixer les idées, prenons par exemple quatre disques. Parmi eux, un est remarquable : le plus grand. Il ne peut être posé sur aucun autre, il est donc le plus contraignant à déplacer. Une solution idéale ne devrait déplacer ce disque qu'une seule fois. Nommons *A*, *B* et *C* les trois piquets, la pile de disques au

départ sur *A*. Idéalement, donc, le plus grand disque sera déplacé de *A* à *C* en une seule fois. Comme il ne peut être posé sur aucun autre disque – car il est le plus grand –, cela ne pourra se faire que si aucun disque n'est présent sur *C*. Autrement dit, on ne pourra effectuer ce déplacement particulier que si le grand disque est seul sur *A*, et que tous les autres disques sont sur *B*. Incidemment, nous venons de réduire le problème : pour pouvoir déplacer nos quatre disques de *A* vers *C*, il faut d'abord déplacer (seulement) trois disques de *A* vers *B*, puis déplacer le grand sur *C*, et enfin déplacer les trois disques de *B* vers *C*.



Le grand disque présente une autre particularité : il est le seul sur lequel on peut poser n'importe lequel des autres disques. Donc, dans le cadre d'un déplacement des trois disques supérieurs, il n'a aucune incidence : tout se passe comme s'il n'était pas là. Cette remarque nous amène à un constat intéressant : on peut traiter le problème du déplacement des trois disques exactement de la même manière que nous avons traité celui des quatre.

La figure ci-dessous, dans sa partie droite, montre comment la première des trois étapes peut elle-même être décomposée en trois « sous-étapes » : déplacer les deux disques supérieurs, puis le disque inférieur, puis déplacer les deux disques supérieurs. La troisième étape de la partie gauche pourrait être décomposée de la même façon.



En résumé, déplacer n disques de A vers C en passant par B consiste à :

1. déplacer (n–1) disques de A vers B (en passant par C);
2. déplacer le plus grand disque de A vers C ;
3. déplacer (n–1) disques de B vers C (en passant par A).

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

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

Et ainsi de suite...
Ce qui donne en Python :

```
def hanoi(n,de,a,par):
    if n>0:
        hanoi(n-1, de, par, a)
        print(de,"-->",a)
        hanoi(n-1, par, a, de)

print("""
Tours de Hanoi
Il faut deplacer les disques de la tour A vers la tour C
""")
n = int(input("Nombre de disques : "))
hanoi(n,"A","C","B")
```

7.4.2. Résolution itérative

Il existe également une procédure itérative pour résoudre le problème des tours de Hanoï. Elle consiste à effectuer successivement les deux déplacements suivants :

- déplacer le plus petit disque d'un emplacement à l'emplacement suivant (de A vers B, de B vers C, ou de C vers A)
- déplacer un autre disque

et à poursuivre itérativement ces deux déplacements jusqu'à ce que la tour complète soit déplacée, le dernier déplacement se limitant alors à celui du plus petit disque sur le sommet de la tour. L'action de déplacer un autre disque est non ambiguë puisque, en dehors du plus petit disque, un seul mouvement d'un autre disque est possible.

