# Récursivité

## Table des matières



## 1] Contexte : la somme des premiers entiers

Pour définir la somme des $n$ premiers entiers, on a l'habitude de d'écrire la formule suivante :

$$0 + 1 + 2 + 3 + .... + n$$

Cette formulation peut nous apparaître simple et intuitive.

**Exercice 1**

Écrire une fonction `somme(n)` qui retourne la somme des $n$ premiers entiers.

Or ce code n'est pas directement lié à la formule précédente. En effet, il n'y a rien dans cette formule qui laisse penser qu'il faille une variable intermédiaire pour calculer cette somme.

C'est pourquoi, nous allons tenter de définir autrement cette fonction :

$$somme(n) = \begin{cases} 0 & si \quad n = 0 \\ n + somme(n-1) & si \quad n > 0 \end{cases}$$

Cette définition nous indique ce que vaut `somme(n)` pour un entier $n$ quelconque, selon que $n$ soit égal à 0 ou strictement positif. Ainsi, pour, pour le cas où $n$=0, la valeur de `somme(n)` vaut 0, et dans le cas où $n$ est strictement positif, la valeur de `somme(n)` est $n + somme(n-1)$.

Par exemple, voici ci-dessous les valeurs de `somme(n)`, pour n valant 0,1,2 et 3.

$$somme(0) = 0$$
$$somme(1) = 1 + somme(0) = 1 + 0 = 1$$
$$somme(2) = 2 + somme(1) = 2 + 1 = 3$$

L'intérêt de cette formulation est qu'elle est directement programmable. En python, cela donne le programme 1.

```python
def somme(n:int)-> int:
    if n == 0:
        return 0
    else :
        return n + somme(n-1)
```

Listing 1 – Version récursive de la fonction somme

# 2] Récursivité

## 2.1) Définition

On dit qu'un sous-programme (procédure ou fonction) est récursif s'il s'appelle lui-même

Il est indispensable de prévoir une condition d'arrêt à la récursion sinon la fonction va s'appeler une infinité de fois. Dans la pratique, la pile qui stocke les appels récursifs est de taille finie. Une fois qu'elle est pleine, le programme ne répondra plus

**Exercice 2**

Donner une définition récursive qui correspond au calcul de la fonction factorielle $n!$ définie par $n! = 1 \times 2 \times 3 \times ... \times n$ si $n > 0$ et $0! = 1$. Écrire également le code d'une fonction `fact(n)` qui implémente cette définition. Pour les plus rapides, écrire la version itérative.

**Exercice 3**

Écrire une fonction récursive `boucle(i,k)` qui affiche les entiers entre `i` et `k`. Par exemple, `boucle(0,3)` affiche 0,1,2,3. Pour les plus rapides, écrire la version itérative.

**Exercice 4**

La méthode du paysan russe est un très vieil algorithme de multiplication de deux nombres entiers déjà décrit, sous forme légèrement différente, sur un papyrus égyptien rédigé autour de 1650 avant J.C. Il s'agissait de la principale méthode de calcul en Europe avant l'introduction des chiffres arabes.
Les ordinateurs l'ont utilisé avant que la multiplication ne soit directement intégré dans le processeur. Sous une forme moderne, il peut être décrit par le programme donné dans l'algorithme 2 donné ci-dessous.

1. Appliquer cette fonction pour effectuer la multiplication de 105 par 253. Détailler les étapes dans le tableau suivant :

| x | y | p | ligne |
|---|---|---|---|
| 105 | 253 | .... | .... |

2. On admet que cet algorithme repose sur les relations suivantes :

$$x * y = \begin{cases} 0 & si \quad x = 0 \\ (x//2) * (y + y) & si \quad x \ est \ pair \\ (x//2) * (y + y) + y & si \quad x \ est \ impair \end{cases}$$

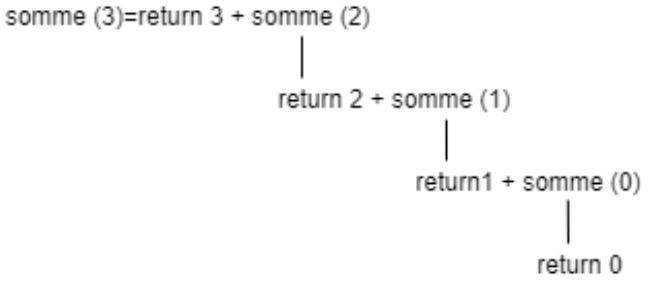
Proposer une version récursive de cet algorithme.

```
Fonction multiplication(x,y):
    p=0
    TANT QUE x>0:
        Si x est impair:
            p=p+y
        x=x//2
        y=y+y
    Retourner p
```

