# Récursivité

La récursivité est un principe assez général, qui ne se limite pas à la programmation et l'algorithmique. Ce principe décrit tout objet ou concept qui fait référence à lui même pour se définir.
En informatique, une fonction ou plus généralement un algorithme qui contient un ou des appel(s) à lui-même est dit récursif.

## 1 Le problème de la somme des $n$ premiers entiers

La somme des $n$ premiers entiers est la somme :

$$0 + 1 + 2 + \cdots + n \tag{1}$$

Une solution pour calculer cette somme consiste à utiliser une boucle **for** pour parcourir tous les entiers `i` entre `0` et `n`, en s'aidant d'une variable locale `s` pour accumuler la somme des entiers de `0` à `i`. On obtient par exemple le programme Python suivant :

```python
def somme(n):
    s = 0
    for i in range(n+1):
        s += i
    return s
```

Si la fonction `somme(n)` ci-dessus calcul bien la somme des $n$ premiers entiers, on peut remarquer que ce code Python n'est pas directement lié à la formule (1). En effet, il n'y a rien dans cette formule qui puisse laisser deviner qu'une variable `s` est nécessaire pour calculer cette somme.
Une autre manière d'aborder ce problème est de définir une fonction mathématique $somme(n)$ de la manière suivante :

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

En effet, pour $n = 0$, $somme(0)$ est égale à 0 et pour $n$ entier strictement positif, $somme(n)$ est égale à $n+somme(n-1)$. Il s'agit d'une définition **récursive**, la définition de $somme(n)$ fait appel à $somme(n-1)$. Cette définition est directement programmable en Python, comme le montre le code ci-dessous :

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

La fonction ainsi obtenue est une **fonction récursive**, l'appel à `somme(n-1)` dans le corps de la fonction est un **appel récurssif**.
Par exemple, l'évaluation de l'appel à `somme(3)` peut se représenter à l'aide de l'**arbre d'appels** suivant :

```
somme(3) = return 3 + somme(2)
                          |
                     return 2 + somme(1)
                                    |
                               return 1 + somme(0)
                                              |
                                         return 0
```

Pour calculer la valeur renvoyé par `somme(3)`, il faut d'abord appeler `somme(2)` qui fait appel à `somme(1)` qui à son tour nécessite un appel à `somme(0)`.
Le dernier appel se termine directement en renvoyant la valeur 0. Le calcul de `somme(3)` se fait «à rebourd». L'abre d'appels à alors la forme suivante :

```
somme(3) = return 3 + somme(2)
                         |
                   return 2 + somme(1)
                                 |
                           return 1 + 0
```

L'appel à `somme(1)` peut alors se terminer et renvoyer 1.

```
somme(3) = return 3 + somme(2)
                         |
                   return 2 + 1
```

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

```
somme(3) = return 3 + 3
```

On obtient bien au final la valeur `6` attendue.

# 2 Formulations récursives

## 2.1 Cas de base et cas récursifs

Une formulation d'une fonction récursive est toujours constituée de un ou plusieurs **cas de base (ou conditions d'arrêt)** et de un ou plusieurs **cas récursifs**.
Les cas récursifs sont ceux qui renvoient à la fonction entrain entrain d'être définie et les cas de base sont ceux qui donne un résultat.
Dans notre exemple, il y a un cas de base :

$$somme(0) = 0$$

et un cas récursif :

$$somme(n) = somme(n-1) + n$$

## 2.2 Cas multiples

Il est également possible de définir une fonction avec plusieurs cas récursifs.
Prenons comme exemple la fonction $puissance(x, n)$ qui calcul $x^n$ :

$$x^n = \underbrace{x \times \cdots \times x}_{n \text{ fois}}$$

avec pour convention que $x^0 = 1$.
En remarquant que $x^n = x \times x^{n-1}$, on obtient la formulation récursive suivante :

$$puissance(x, n) = \begin{cases} 1 & \text{si } n = 0 \\ x \times puissance(x, n-1) & \text{si } n > 0 \end{cases}$$

On peut éviter la multiplication (inutile) $x \times 1$ de la définition précédente en ajoutant un cas de base : $puissance(x, 1) = x$. On obtient ainsi la définition suivante avec deux cas de base :

$$puissance(x, n) = \begin{cases} 1 & \text{si } n = 0 \\ x & \text{si } n = 1 \\ x \times puissance(x, n-1) & \text{si } n > 0 \end{cases}$$