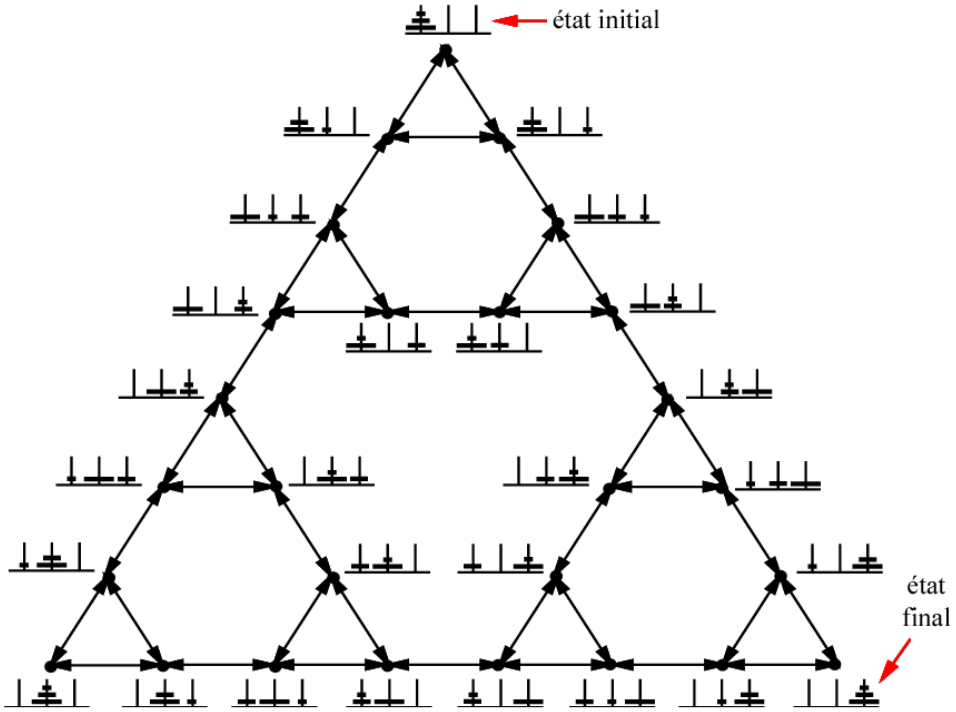
Contrairement à la procédure récursive, la procédure itérative n'utilise aucune mémorisation de l'arbre des déplacements à effectuer et nécessite seulement de se souvenir si on doit déplacer le plus petit disque ou non, et dans quel sens sont effectués les déplacements du petit disque. Il permet également, à tout moment, de revenir à la situation de départ : il suffit pour cela d'inverser le sens dans lequel se déplace le plus petit disque.

Exercice 7.5

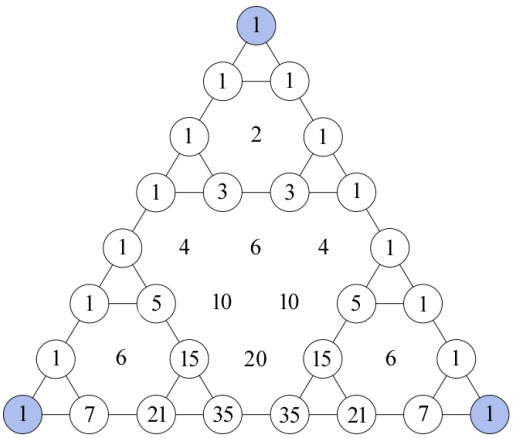
Écrivez un programme en Python résolvant les tours de Hanoï sans utiliser la récursivité. Vous trouverez une ébauche sur www.apprendre-en-ligne.net/info/recursivite

7.4.3. Hanoï, Pascal et Sierpinski

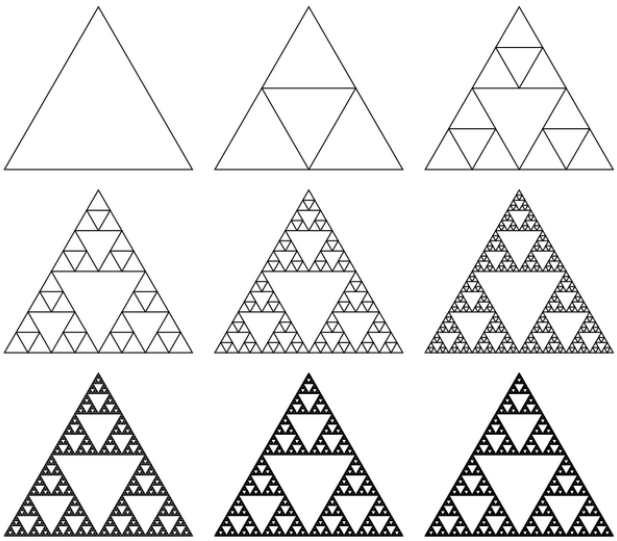
Si l'on représente sur un graphique tous les mouvements possibles des tours de Hanoï, on obtient une figure fractale :



Ce triangle ressemble étrangement à deux autres triangles célèbres : le triangle de Pascal (déjà rencontré au § 7.2.3) et celui de Sierpinski :



Triangle de Pascal



Construction du triangle de Sierpinski

7.5. Le compte est bon

D'après le jeu « Des chiffres et des lettres » d'Armand Jammot (France 2).

