

# XVI – La récursivité

## Introduction

D'après la méthode de Héron<sup>1</sup>, le réel  $\sqrt{3}$  est la limite de la suite  $(u_n)_{n \in \mathbb{N}}$  définie par

$$\begin{cases} u_0 = 2 \\ u_n = \frac{1}{2} \left( u_{n-1} + \frac{3}{u_{n-1}} \right) \text{ pour tout } n \in \mathbb{N}^* \end{cases}$$

Le calcul des termes successifs d'une telle suite est aisé : une simple boucle `for` permet d'écrire une **fonction itérative** qui prend en argument l'indice  $n$  et renvoie le terme  $u_n$ .

**Exemple** : voici la fonction en question ainsi qu'une comparaison des résultats à  $\sqrt{3}$ .

|                                                                                                        |                                                                                                      |
|--------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------|
| <pre>def u(n):<br/>    u=2<br/>    for i in range(n):<br/>        u=0.5*(u+3/u)<br/>    return u</pre> | <pre>== TEST DE LA CONVERGENCE ==<br/><br/>for n in range(10):<br/>    print(abs(u(n)-3**0.5))</pre> |
|--------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------|

Néanmoins, il est tentant de copier la définition mathématique de la suite  $(u_n)_{n \in \mathbb{N}}$  en écrivant la fonction `u` de la manière suivante :

```
def u(n):  
    if n==0:  
        return 2  
    else:  
        return 0.5*(u(n-1)+3/u(n-1))
```

Cela fonctionne bien. En fait, une fonction peut s'appeler elle-même lors de sa définition : on parle alors de **fonction récursive**.

**Exemple** : tester la fonction ci-dessus, et comparer les vitesses des versions itérative et récursive pour  $n = 20$ .

Les fonctions récursives sont beaucoup plus lentes que les itératives, mais plus simples écrire. Il faudra donc les utiliser à bon escient et essayer d'optimiser les calculs.

**Remarque** : le code précédent contient **une monstruosité** qui rend la fonction récursive franchement inefficace. On verra plus loin comment remédier à ce problème.

---

1. C'est un cas particulier de la méthode de Newton, appliquée à  $f : x \mapsto x^2 - a$  pour approcher  $\sqrt{a}$

# 1/ Principes de la récursivité

## 1.1) Définitions

**Définition** Une fonction *récursive* est une fonction qui s'appelle elle-même.

**Exemple** : voici une fonction qui calcule la factorielle de  $n$  de manière récursive. On peut l'utiliser pour obtenir  $6!$  et  $80!$  par exemple. Mais ... que se passe-t-il précisément lorsque l'on invoque `facto(3)` ?

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

**Remarques** :

- ❶ Lorsque l'on définit une fonction récursive, il y a toujours un ou plusieurs cas de base qui sont traités de manière non récursive. Ce sont les **cas terminaux**.
- ❷ Ceci fait penser au principe du raisonnement par récurrence. En effet, la récursivité est particulièrement adaptée au calcul des termes d'une suite définie par récurrence.

**Définition** La *pile d'appels* est une structure de données qui sert à enregistrer les informations concernant les fonctions actives (c'est-à-dire en cours d'exécution).

Lors de l'exécution d'une fonction, la pile d'appels emmagasine son adresse de retour (c'est celle de l'instruction suivante), les variables locales de la fonction, etc ... On risque alors un **débordement de pile** (en anglais *stack overflow*) quand les appels sont imbriqués, comme avec une fonction récursive. Pour éviter ceci, le nombre d'appels imbriqués est limité.

**Exemple** : lancer `facto(1000)`.

On peut l'augmenter en utilisant la fonction `setrecursionlimit` de la bibliothèque `sys`. Mais attention aux débordements de pile, voire aux plantages de la machine !