Listing 2 – Pseudo-code de la fonction itérative

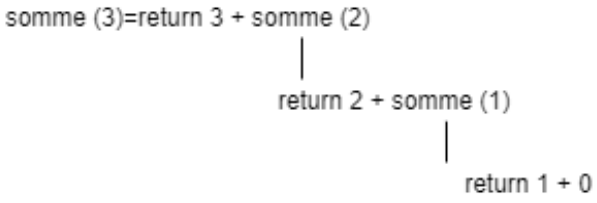
## 2.2) Fonctionnement d'une fonction récursive

Par exemple, l'évaluation de l'appel à `somme(3)` du programme 1 peut se représenter de la manière suivante :

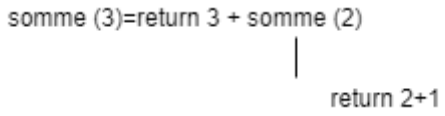



où on indique uniquement pour chaque appel à `somme(n)` l'instruction qui est exécutée après le test `n==0` de la conditionnelle. Cette manière de représenter l'exécution d'un programme en indiquant les différents appels effectués est appelée un *arbre d'appels*

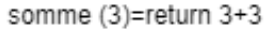
Ainsi, pour calculer la valeur renvoyée par `somme(3)`, il faut tout d'abord appeler `somme(2)`. Cet appel va lui-même déclencher un appel à `somme(1)`, qui à son tour nécessite un appel à `somme(0)`. Ce dernier appel se termine en renvoyant la valeur 0. Le calcul de `somme(3)` se fait donc "à rebours". Une fois que l'appel à `somme(0)` est terminé, c'est-à-dire que la valeur 0 a été renvoyée, l'arbre d'appels a la forme suivante.




A cet instant, l'appel à `somme(1)` peut alors se terminer et renvoyer le résultat de la somme 1+0. L'arbre d'appels est alors le suivant :

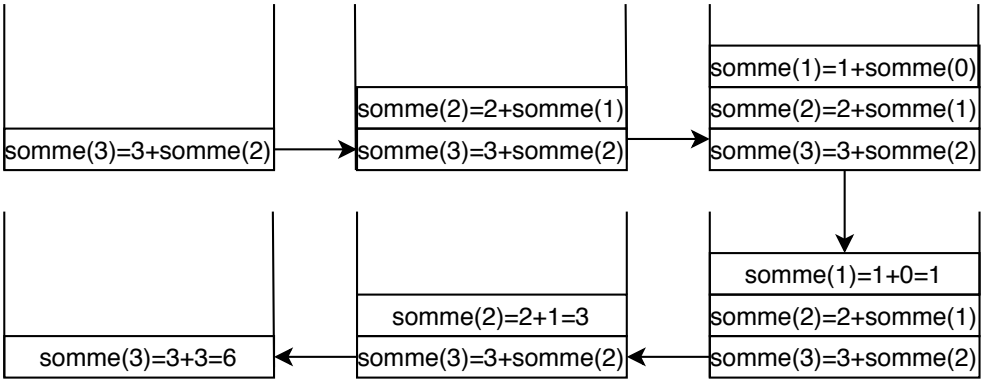



Enfin, l'appel à `somme(2)` peut lui-même renvoyer la valeur 2+1 comme résultat, ce qui permet à `somme(3)` de se terminer en renvoyant le résultat 3+3.



On obtient bien au final la valeur 6 attendue.

**Autre Représentation : sous forme de pile d'exécution**




## 2.3) Récursivité ou itérativité ?

Lorsque l'on programme des fonctions qui ne s'appellent pas, on dit que l'on programme de manière itérative. Il est toujours possible de transformer une fonction itérative en fonction récursive et vice et versa. La méthode itérative nous est plus familière et est plus rapide une fois le code implémenté dans un langage de programmation.

La méthode récursive est plus élégante et lisible et évite d'utiliser de nombreuses structures itératives. Elle est également très utile pour concevoir des structures de données complexes comme les listes, les arbres et les graphes. L'inconvénient le plus important de cette méthode, est qu'une fois implémentée dans un langage de programmation, elle est très gourmande en ressource mémoire. Du fait que l'on empile, tous les appels récursifs, des débordements de capacité peuvent se produire lorsque la pile est pleine.