Le principe du jeu

En choisissant 6 nombres (on peut choisir plusieurs fois le même) dans l'ensemble {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 25, 50, 75, 100} et en leur appliquant les quatre opérations élémentaires (addition, soustraction, multiplication et division), il s'agit d'atteindre le résultat demandé (ceci est possible dans 94% des cas). Tous les résultats des calculs intermédiaires doivent être des nombres entiers positifs. Chacun des nombres (parmi ceux de départ et ceux obtenus par un calcul intermédiaire) ne peut être utilisé qu'une fois au plus. Si le résultat demandé ne peut pas être atteint, il faut l'approcher au plus près.

Exemple : atteindre 952 avec les nombres 25, 50, 75, 100, 3, 6.

100 + 3 = 103
103 x 6 = 618
618 x 75 = 46350
46350 / 50 = 927
927 + 25 = 952

Algorithme de résolution

1. on sélectionne une paire de nombres
2. on fait une opération sur cette paire
3. si on a trouvé le nombre voulu, STOP
4. on sauvegarde le tableau courant dans un tableau auxiliaire
5. on remplace les deux nombres utilisés par le résultat de l'opération. On obtient donc un tableau auxiliaire plus petit d'un élément
6. on trie le tableau auxiliaire par ordre décroissant
7. on fait un appel récursif pour recommencer le raisonnement sur ce tableau
8. si on n'a pas fini les quatre opérations, aller à 2
9. si on n'a pas fini toutes les paires, aller à 1

Ce qui donne en Python :

```
# "le compte est bon" récursif

def operations(t, max):
    global trouve, t1, signe, objectif
    for i in range(4):
        for j1 in range(max-1):
            for j2 in range(j1+1, max):
                if i==0:                # addition
                    a = t[j1] + t[j2]
                elif i==1:              # soustraction
                    a = t[j1] - t[j2]
                elif i==2:              # multiplication
                    a = t[j1] * t[j2]
                else:                   # division (si possible)
                    if t[j1] % t[j2] == 0:
                        a = t[j1] // t[j2]
                    else:
                        a = 0
                if a > 0 :
                    if a == objectif :
                        print(t[j1],signe[i],t[j2],'=',a)
                        trouve = True
                        break
                    t1 = t[:]
```

```
                    t1[j1] = a
                    t1[j2] = 0
                    t1.sort()
                    t1.reverse()
                    operations(t1, max-1)
                    if trouve :
                        print(t[j1],signe[i],t[j2],'=',a)
                        break
            if trouve :
                break
        if trouve :
            break

signe = '+-*/'
t1 = [0]*6
trouve = False
objectif = 951
nombres = [100, 75, 50, 25, 6, 3]
print("Objectif",objectif)
print("Tirage",nombres)
print("Lire la solution (si elle existe) de bas en haut")
print()
operations(nombres, 6)
```

7.6. Résolution d'un Sudoku

Le programme complet se trouve sur le site web compagnon.

Voici la partie « Résolution » d'un programme Python qui trouve la solution d'un Sudoku par *backtracking*. Le backtracking (*retour sur trace* en français, mais le terme est peu employé) est une méthode qui consiste à revenir légèrement en arrière sur des décisions prises afin de sortir d'un blocage. Le terme est surtout utilisé en programmation, où il désigne une stratégie pour trouver des solutions à des problèmes de satisfaction de contraintes.

La grille de 9 cases sur 9 sera implémentée par une liste de 81 éléments. Chaque élément de la liste sera un chiffre de 0 à 9. 0 indique une case vide.