**Exemple** : après un `setrecursionlimit(150000)`, tenter `facto(1000)` et `facto(100000)`.

## 1.2) Deux exemples fondamentaux

### L'algorithme d'Euclide

Cet algorithme permet de trouver facilement le plus grand diviseur commun (noté pgcd) de deux entiers naturels  $a$  et  $b$ . Il est basé sur le lemme d'Euclide :

**Lemme**

*Si  $b \neq 0$ , alors  $\text{pgcd}(a, b) = \text{pgcd}(b, r)$ , où  $r$  est le reste de la division euclidienne de  $a$  par  $b$ .*

On enchaîne ainsi les divisions euclidiennes, en prenant à chaque étape le diviseur et le reste obtenu comme dividende et diviseur de la nouvelle division. Enfin  $\text{pgcd}(a, 0) = a$  si bien que le dernier reste non nul est le pgcd recherché.

**Exemple** : calculer le pgcd de 17 et 5, puis celui de 222 et 30.

L'algorithme d'Euclide conduit tout naturellement à une fonction itérative comportant une boucle `while`. La version récursive est la simple traduction du lemme d'Euclide.

|                                                                                |                                                                                    |
|--------------------------------------------------------------------------------|------------------------------------------------------------------------------------|
| <pre>def pgcd_I(a,b):     while b&gt;0:         a,b = b,a%b     return a</pre> | <pre>def pgcd_R(a,b):     if b==0:         return a     return pgcd_R(b,a%b)</pre> |
|--------------------------------------------------------------------------------|------------------------------------------------------------------------------------|

### La méthode d'exponentiation rapide

Cette méthode permet de minimiser le nombre de multiplications effectuées lors du calcul de  $x^n$  avec  $x \in \mathbb{R}$  et  $n \in \mathbb{N}$ , en observant que

$$\forall k \in \mathbb{N} \quad \begin{cases} x^{2k} = (x^2)^k \\ x^{2k+1} = x \times (x^2)^k \end{cases}$$

On ramène ainsi le calcul de  $x^n$  à celui de  $(x^2)^k$ , où  $k = \lfloor n/2 \rfloor$ . Nous avons déjà croisé la version itérative de cet algorithme. La version récursive est beaucoup plus simple à écrire.

|                                                                                                                                                                               |                                                                                                                                  |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------|
| <pre>def expo_I(x,n):     p=1     while n&gt;0:         if n%2==0:             x=x**2             n=n//2         else:             p=p*x             n=n-1     return p</pre> | <pre>def expo_R(x,n):     if n==0:         return 1     y=expo_R(x**2,n//2)     if n%2==0:         return y     return x*y</pre> |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------|

### 1.3) Avantages et inconvénients

La récursivité permet des formulations claires et proches des définitions mathématiques. Certains problèmes qui s'écrivent très simplement sous forme récursive sont franchement indigestes sous forme itérative, comme on le verra dans la section 4.

Mais la récursivité possède un inconvénient majeur : la simplicité d'écriture masque le nombre d'opérations sous-jacentes et peut conduire à des algorithmes extrêmement lents, alors que les versions itératives sont optimisées en terme de rapidité d'exécution. Et même à complexité équivalente, l'engorgement de la pile d'appels ralentit l'exécution.

### Le problème

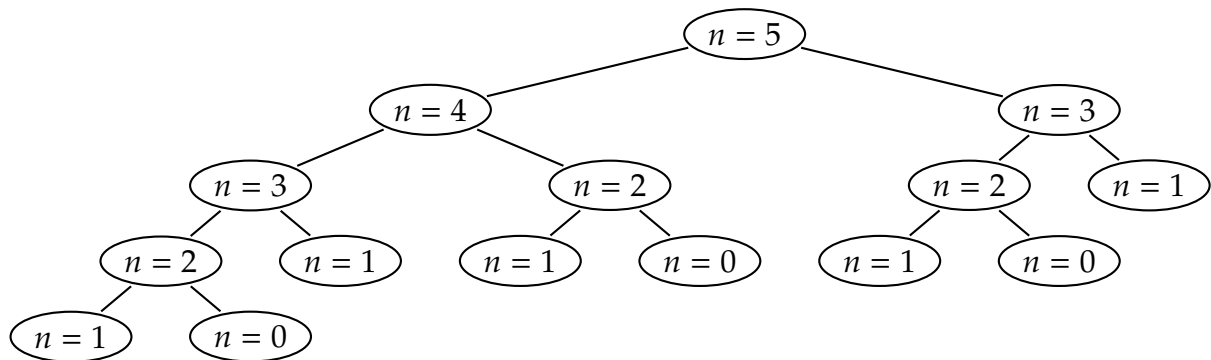
La programmation naïve d'une fonction récursive  $f$  peut parfois conduire à une explosion combinatoire des appels récursifs, en particulier lorsque le calcul de  $f(n)$  provoque plusieurs appels à  $f(n-1)$  ou à d'autres valeurs calculées antérieurement.

