# TP 10 Python – Récursivité

## 1. La récursivité – cours

Il existe différentes manières de programmer un même algorithme. La façon de programmer basée sur les boucles (while ou for) étudiée jusqu'à maintenant est appelée programmation **itérative**.
On introduit ici une nouvelle manière de programmer : la programmation **récursive**.
Cette façon permet d'écrire un certain nombre de programmes plus simplement et *de façon plus concise*.
Cependant, elle a plusieurs inconvénients à ne pas négliger : elle nécessite de faire très attention à ne pas avoir de "boucle infinie", et à ne pas faire des calculs redondants qui pourraient faire exploser le temps d'exécution. De plus, le langage Python est mal adapté pour ce type de programmation.

Le concept de fonction récursive est assez proche de la notion mathématique de **récurrence.**

**Exemple**

On peut coder la fonction factorielle de la manière suivante :

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

On a toujours un ou plusieurs "cas de base", ici $n = 0$. Pour les autres valeurs de l'argument, on utilise la fonction elle-même, appliquée à un argument "plus petit". L'idée est de se ramener au cas de base, étape par étape.
Par exemple, ici, pour calculer `factorielle(5)`, Python doit d'abord calculer `factorielle(4)`. Mais il doit alors calculer `factorielle(3)`, etc. il "remonte" alors la *pile d'appels*, à partir de la valeur connue `factorielle(0)`, jusqu'à la valeur demandée `factorielle(5)`.
Concrètement, on a utilisé la définition par récurrence suivante : $0! = 1$ et pour tout $n \in \mathbb{N}^*, n! = n \times (n-1)!$.

**Définition**

On dit qu'une fonction (informatique) est **récursive** si, dans certains cas, **elle s'appelle elle-même**, avec de nouveaux paramètres (souvent plus petits). On appelle cela un **appel récursif**.

*Remarque.* ⚠ Une fonction récursive doit toujours comprendre au moins un cas terminal qui ne demande pas d'appel récursif.
Si elle est bien programmée, la fonction récursive doit toujours finir par aboutir à un tel cas, faute de quoi votre fonction part en récursivité infinie !

**Exemple**

```
def somme_liste(L):
    if L == [] :  # Condition d'arrêt
        return 0
    else :
        return L[0] + somme_liste(L[1:])  # Appel récursif
```

Ici, l'appel récursif s'effectue avec une liste qui contient un élément de moins (on retire le premier élément).
Les appels successifs aboutissent donc nécessairement une liste vide, dont la somme vaut 0, avant de "remonter" jusqu'à la liste d'origine.

## 2. Exercices de récursivité

⚠ Dans tout ce TP, on écrira exclusivement des fonctions **récursives**, sans boucles for ni boucles while !

**Exercice 1.** Pour tout $n \in \mathbb{N}$, on pose $S_n = \sum_{k=0}^{n} k^5$.

1. Que vaut $S_0$ ?
2. Exprimer $S_n$ en fonction de $S_{n-1}$ pour tout $n \in \mathbb{N}^*$.
3. En déduire une fonction *récursive* (sans boucle for !) qui prend en argument un entier naturel $n$ et qui renvoie la valeur de la somme $S_n$.

**Exercice 2.** Écrire une fonction `produit_liste(L)` qui prend en argument une liste $L$ et qui renvoie le produit de tous ses éléments.

**Exercice 3.** On considère la suite définie par $u_0 = 1$ et, pour tout $n \in \mathbb{N}$, $u_{n+1} = \dfrac{1}{1+u_n^2}$.
Écrire une fonction qui prend en argument un entier naturel $n$ et qui renvoie la valeur de $u_n$.

**Exercice 4.** On rappelle la suite de Fibonacci : $F_0 = 0$, $F_1 = 1$, et $\forall n \in \mathbb{N}, F_{n+2} = F_{n+1} + F_n$.

1. Écrire une fonction (récursive) `Fibo_rec(n)` qui prend en argument un entier naturel $n$ et qui renvoie la valeur de $F_n$. *Indication : Combien faut-il de "cas de base" ici ?*
2. Écrire une fonction itérative `Fibo_iter(n)` qui fait de même avec une boucle `for` (objectif : l'écrire en moins d'une minute !)
3. Comparez le temps d'exécution de `Fibo_rec(40)` et de `Fibo_iter(40)` (faites preuve de patience)
4. Proposez une explication.
5. Pour résoudre ce problème, on garde en mémoire les valeurs déjà calculées, en calculant plutôt la liste de tous les termes de $F_0$ à $F_n$. Écrire la version récursive de cette fonction, `liste_Fibo_rec(n)`. Que pensez-vous de son temps d'exécution pour $n = 40$ ?