Les cases sont numérotées de gauche à droite et de haut en bas :

0	1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16	17
18	19	20	21	22	23	24	25	26
27	28	29	30	31	32	33	34	35
36	37	38	39	40	41	42	43	44
45	46	47	48	49	50	51	52	53
54	55	56	57	58	59	60	61	62
63	64	65	66	67	68	69	70	71
72	73	74	75	76	77	78	79	80

La fonction `conflit(i,j)` permet de savoir si les cases i et j sont

- dans la même colonne ou
- dans la même ligne ou
- dans le même bloc .

```
# Solveur de Sudoku par backtracking

# ----- Partie résolution ---------

bloc = [0]*9
bloc[0] = {0,1,2,9,10,11,18,19,20}
```

```
bloc[1] = {3,4,5,12,13,14,21,22,23}
bloc[2] = {6,7,8,15,16,17,24,25,26}
bloc[3] = {27,28,29,36,37,38,45,46,47}
bloc[4] = {30,31,32,39,40,41,48,49,50}
bloc[5] = {33,34,35,42,43,44,51,52,53}
bloc[6] = {54,55,56,63,64,65,72,73,74}
bloc[7] = {57,58,59,66,67,68,75,76,77}
bloc[8] = {60,61,62,69,70,71,78,79,80}

def conflit_bloc(i,j):
    k=0
    while not i in bloc[k]:
        k+=1
    return j in bloc[k]

def conflit_colonne(i,j):
    return (i-j)%9 == 0

def conflit_ligne(i,j):
    return i//9 == j//9

def conflit(i,j):
    return (conflit_colonne(i,j) or conflit_ligne(i,j) or conflit_bloc(i,j))
```

C'est la fonction récursive `résoudre(s)`, qui est le cœur du programme.

Le paramètre `s` est la grille du Sudoku, donnée sous la forme d'une liste. Au départ, cette liste contient la donnée du problème, mais ensuite, elle se remplira et se désemplira au gré des essais.

```
def resoudre(s):
    try:
        i  = s.index(0)  # prochaine cellule vide
    except ValueError:
        # plus de case vide : solution trouvée !
        return s
    # mettre dans interdit toutes les valeurs des 20 cases voisines
    # de la case courante
    # ces valeurs sont des valeurs interdites pour la case courante
    interdit = [s[j] for j in range(81) if conflit(i,j)]
    # on essaie toutes les valeurs de 1 à 9
    for v in range(1, 10):
        if v not in interdit:
            # pas de conflit. On essaie de résoudre le nouveau sudoku
            r = resoudre(s[:i]+[v]+s[i+1:])
            if r is not None:
                return r
```

On essaie de remplir la grille selon la numérotation des cases. Pour chaque case vide, on essaiera les chiffres possibles en allant de 1 à 9. Si un chiffre est possible, on l'écrit (peut-être provisoirement) et on passe à la case vide suivante. On avance ainsi dans le remplissage de la grille.

Il arrivera forcément un moment où l'on sera bloqué : aucun chiffre de 1 à 9 ne pourra être placé. On revient alors une case en arrière et on essaie un autre chiffre. Si on trouve une nouvelle possibilité, on remplace l'ancienne et on pourra continuer d'avancer. Sinon, on efface la valeur écrite et on revient encore une case en arrière pour un autre essai. Et ainsi de suite. Il est tout à fait possible que l'on doive ainsi remonter jusqu'à la première case.

Le temps de résolution est très variable : il va de quelques secondes à plusieurs minutes.

Vous pourrez télécharger ce programme sur www.apprendre-en-ligne.net/info/recursivite. Vous trouverez aussi un fichier contenant une liste de grilles à résoudre.

On le voit, cette méthode n'utilise aucun raisonnement. C'est une recherche exhaustive ou recherche par force brute : une technique extrêmement simple et très générale.

7.7. Les fractales

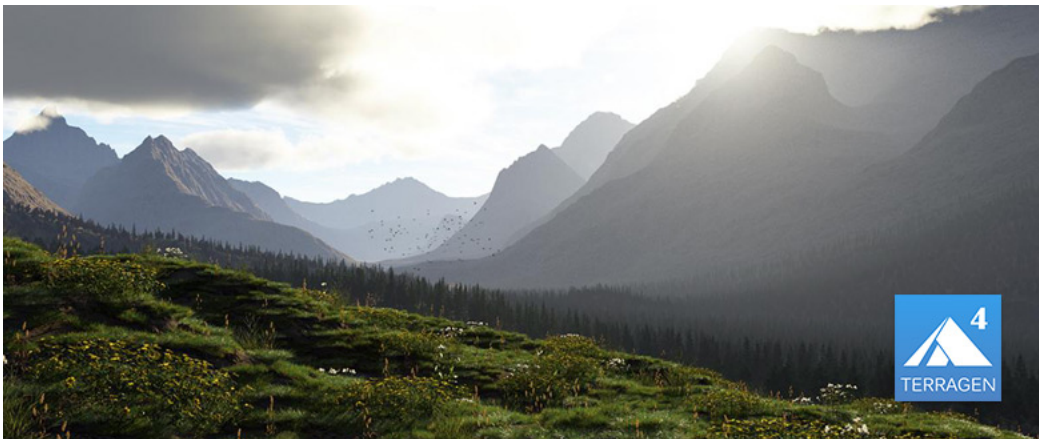
Benoît **Mandelbrot**
(1924 - 2010)