On calculera précisément les complexités dans la prochaine section.

**Exemple :** prenons la version récursive naïve du calcul de la suite de Fibonacci

```
def Fibo(n):
    if n==0:
        return 0
    elif n==1:
        return 1
    else:
        return Fibo(n-2)+Fibo(n-1)
```

Cette fonction est appelée à plusieurs reprises avec les mêmes arguments, comme on peut le voir en représentant à l'aide d'un arbre les différents appels provoqués par `Fibo(5)` :



La présence d'appels redondants rend ainsi la fonction complètement inefficace.

### Premier remède

Réécrire la fonction récursive en utilisant une fonction auxiliaire qui ne contient qu'un seul appel récursif.

L'idée est la suivante : en partant de  $(u_0, u_1)$ , on calcule  $(u_1, u_2)$ , puis  $(u_2, u_3)$ , ..., et ainsi de suite jusqu'à obtenir  $(u_{n-1}, u_n)$ . Il suffit donc d'écrire une fonction récursive (contenant un seul appel) qui « transporte » les couples  $(u_{p-1}, u_p)$  de  $p = 1$  jusqu'à  $n$ .

C'est la technique de **fenêtre glissante**, une méthode efficace et assez générale, bien que parfois délicate à mettre en œuvre. Voici comment procéder ici.

```
def Fibo(n):
    if n==0:
        return 0
    elif n==1:
        return 1
    else:
        return G(0,1,n,1)
```

```
def G(a,b,n,p):
    """ Si a=u_{p-1} et b=u_p,
        on renvoie u_n """
    if n==p:
        return b
    else:
        return G(b,a+b,n,p+1)
```

**Deuxième remède**

Écrire la relation de récurrence sous forme matricielle et utiliser l'exponentiation rapide pour calculer les puissances de matrices.

Cette méthode, bien que spécifique au cas présent, mérite néanmoins d'être examinée : en notant  $X_n = \begin{pmatrix} u_n \\ u_{n+1} \end{pmatrix}$ , la relation de récurrence devient alors  $X_{n+1} = AX_n$  avec  $A = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$ . On en déduit aisément que  $X_{n-1} = A^{n-1}X_0$  avec  $X_0 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$ . Ainsi,  $u_n$  est le coefficient en bas à droite de la matrice  $A^{n-1}$ , dont le calcul se fait récursivement par exponentiation rapide (en exercice).

**Troisième remède**

Utiliser un dictionnaire pour stocker les valeurs déjà calculées et éviter les appels redondants.

C'est la méthode de **mémoïsation** : universelle et simple à mettre en œuvre, elle vient à bout des problèmes d'explosion combinatoire en évitant les multiplications d'appels.

L'idée essentielle est la suivante : si l'on appelle une fonction qui prend en argument un dictionnaire déjà existant, toute modification de celui-ci dans la fonction le modifie aussi dans le programme global. Ceci permet d'utiliser ce dictionnaire pour stocker des informations : on parle alors de **boîte noire**.