**Exercice 5.** On définit la suite de Catalan $(C_n)_{n\in\mathbb{N}}$ par :

$$C_0 = 1, \text{ et pour tout } n \in \mathbb{N}, \quad C_{n+1} = \sum_{k=0}^{n} C_k C_{n-k}$$

1. Écrire une fonction `Catalan(n)`, qui prend en paramètre un entier naturel $n$, et qui renvoie la valeur de $C_n$. On s'autorisera une boucle for pour le calcul de la somme !
2. Tester la fonction avec $n = 10$ ($C_{10} = 16796$) et $n = 16$. Que pensez vous de son temps d'exécution ?
3. Sur le modèle de l'exercice précédent, écrire une fonction améliorée (mais toujours récursive), qui fait le travail plus rapidement.

**Exercice 6.** On souhaite écrire une fonction `fusion(L1, L2)` qui prend en paramètre deux listes de nombres supposées **triées dans l'ordre croissant**, et qui renvoie une liste qui contient tous les éléments de $L1$ et de $L2$, encore triés dans l'ordre croissant.
Par exemple, `fusion([3,5,6,9], [5,7,11])` doit renvoyer `[3, 5, 5, 6, 7, 9, 11]`

1. Que renvoie la fonction `fusion` si l'une des deux listes est vide ?
2. On suppose que les deux listes sont non vides et que le premier élément de $L1$ est plus petit que celui de $L2$. Alors la fonction doit renvoyer `[L1[0]]+fusion(....)` (à compléter).
3. À l'aide des éléments ci-dessus, écrire la fonction `fusion`.
4. (**) En déduire une fonction `tri_fusion(L)` qui prend en paramètre une liste de nombres $L$, et qui renvoie la liste qui contient les mêmes éléments triés dans l'ordre croissant.
   L'idée est de "couper en 2" la liste $L$, d'appliquer la fonction `tri_fusion` aux deux petites listes, puis de les fusionner.
   NB : C'est un algorithme de tri optimal en terme de "complexité", c'est-à-dire de nombre de d'opérations effectuées !

### Complément (à faire après les fractales !)

**Exercice 7.**

Écrire une fonction `motsAB(n)` qui renvoie une liste contenant tous les mots de longueur $n$ qu'on peut écrire en utilisant seulement les lettres "a" et "b".

Écrire une fonction `motsABsansAA(n)` qui renvoie une liste contenant tous les mots de longueur $n$ qu'on peut écrire en utilisant seulement les lettres "a" et "b", **sans jamais** avoir deux "a" consécutifs.

Écrire une fonction `Parties(n)` qui renvoie une liste contenant toutes les parties de $[\![1, n]\!]$ (sans répétition). On représentera les parties de $[\![1, n]\!]$ par la liste des éléments qu'elle contient.

**Exercice 8.** On souhaite comparer deux mots pour savoir s'ils sont ou non dans l'ordre alphabétique.
On a écrit pour cela au TP8 une fonction `numéro(lettre)` qui associe à chaque lettre sa position dans l'alphabet (a↔1, b↔2, etc.), afin de pourvoir facilement comparer deux lettres :

```
alphabet = "abcdefghijklmnopqrstuvwxyz"
def numéro(c):
    for i in range(26):
        if c == alphabet[i]:
            return i+1
```

Pour la fonction `ordre_alpha(mot1, mot2)`, on peut utiliser l'idée suivante :

- Si la première lettre du mot1 est différente de celle du mot2, on conclut facilement.
- Si les premières lettres sont les mêmes, il faut comparer les mots auxquels on a enlevé la première lettre (à l'aide du slicing `mot[1:]`).
- Attention ! On risque d'arriver à un mot vide pendant ce procédé… Le cas où l'un des mots est vide doit donc apparaître dans les "cas de base" de la fonction.

À vous de jouer !

## 3. Fractales

Une figure fractale est un objet mathématique qui présente une structure similaire à toutes les échelles. En zoomant sur une partie de la figure, il est possible de retrouver toute la figure ; on dit alors qu'elle est « autosimilaire ». La définition de ces structures se fait donc naturellement de manière récursive, mais infinie.

On en donne dans la suite des exemples classiques, qu'on dessinera du mieux possible en Python à l'aide de fonctions récursives et du module `matplotlib.pyplot`.