Une figure fractale ou fractale est une courbe ou surface de forme irrégulière ou morcelée qui se crée en suivant des règles déterministes ou stochastiques impliquant une homothétie interne. Le terme « fractale » est un néologisme créé par Benoît **Mandelbrot** en 1974 à partir de la racine latine *fractus*, qui signifie brisé, irrégulier .

Un des plus beaux exemples de fractale donné par la nature est le chou Romanesco (à gauche) :

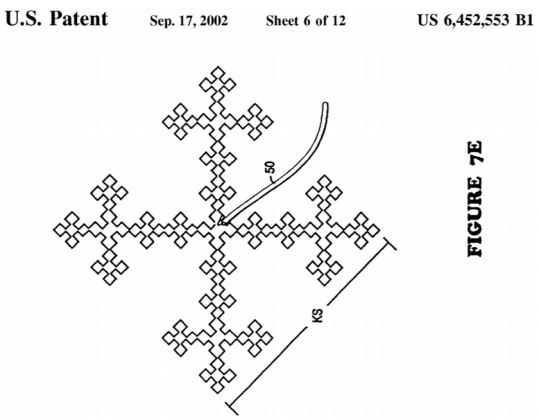
Si l'on ne regarde qu'une des pointes, on a l'impression de voir un chou en entier. C'est le principe d'auto-similarité. On retrouve de l'auto-similarité dans les fougères (à droite) : chaque feuille ressemble à la fougère entière.

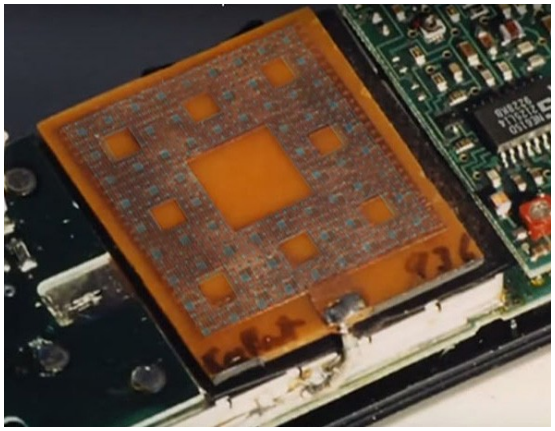
Les fractales sont très utilisées pour générer des paysages virtuels. L'image ci-dessous a été réalisée avec le logiciel gratuit **Terragen**.



Les fractales ne sont pas seulement belles, elle sont aussi très utiles en technologie, par exemple dans la conception d'antennes. Le but des antennes fractales est de maximiser la longueur efficace ou encore d'augmenter le périmètre de matériau (sur les sections internes ou la structure externe) pouvant recevoir ou transmettre un rayonnement électromagnétique dans une surface ou un volume total donné.

On qualifie aussi de telles antennes fractales de courbes remplissantes ou courbes à plusieurs niveaux. Leur aspect clé réside dans leur répétition d'un motif sur plusieurs tailles d'échelle. Pour cette raison, les antennes fractales sont très compactes, multibandes ou à large bande, et ont des applications en télécommunications et dans les communications micro-ondes.