Notre fonction récursive s'écrira donc `Fibo(n,D={})` : le premier appel crée par défaut un dictionnaire vide, que les appels récursifs imbriqués feront passer ensuite en argument et se chargeront de remplir.

```
def Fibo(n,D={}):
    if n in D:
        return D[n]
    elif n==0:
        D[0]=0
    elif n==1:
        D[1]=1
    else:
        D[n]=Fibo(n-1,D)+Fibo(n-2,D)
    return D[n]
```

C'est très efficace ... et beau ! Notons que deux cas se posent lors de l'appel de `Fibo(n,D)` :

- Si  $n$  est présent dans  $D$ , on renvoie  $D[n]$ .
- Sinon on calcule récursivement `Fibo(n,D)` avant de stocker sa valeur dans  $D$ .

**Remarque :** la même idée permet de calculer récursivement les coefficients binomiaux en utilisant la relation de Pascal et les cas terminaux  $\binom{n}{n} = \binom{n}{0} = 1$  et  $\binom{n}{n-1} = \binom{n}{1} = n$ .

## 2/ Terminaison, correction et complexité

### 2.1) Preuve d'un algorithme

#### Terminaison

Pour montrer qu'un algorithme se termine, on cherche un **variant de boucle**. Dans le cas d'une fonction récursive, c'est son argument (ou la taille de celui-ci).

**Exemple :** pour `facto(n)`, c'est l'entier  $n$ . Mais éviter de calculer `facto(n)` avec  $n \notin \mathbb{N}$ !

#### Correction

Elle se démontre par récurrence simple, double ou même forte selon les cas, en utilisant un **invariant de boucle**.

**Exemple :** montrer que la proposition  $\mathcal{P}(n)$  : « `facto(n) = n!` » est vraie pour tout  $n \in \mathbb{N}$ .

### 2.2) Premiers calculs de complexité

Nous allons évaluer sur trois exemples simples le nombre  $C(n)$  d'opérations arithmétiques élémentaires effectuées pour un argument de valeur  $n$ .

#### Premier exemple

Considérons le calcul de  $n!$  avec `facto(n)`. Alors  $C(0) = 0$  et

$$\forall n \geq 1 \quad C(n) = C(n-1) + 1$$

On en déduit que  $C(n) = n$ , d'où une complexité en  $\mathcal{O}(n)$  comme pour la version itérative.

#### Deuxième exemple

Reprenons la fonction récursive vue en introduction pour la méthode de Héron. On a encore  $C(0) = 0$  et

$$\forall n \geq 1 \quad C(n) = 2C(n-1) + 3$$

On reconnaît une suite arithmético-géométrique, dont le terme général est  $C(n) = 3(2^n - 1)$  après savants calculs. Ainsi, la complexité est en  $\mathcal{O}(2^n)$ .

#### Deuxième exemple - amélioration

Pour éviter d'appeler deux fois `u(n-1)`, il suffit simplement de stocker sa valeur dans une variable locale. On remplace donc l'instruction `return 0.5*(u(n-1)+3/u(n-1))` par :

```
x=u(n-1)
return 0.5*(x+3/x)
```

On a toujours  $C(0) = 0$  et

$$\forall n \geq 1 \quad C(n) = C(n-1) + 3$$

si bien que  $C(n) = 3n$ . De ce fait, la complexité est en  $\mathcal{O}(n)$  : le progrès est incontestable.

## Conclusion

Pour déterminer les complexités de nos fonctions récursives, on établit les relations de récurrence qu'elles vérifient, puis on les résout si l'on peut. On a pu voir sur un exemple simple le phénomène d'explosion combinatoire : une multiplication des appels récursifs transforme une complexité linéaire en complexité exponentielle.

**Remarque :** on montre de même que la version récursive naïve de la suite de Fibonacci a une complexité en  $\mathcal{O}(\varphi^n)$  – où  $\varphi$  est le nombre d'or – alors que celle de l'itérative est en  $\mathcal{O}(n)$ .