**Matplotlib.pyplot**

`import matplotlib.pyplot as plt`

`plt.plot(X,Y,'+-r')` ---- Génère la courbe des points définis par les listes `X` et `Y` (abscisses et ordonnées) avec les options :
- symbole : `'.'` point, `'o'` rond, `'h'` hexagone, `'+'` plus, `'x'` croix, `'*'` étoile, …
- ligne : `'-'` trait plein, `'--'` pointillé, `'-.'` alterné, …
- couleur : `'b'` **bleu** , `'r'` **rouge** , `'g'` **vert** , `'c'` **cyan** , `'m'` **magenta** , `'k'` **noir** , …

`plt.bar(X,Y)` ---------- Génère l'histogramme des points définis par les listes `X` et `Y` (abscisses et ordonnées)

`plt.axis('equal')` ------ Rend le repère orthonormé

`plt.xlim(xmin,xmax)` ---- Fixe les bornes de l'axe des abscisses

`plt.ylim(ymin,ymax)` ---- Fixe les bornes de l'axe des ordonnées

`plt.show()` ------------ Affiche le graphique

Dans ce TP, seules les commandes suivantes seront utiles : `plt.plot(X, Y, "k")`, `plt.axis("equal")` et `plt.show()`.

On utilisera aussi la fonction `plt.fill(X, Y, "k")`, qui permet de colorier la zone délimitée par les points associés aux listes $X$ et $Y$.

### 3.A) Triangle de Sierpinski

Le triangle de Sierpinski est construit de la manière suivante :

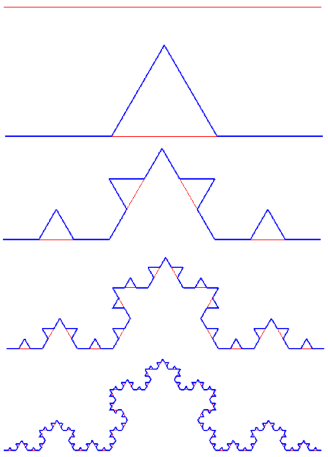
On remarquera que l'étape $n+1$ est toujours constituée de 3 petits triangles de l'étape $n$ disposés en pyramide.
C'est de cette manière qu'on construit la fonction récursive `triangle_sierp`.
Cette dernière doit donc prendre en argument non seulement le numéro de l'étape, mais aussi un "point de départ" sous la forme d'un couple de coordonnée $(x, y)$ et une longueur pour le côté du triangle.

À vous d'écrire cette fonction ! Pour vous aider, vous pouvez répondre aux questions suivantes :

1. Quel est le "cas de base" ? Comment le réaliser ?
2. Si je veux construire le triangle de l'étape $n$, à partir du point de départ $(x, y)$ :
   (a) Quelle doit être la longueur des petits triangles de l'étape $n-1$ ?
   (b) Quels sont les coordonnées de leurs point de départ respectifs ?

### 3.B) Flocon de Koch

La courbe de Koch est construite de la manière ci-contre.
L'étape 0 est seulement un segment, et pour construire la courbe à l'étape $n+1$, on met bout à bout 4 courbes de l'étape $n$... mais il faut choisir soigneusement le point de départ et le point d'arrivée !



Pour ce faire, on aura besoin des fonctions suivantes :

1. Écrire une fonction `trace_segment(M, N)` qui prend en argument deux point (donné chacun comme couple de coordonnées), et qui trace le segment qui les relie.
2. Écrire une fonction `divise_en_3(A, D)` qui prend en argument deux points et qui renvoie les coordonnées des deux points $B$ et $C$ tels que $\overrightarrow{AB} = \overrightarrow{BC} = \overrightarrow{CD}$ (on divise le segment $[AD]$ en trois parts égales).
3. On admet le fonctionnement de la fonction ci-dessous, qui prend en argument deux points $A$ et $B$, et qui renvoie le point $C$ tel que $ABC$ est un triangle équilatéral parcouru dans le sens direct.

```
def sommet_manquant_equilateral(A,B):
    xA, yA = A
    xB, yB = B
    xC = (xB + xA - sqrt(3)*(yB - yA)) / 2
    yC = (yB + yA + sqrt(3)*(xB + xA)) / 2
    return xC, yC
```

À l'aide des fonctions ci-dessus, écrire une fonction récursive `courbe_koch(départ, arrivée, n)` qui trace la courbe de Koch à l'étape $n$ partant du point départ et arrivant au point arrivée.

En déduire une fonction `flocon_de_koch(n)` qui trace le flocon de Koch à l'étape $n$.