Deux antennes fractales en forme d'*Île de Minkowski* (à gauche) et de *Tapis de Sierpinski*

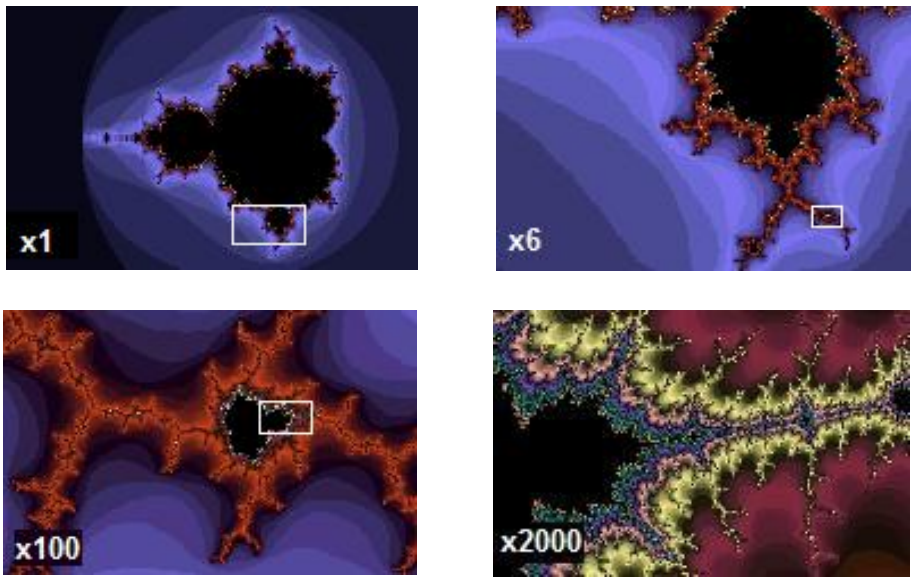
En algèbre, un des objets mathématiques les plus fascinants et les plus complexe est *l'ensemble de Mandelbrot*, qui est devenu l'icône des fractales. Il est défini comme l'ensemble des points du plan complexe pour lesquels la suite définie par la récurrence

$$\begin{cases} z_0 & = & 0 \\ z_{n+1} & = & z_n^2 + c \end{cases}$$

ne tend pas vers l'infini (en module).

Comme on le voit, la formule est simple et le programme aussi (voir site web compagnon). Pourtant le résultat est d'une incroyable complexité. On trouve sur le web quantité d'images et de vidéos qui vous en convaincront.

La séquence ci-dessous montre différents agrandissements qui mettent en évidence l'auto-similarité.

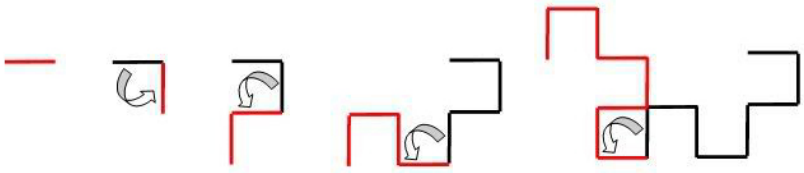


7.7.1. La courbe du dragon

La **courbe du dragon** (ou « Fractale du dragon » ou « courbe de Heighway » ou « dragon de Heighway ») a été pour la première fois étudiée par les physiciens de la NASA John **Heighway**, Bruce **Banks**, et William Harter. Elle a été décrite par Martin **Gardner** dans sa chronique de jeux

mathématiques du Scientific American en 1967. Nombre de ses propriétés ont été publiées par Chandler **Davis** et Donald **Knuth**.

La courbe du dragon se construit ainsi :



- Si $t = 0$, 'ordinateur doit dessiner une ligne. C'est la base (ou l'initiateur). La longueur a peu d'importance. On définit la longueur une fois avec s. La commande tortue est : av(s).
- Sinon, si $t > 0$:
  Dragon(t) = Dragon (t–1) ┐ Dragon (t –1). C'est la règle de récursivité (ou le générateur).

L'ordinateur doit dessiner une courbe de dragon avec profondeur de récursion t–1. Cela donne :

Dessiner Dragon (t–1)
Tourner à gauche (90°)
Dessiner Dragon (t–1)

Il y a un petit problème : on ne peut pas dessiner Dragon(t–1) exactement de la même façon les deux fois. En effet, le premier Dragon (t–1) est dessiné vers l'extérieur en partant du milieu de Dragon(t). Ensuite on tourne de 90°. Le deuxième Dragon(t–1) est dessiné du milieu de Dragon(t) vers l'extérieur. Pour que les deux Dragon(t–1) soient représentés de la même façon, le deuxième Dragon(t–1) doit être dessiné en miroir. Cela veut dire que tous les angles

(a) sont *en miroir* et
(b) doivent être dessinés *dans l'ordre inverse*.