## 2.3) La méthode « diviser pour régner »

### Définition

La méthode **diviser pour régner** consiste, lors de la résolution d'un problème, à :

- ❶ découper le problème principal en sous-problèmes de tailles plus petites
- ❷ résoudre (récursivement) chacun de ces sous-problèmes
- ❸ recomposer la solution du problème principal à l'aide des solutions des sous-problèmes

**Exemple :** tous les algorithmes utilisant le principe de dichotomie rentrent dans ce cadre.

Cette méthode est d'un usage fréquent en informatique et on la retrouvera dans les algorithmes de tri les plus efficaces (tri rapide ou tri fusion). Pour calculer les complexités de tels algorithmes, on aura besoin du résultat suivant :

### Théorème (sur les récurrences de partition)

Supposons qu'il existe  $a, b \in \mathbb{N}^*$  et  $\alpha \geq 0$  tels que la complexité  $C(n)$  vérifie

$$\boxed{C(n) = a C(\lfloor n/b \rfloor) + \mathcal{O}(n^\alpha)} \quad \text{ou} \quad \boxed{C(n) = a C(\lfloor n/b \rfloor + 1) + \mathcal{O}(n^\alpha)}$$

Trois cas peuvent alors se présenter :

- Si  $a > b^\alpha$ , alors  $C(n) = \mathcal{O}(n^{\log_b(a)})$
- Si  $a = b^\alpha$ , alors  $C(n) = \mathcal{O}(n^\alpha \ln n)$
- Si  $a < b^\alpha$ , alors  $C(n) = \mathcal{O}(n^\alpha)$

**Exemple :** reprenons l'algorithme d'exponentiation rapide. Sa complexité  $C(n)$  vérifie alors la relation de récurrence

$$C(n) = C(\lfloor n/2 \rfloor) + \mathcal{O}(1)$$

On est ainsi dans le second cas du théorème et  $C(n) = \mathcal{O}(\ln n)$ .

On se convaincra sans trop de peine que les complexités des algorithmes dichotomiques vont toutes satisfaire *a priori* la relation de récurrence ci-dessus. On aura donc à chaque fois des complexités logarithmiques, ce qui montre tout l'intérêt de ces algorithmes par rapport aux algorithmes itératifs naïfs, qui sont eux complexités linéaires.

## 2.4) Un exemple fameux

### Le problème

Soit  $n \in \mathbb{N}^*$ . On considère deux polynômes  $P = \sum_{i=0}^{n-1} p_i X^i$  et  $Q = \sum_{j=0}^{n-1} q_j X^j$  de  $\mathbb{R}_{n-1}[X]$ .

Ils sont représentés par les listes  $[p_0, \dots, p_{n-1}]$  et  $[q_0, \dots, q_{n-1}]$  de leurs coefficients respectifs. Leur produit  $P \times Q$  est un polynôme de  $\mathbb{R}_{2n-2}[X]$  dont on va calculer les  $2n - 1$  coefficients.

### L'approche naïve

Avec les notations précédentes, on a par définition du produit polynomial

$$P \times Q = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} p_i q_j X^{i+j} = \sum_{k=0}^{2n-2} \left( \sum_{i+j=k} p_i q_j \right) X^k = \sum_{k=0}^{2n-1} c_k X^k$$

où  $\forall k \in \llbracket 0; 2n-2 \rrbracket \quad c_k = \sum_{i+j=k} p_i q_j = \sum_{i=\max(k+1-n, 0)}^{\min(k, n-1)} p_i q_{k-i}$

On obtient ainsi un algorithme de complexité  $\mathcal{O}(n^2)$ , assez facile à écrire en Python.