Il est également possible de définir la fonction $puissance(x, n)$ avec plusieurs cas récursifs. En effet si $n$ est pair, $x^n = \left(x^{\frac{n}{2}}\right)^2$ et si $n$ est impair $x^n = x \times \left(x^{\frac{n-1}{2}}\right)^2$.
En supposant que l'on dispose de la fonction $carre(x) = x \times x$, on obtient la définition récursive suivante :

$$puissance(x, n) = \begin{cases} 1 & \text{si } n = 0 \\ carre(puissance(x, n/2) & \text{si } n \text{ est pair} \\ x \times carre(puissance(x, (n-1)/2) & \text{si } n \text{ est impair} \end{cases}$$

## 2.3 Récursion multiple

Dans l'expression d'une fonction récursive, un même cas récursif peut faire plusieurs appels récursifs. Prenons pour exemple la suite de Fibonacci qui doit son nom à Leonardo Fibonacci. Dans un problème récréatif posé dans l'ouvrage *Liber abaci* publié en 1202, il y décrit la croissance d'une population de lapins.

$$fibonacci(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ fibonacci(n-2) + fibonacci(n-1) & \text{si } n > 1 \end{cases}$$

Voici par exemple les premières valeurs de cette fonction :

$$\begin{aligned} &fibonacci(0) = 0 \\ &fibonacci(1) = 1 \\ &fibonacci(2) = fibonacci(0) + fibonacci(1) && = 0 + 1 = 1 \\ &fibonacci(3) = fibonacci(1) + fibonacci(2) && = 1 + 1 = 2 \\ &fibonacci(4) = fibonacci(2) + fibonacci(3) && = 1 + 2 = 3 \\ &fibonacci(5) = fibonacci(3) + fibonacci(4) && = 2 + 3 = 5 \end{aligned}$$

## 2.4 Récursion mutuelle

Il est également possible de définir plusieurs fonctions récursives en *même temps*, quand ces fonctions font référence les unes aux autres. On parle alors de définitions **récursives mutuelles**.
Prenons par exemple les deux fonctions ci-dessous permettant de tester si un nombre est pair ou impair.

$$pair(n) = \begin{cases} vrai & \text{si } n = 0 \\ impair(n-1) & \text{si } n > 0 \end{cases}$$

$$impair(n) = \begin{cases} faux & \text{si } n = 0 \\ pair(n-1) & \text{si } n > 0 \end{cases}$$

## 2.5 Définition récursive bien formulée

Il y a quelques règles à respecter lorsque l'on écrit une fonction récursive :
- la récursion doit s'arrêter, c'est-à-dire que l'on fini toujours par arriver à un cas de base,
- les valeurs utilisées pour appeler la fonction sont toujours dans le domaine de la fonction,
- il y a bien une définition pour toutes les valeurs du domaine.

Pour la fonction $f(n)$ ci-dessous, la condition d'arrêt n'est jamais vérifier :

$$f(n) = \begin{cases} 1 & \text{si } n = 0 \\ n + f(n+1) & \text{si } n > 0 \end{cases}$$

La fonction $g(n)$ suivante n'est pas définie pour tous les entiers :

$$g(n) = \begin{cases} 1 & \text{si } n = 0 \\ n + g(n-2) & \text{si } n > 0 \end{cases}$$

En effet, $g(1)$ par exemple ne peut être calculé.

# 3 Programmer avec des fonctions récursives

## 3.1 type de données

La fonction `somme(n)` ne se comporte pas exactement la fonction mathématiques $somme(n)$.

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

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

La fonction mathématiques est uniquement définie pour les entiers naturels alors que la fonction `somme(n)` peut être appelée avec une valeur quelconque.
Une première solution consiste à modifier le premier test :

```
def somme(n):
    if n <= 0:
        return 0
    else:
        return n + somme(n-1)
```

Cette solution assure la terminaison de la fonction mais modifie sa spécification.

Une autre solution est de restreindre les appels à la fonction `somme(n)`.

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

Cette solution est correcte mais a le défaut de faire le test `n >= 0` à chaque appel de `somme(n)` alors que `n` sera positif une fois le premier test réalisé.
Une solution pour éviter ces tests inutiles est de définir une deuxième fonction :

```
def somme(n):
    assert n >= 0
    return somme_bis(n)

def somme_bis(n):
    if n == 0:
        return 0
    else:
        return n + somme_bis(n-1)
```

## 3.2 Modèle d'exécution