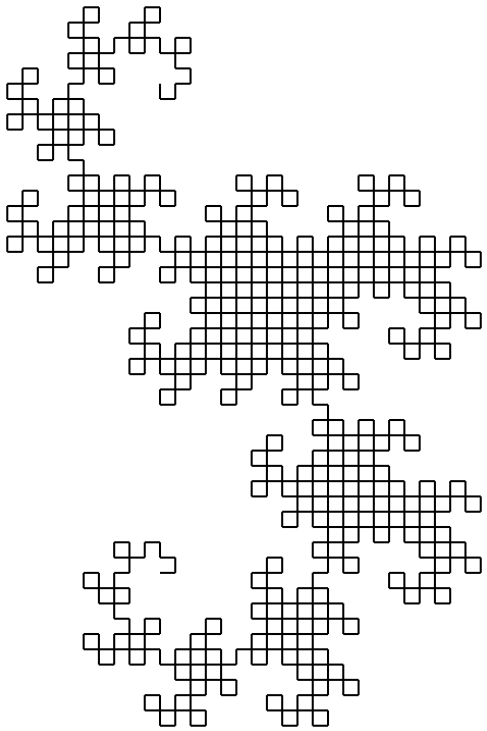
L'astuce consiste à donner un signe qui indique le sens (vz = 1 veut dire « + », vz = –1 veut dire « – »). On dessine d'abord un Dragon (t–1) avec signe positif (vz = 1). Ensuite on tourne de 90° et dessinons un Dragon (t–1) avec signe négatif (vz = –1).

Dessiner Dragon (t –1) signe (+)
Tourner à gauche (vz·90°)
Dessiner Dragon (t–1) signe (–)

```
# Courbe du dragon avec instructions Logo

from tkinter import *
from math import sin, cos, radians

# ------------------------ commandes logo --------------------------

def fpos(x0,y0):
    # place la tortue en (x0; y0)
    global x,y
    x = x0
    y = y0

def fcap(angle0):
    global angle
    # oriente la tortue dans une direction (en degrés)
    angle = angle0

def av(d):
    # avance en dessinant
    global x, y
    x2 = x + d*cos(angle)
```

```
    y2 = y + d*sin(angle)
    can.create_line(x, y, x2, y2, width=2, fill="black")
    x = x2
    y = y2

def tg(a):
    # tourne à gauche de a degrés
    global angle
    angle -= radians(a)

# --------------------------------------------------------------------

def dragon(t,vz):
    if t==0:
        av(15)
    else:
        dragon(t-1,1)
        tg(vz*90)
        dragon(t-1,-1)

def dessiner():
    fpos(300,600)
    fcap(0)
    dragon(10,1)

# --------------------------------------------------------------------

fen = Tk()
can = Canvas(fen, bg='white', height=800, width=800)
can.pack(side=TOP)
bou1 = Button(fen, text='Quitter', width=10, command=fen.quit)
bou1.pack(side=RIGHT)
dessiner()
fen.mainloop()
fen.destroy()
```

Et voici la courbe que l'on obtient avec un niveau de récursivité de 10 :



7.7.2. Le flocon de von Koch

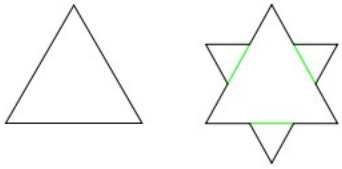
Helge **von Koch**
(1870 - 1924)

Le flocon de von Koch est l'une des premières courbes fractales à avoir été décrite (bien avant l'invention du terme « fractal(e) »). Elle a été inventée en 1906 par le mathématicien suédois Helge **von Koch**.

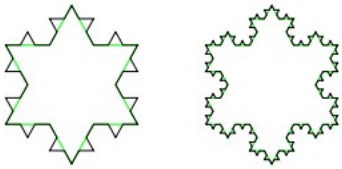
La courbe de von Koch d'un segment de droite, en modifiant récursivement chaque segment de droite de la façon suivante :

1. On divise le segment de droite en trois segments de longueurs égales.
2. On construit un triangle équilatéral ayant pour base le segment médian de la première étape.
3. On supprime le segment de droite qui était la base du triangle de la deuxième étape.