### L'algorithme de Karatsuba

Lorsque  $n = 1$ , les polynômes sont des constantes que l'on va multiplier « bêtement ». Lorsque  $n > 1$ , on pose  $m = \lfloor n/2 \rfloor$  et l'on divise les deux polynômes par  $X^m$  : ainsi,

$$P = X^m P_1 + P_0 \quad \text{et} \quad Q = X^m Q_1 + Q_0$$

avec  $P_0, P_1, Q_0$  et  $Q_1 \in \mathbb{R}_m[X]$ . En effet, on a  $n = 2m + 1$  ou bien  $n = 2m$  donc  $P, Q \in \mathbb{R}_{2m}[X]$ . Introduisons les polynômes  $T_0 = P_0 Q_0$ ,  $T_1 = P_1 Q_1$  et  $T_2 = (P_0 + P_1)(Q_0 + Q_1)$ , de sorte que

$$\begin{aligned} PQ &= (X^m P_1 + P_0)(X^m Q_1 + Q_0) \\ &= X^{2m} P_1 Q_1 + X^m (P_0 Q_1 + P_1 Q_0) + P_0 Q_0 \\ &= X^{2m} T_1 + X^m (T_2 - T_1 - T_0) + T_0 \end{aligned}$$

Notons  $C(n)$  la complexité de cet algorithme. On vient de voir que pour calculer  $P \times Q$  :

- ❶ on réalise deux divisions euclidiennes et deux additions
- ❷ on effectue trois multiplications de polynômes de taille  $m + 1$
- ❸ on effectue deux multiplications par  $X^m$  et  $X^{2m}$ , et deux additions

La première et la dernière étapes ont une complexité en  $\mathcal{O}(n)$  tandis que celle de la deuxième est  $3C(\lfloor n/2 \rfloor + 1)$ . De ce fait,

$$C(n) = 3C(\lfloor n/2 \rfloor + 1) + \mathcal{O}(n)$$

On se retrouve ainsi dans le premier cas du théorème et  $C(n) = \mathcal{O}(n^{\log_2(3)})$ .

On programmera les deux algorithmes en exercice et l'on comparera les temps de calcul sur des polynômes de grands degrés générés de manière aléatoire en utilisant `randint`.

### 3/ Quelques algorithmes incontournables

#### 3.1) Recherche dichotomique dans une liste triée

Voici comment savoir si un élément donné  $x$  appartient ou non à une liste triée  $L$  :

- On prend une valeur  $a$  au milieu de cette liste.
- Si  $a = x$ , on a fini. Sinon, on réitère le procédé avec :
  - la première moitié de la liste si  $a > x$
  - la seconde moitié si  $a < x$ .

Si  $x \in L$ , on finit par tomber dessus ; sinon, on obtient une liste vide. Cet algorithme conduit ainsi à une première fonction récursive utilisant des sous-listes. Elle est peu efficace car l'extraction d'une sous-liste est une opération de complexité linéaire.

Pour remédier à ce problème, on utilise une fonction auxiliaire récursive qui prend en argument deux entiers, les indices extrêmes de la liste considérée (le dernier étant exclu).

```
def dichom(L,x):
    n=len(L)
    if n==0:
        return False
    m=n//2
    if L[m]==x:
        return True
    elif L[m]<x:
        return dichom(L[m+1:],x)
    else:
        return dichom(L[:m],x)
```

```
def dichob(L,x):
    def aux(g,d):
        if g>=d:
            return False
        m=(g+d)//2
        if L[m]==x:
            return True
        elif L[m]<x:
            return aux(m+1,d)
        else:
            return aux(g,m)
    return aux(0,len(L))
```

Les complexités de ces deux fonctions vérifient respectivement

$$C_1(n) = C_1(\lfloor n/2 \rfloor) + \mathcal{O}(n) \quad \text{et} \quad C_2(n) = C_2(\lfloor n/2 \rfloor) + \mathcal{O}(1)$$

de sorte que  $C_1(n) = \mathcal{O}(n)$  et  $C_2(n) = \mathcal{O}(\ln n)$  : effectivement, la première fonction est bien moins efficace que la seconde.