## 2.4) Récursivité croisée

Dans cette méthode récursive, il arrive qu'une fonction appelle une autre fonction qui appelle elle-même la première, ce cas est appelée récursivité croisée. Prenons, par exemple deux fonctions ci-dessous permettant de tester si un nombre est pair ou impair comme dans le programme 3.

```
def Pair(n):
    if n==0:
        return True
    else :
        return Impair(n-1)

def Impair(n):
    if n==0:
        return False
    else :
        return Pair(n-1)
```

Listing 3 – Exemple de récursivité croisée

Ce n'est évidemment pas la méthode la plus simple mais elle fonctionne. On aurait pu par exemple tester le reste de la division euclidienne de n par deux.

### 2.5) Récursivité multiple

Il existe un autre cas particulier où la fonction s'appelle plusieurs fois. On parle alors de récursivité multiple. C'est le cas par exemple dans le cas du calcul des coefficients binomiaux. On peut donner une rappel mathématique de ces coefficients binomiaux qui sont caractérisés par la définition suivante pour toute valeur entière de $n$ et $k$ telles que $0 \leq k \leq n$ :

$$\binom{n}{k} = \begin{cases} 1 & si \quad k = 0 \\ 1 & si \quad k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & sinon \end{cases}$$

Alors on peut donner la fonction algorithmique du programme 4.

```
Fonction CoeffBinomial(n,k):
    Si k==0 OU k==n:
        Retourner 1
    Sinon :
        Retourner  CoeffBinomial(n-1,k-1)+ CoeffBinomial(n-1,k)
```

Listing 4 – Coefficient Binomiaux

## 3] Exercices

**Exercice 5**

La suite de Fibonacci est une suite d'entiers. Elle doit son nom à Leonardo Fibonacci, dit Leonardo Pisano, un mathématicien italien du XIII$^{e}$

En mathématiques, la suite de Fibonacci est une suite d'entiers dans laquelle chaque terme est la somme des deux termes qui le précèdent.

Notée $F_n$, elle est définie par $F_0 = 0$, $F_1 = 1$, et $F_n = F_{n-1} + F_{n-2}$ pour $n > 2$

Les termes de cette suite sont appelés nombres de Fibonacci et forment la suite A000045 de l'OEIS :

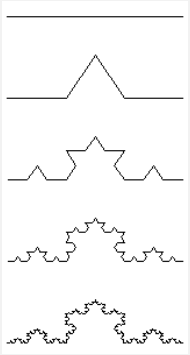
| n | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| F(n) | 0 | 1 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | 34 | 55 |

Cette suite est liée au nombre d'or, $\phi$ (phi) : ce nombre intervient dans l'expression du terme général de la suite. Inversement, la suite de Fibonacci intervient dans l'écriture des réduites de l'expression de $\phi$ en fraction continue : les quotients de deux termes consécutifs de la suite de Fibonacci sont les meilleures approximations du nombre d'or.

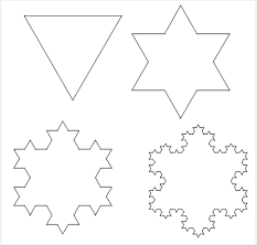
Écrire la fonction qui donne le n$^{ième}$ terme de la suite de Fibonacci de manière récursive et pour les plus rapides de manière itérative.

## Exercice 6

L'objectif de cet exercice est de réaliser la fractale de Von Koch à l'aide du module python *turtle*. **Il faudra lire attentivement la documentation**. Vous ferez cet exercice dans le cadre du Projet 1.
Une fractale est une sorte de courbe mathématique un peu complexe riche en détail, et qui possède une propriété intéressante visuellement : lorsque l'on regarde des détails de petite taille, on retrouve des formes correspondant aux détails de plus grande taille (auto-similarité). Cela nous rappelle étrangement la récursivité ! La première courbe à tracer a été imaginée en 1904 par le mathématicien suédois Niels Fabian Helge Von Koch.
Le principe est simple : on divise un segment initial en trois morceaux, et on construit un triangle équilatéral sans base au-dessus du morceau central. On réitère le processus $n$ fois, $n$ est appelé l'ordre. Dans la figure suivante, on voit les ordres 0,1,2 et 3 de cette fractale.



1. Proposer une fonction récursive permettant de dessiner la fractale de Von Koch en lui donnant comme paramètre l'ordre et la longueur du segment initial.
2. Le flocon de 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.



Proposer une fonction permettant de faire le flocon de Koch complet à partir de la fonction réalisée précédente.