Au bout de ces trois étapes, l'objet résultant a une forme similaire à une section transversale d'un chapeau de sorcière. La courbe de von Koch est la limite des courbes obtenues, lorsqu'on répète indéfiniment les étapes mentionnées ci-avant.



Le **flocon de von Koch** s'obtient de la même façon que la fractale précédente, en partant d'un triangle équilatéral au lieu d'un segment de droite, et en effectuant les modifications en orientant les triangles vers l'extérieur.



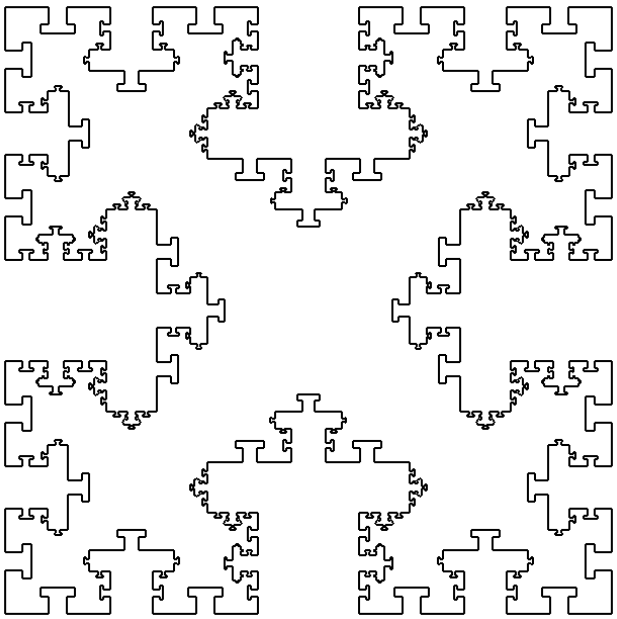
Exercice 7.6

En utilisant les commandes Logo vues avec la courbe du dragon (vous pouvez ajouter la procédure `td(a)` qui permettra de tourner la tortue vers la droite), dessinez le flocon de von Koch avec un niveau de récursivité de 4.

Vous trouverez une ébauche sur www.apprendre-en-ligne.net/info/recursivite

Exercice 7.7

Voici une fractale obtenue en modifiant l'exercice 7.6. Saurez-vous la reproduire ?



Sources

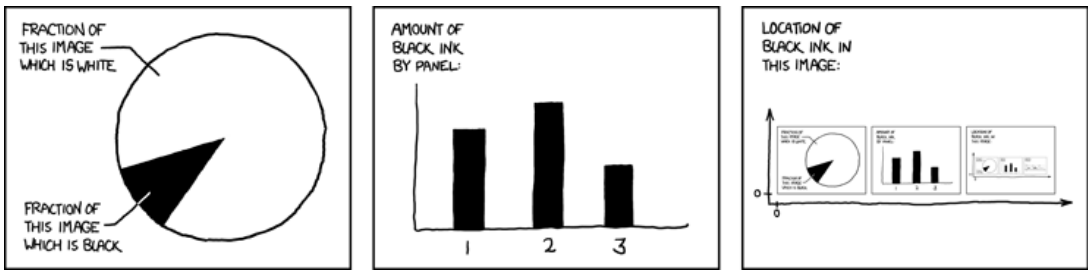
[1] *ETH-Programme dirigé de Programmation récursive*, A. Alder et al., ETHZ, 1995 (traduction de Brice Canvel)

[2] *Cours modulaire de programmation*, G. Coray, EPFL, 1986

[3] « Cours sur la récursivité », Axel Chambily-Casadesus et Pétrut Constantine, <http://recursivite.developpez.com>, 2009

[4] *Initiation à la programmation avec Python et C++*, Y. Bailly, Pearson, 2011 (2ème édition)

[5] « Ensemble de Mandelbrot », Wikipédia, <http://fr.wikipedia.org/wiki/Ensemble_de_Mandelbrot>



xkcd : self-description