#### 3.2) Les tours de Hanoï

C'est un jeu inventé à la fin du XIX<sup>e</sup> siècle. Le principe en est le suivant :

- On dispose de  $n$  disques troués de tailles distinctes et de trois piquets notés A, B et C.
- Initialement, les  $n$  disques sont enfilés sur le piquet A par diamètre décroissant.
- On doit les enfiler dans le même ordre sur le piquet C, en utilisant éventuellement le piquet B, tout en respectant les règles suivantes :
  - ❶ on déplace les disques un par un, d'un piquet à l'autre ;
  - ❷ un disque ne peut pas être posé sur un disque de diamètre inférieur.

Une solution récursive est particulièrement adaptée à ce problème. En effet, pour déplacer les  $n$  disques du piquet A au piquet C, il suffit de :

- ne rien faire si  $n = 0$  ;
- si  $n \geq 1$ , procéder comme suit :
  - ❶ déplacer les  $n - 1$  disques du dessus de A à B ;
  - ❷ déplacer le  $n^{\text{e}}$  disque de A à C ;
  - ❸ déplacer les  $n - 1$  disques de B à C.

C'est bien la définition d'une fonction récursive qu'on programmera en exercice. En ce qui concerne sa complexité, notons  $D(n)$  le nombre de déplacements à effectuer avec  $n$  disques. Alors  $D(0) = 0$  et

$$\forall n \geq 1 \quad D(n) = 2D(n-1) + 1$$

si bien que  $D(n) = 2^n - 1$ . La complexité est élevée, mais comment procéder itérativement ?

### 3.3) Les fractales

**Définition** *Une figure fractale est un objet mathématique, telle une courbe ou une surface, dont la structure est invariante par changement d'échelle. Un tel objet possède une structure autosimilaire, c'est-à-dire que l'ensemble est semblable à chacune de ses parties.*

Dans la nature, on rencontre des formes fractales approximatives, qui possèdent une structure autosimilaire sur une échelle étendue mais finie, comme par exemple les flocons de neige, le brocoli ou encore le chou Romanesco.

Nous allons étudier l'exemple de la courbe de Von Koch, initialement introduite afin d'illustrer l'existence de fonctions continues en tout point mais dérivables en aucun point. Voici comment la construire :

- On part d'un segment  $[AB]$  de longueur donnée.
- On le divise en trois parties égales  $[AC]$ ,  $[CE]$  et  $[EB]$ .
- On construit un triangle équilatéral direct CED sans base au-dessus du segment  $[CE]$ .
- On réitère ensuite ce procédé pour chacun des segments  $[AC]$ ,  $[CD]$ ,  $[DE]$  et  $[EB]$ .

Les points seront représentés en Python à l'aide de nombres complexes (leurs affixes) et la fonction  $VK(a,b,n)$ , qui applique récursivement la transformation ci-dessus à chaque segment, va renvoyer une liste de nombres complexes.

```
def transfo_Koch(a,b):
    h=(b-a)/3
    c=a+h
    e=c+h
    d=c+h*(1+1j*3**0.5)/2
    return (c,d,e)
```

```
def VK(a,b,n):
    if n==0:
        return [a,b]
    [c,d,e]=transfo_Koch(a,b)
    return VK(a,c,n-1)+VK(c,d,n-1)
    +VK(d,e,n-1)+VK(e,b,n-1)
```

Si l'on construit trois fois cette courbe en partant des côtés d'un triangle équilatéral, on obtient une étoile puis un flocon de plus en plus complexe : c'est le **flocon de Von Koch**.

# Exercices

## Exercice 1.

Écrire l'algorithme d'Euclide sous forme récursive en utilisant uniquement des soustractions. Pour cela, on observera que pour tous  $a$  et  $b \in \mathbb{N}$  :

- on a  $\text{pgcd}(a, b) = \text{pgcd}(b, a)$
- si  $a \geq b$ , alors  $\text{pgcd}(a, b) = \text{pgcd}(a - b, b)$
- enfin,  $\text{pgcd}(a, 0) = a$

## Exercice 2.

On cherche à déterminer la liste de toutes les permutations d'une liste donnée.

- 1) On suppose que l'on dispose de la liste P des permutations de la liste  $[b, c, d]$ .  
Donner le code Python permettant d'obtenir la liste Q des permutations de  $[a, b, c, d]$   
*Indication : l'instruction  $L[i:i]=[x]$  permet d'insérer  $x$  dans la liste L à la position  $i$ .*
- 2) Écrire une fonction récursive qui prend en argument une liste L et renvoie la liste de toutes les permutations de L.

## Exercice 3.

Calculer la suite de Fibonacci en mettant la relation de récurrence sous forme matricielle et en calculant les puissances de matrices avec la méthode d'exponentiation rapide.

## Exercice 4.

- 1) Écrire une fonction Comparaison( $n$ ) affichant les temps de calcul du terme d'indice  $n$  de la suite de Fibonacci à l'aide des différentes méthodes vues en cours.
- 2) L'utiliser pour  $n = 20$ ,  $n = 25$  et  $n = 30$ .  
*On pensera à la fonction `time` de la bibliothèque `time`.*

## Exercice 5.

Calculer récursivement les coefficients binomiaux en utilisant la méthode de mémorisation : les valeurs calculées seront stockées dans un dictionnaire de dictionnaire D construit de manière à ce que l'on ait  $D[n][p] = \binom{n}{p}$ .

Pour tester la validité et l'efficacité, on affichera le triangle de Pascal jusqu'à  $n = 10$ , ainsi que les valeurs des coefficients binomiaux  $\binom{2n}{n}$  pour  $n \in \llbracket 0; 25 \rrbracket$ .

**Exercice 6.**

Reprenons la suite de Héron vue en introduction.

- 1) Corriger la fonction récursive proposée dans le cours.
- 2) Procéder comme à l'exercice 2 pour comparer les vitesses de calcul des trois fonctions rencontrées pour  $n = 8$ ,  $n = 16$  et  $n = 24$ .
- 3) Même chose pour  $n = 100$ ,  $n = 300$  et  $n = 900$  en oubliant la fonction inopérante.

**Exercice 7.**

Programmer la multiplication naïve de deux polynômes de  $\mathbb{R}_{n-1}[X]$  vue en cours.  
Afficher le temps de calcul pour des polynômes aléatoires dans le cas où  $n = 5000$ .

**Exercice 8.**

Procéder de même en utilisant cette fois l'algorithme de Karatsuba.

**Exercice 9.**

Écrire une fonction `Hanoi(n, i, k, j)` qui renvoie la liste détaillée des actions à effectuer afin de déplacer  $n$  disques du piquet  $i$  au piquet  $k$ , le dernier étant le piquet  $j$ .

Pour signaler le déplacement d'un disque de  $A$  vers  $B$ , on affichera  $A \rightarrow B$ .

Ainsi, `Hanoi(2, 'A', 'C', 'B')` doit produire  $A \rightarrow B$ ,  $A \rightarrow C$  puis  $B \rightarrow C$ .

**Exercice 10.**

La *courbe du Dragon* est définie de la manière suivante :

- on part d'un segment  $[AB]$  – attention, l'ordre des points est important ;
- on place le point  $C$  tel que  $ABC$  soit un triangle direct isocèle et rectangle en  $C$  ;
- on efface le segment  $[AB]$  et on trace les segments  $[AC]$  et  $[BC]$  ;
- on réitère alors le procédé avec ces deux nouveaux segments.

Construire récursivement cette courbe, en s'inspirant de celle de Von Koch.

**Exercice 11.**

Le *triangle de Sierpinsky* se construit comme ceci :

- on part d'un triangle équilatéral rempli ;
- on le coupe en quatre en traçant les segments reliant les milieux de ses côtés ;
- on enlève ensuite la partie centrale ;
- on répète le procédé avec les trois triangles restants.

Construire cette figure de manière récursive.

Pour remplir une figure, il suffit d'utiliser `fill` au lieu de `plot`.

**Exercice 12.**

Dessiner de même le *tapis de Sierpinsky*, obtenu en coupant cette fois un carré en neuf parties puis en supprimant le carré central.