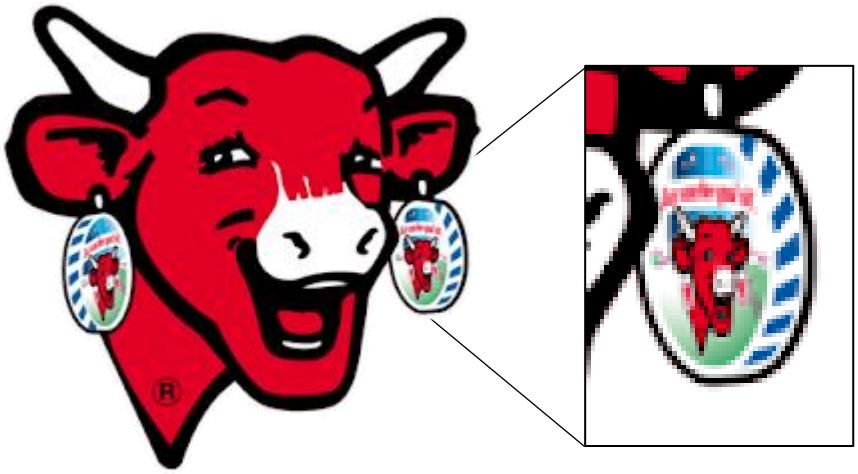
# Récursivité en programmation

## Qu'est-ce que c'est ?

*« Un algorithme récursif est un algorithme qui résout un problème en calculant des solutions d'instances plus petites du même problème.»*

Source : Wikipédia

Plus vulgairement, **un algorithme récursif s'appel lui-même dans son déroulement**, généralement en s'incrémentant ou décrémentant.



Visuellement, on retrouve un exemple de récursivité (que l'on qualifiera de « mise en abyme » pour ce cas-ci) dans le logo de *La Vache qui rit*.

Dans une fonction récursive, il est nécessaire d'effectuer deux vérifications : **Est-ce que l'algorithme se termine ?** et **Si l'algorithme se termine, est-ce que l'algorithme fait ce qu'il est supposé faire ?** .

## Comparaison entre itératif et récursif : la fonction factorielle

*« En mathématiques, la factorielle d'un entier naturel n est le produit des nombres entiers strictement positifs inférieurs ou égaux à n. »*

Source : Wikipédia

Ce qui peut se traduire par **n! = 1 × 2 × 3 × … × (n - 1) × n**, ou plus exactement **n! = (n - 1)! × n**.

$0! = 1$

$1! = 1$

$2! = 1 \times 2 = 2$

$3! = 1 \times 2 \times 3 = 6$

$4! = 1 \times 2 \times 3 \times 4 = 24$

...

12! = 479 001 600

13! = 6 227 020 800

**Algorithme itératif** de la fonction factorielle

```
// entrée : un nombre entier n
// but : calculer la valeur factorielle d'un entier n
// sortie : un nombre entier resultat

DEBUT
    resultat ← n
    si (n < 0)
        alors renvoyer 0
    sinon si (n = 1 OU n = 0)
        alors renvoyer 1
    sinon
        tant que n > 1
            resultat ← resultat x (n - 1)
            n ← n – 1
        fintantque
    finsi
    renvoyer resultat
FIN
```

**Algorithme récursif** de la fonction factorielle

```
// entrée : un nombre entier n
// but : calculer la valeur factorielle d'un entier n
// sortie : un nombre entier resultat

DEBUT
    si (n < 0)
        alors renvoyer 0
    sinon si (n = 1 OU n = 0)
        alors renvoyer 1
    sinon
        resultat ← resultat x factorielle(n - 1)
    finsi
    renvoyer resultat
FIN
```

Pour cet algorithme, on peut aisément vérifier que la récursivité est conforme :

**Est-ce que l'algorithme se termine ?**

Oui, l'algorithme se répète uniquement si la valeur de ***nb*** est strictement supérieure à 1, et il décrémente la valeur de ***nb***. Donc elle finira nécessairement par être égale ou inférieur à 1, arrêtant ainsi le déroulement.

En revanche, si l'on avait `resultat ← resultat x factorielle(n + 1)` en ligne 10, le programme ne pourrait pas se terminer.

**S'il se termine, est-ce que l'algorithme fait ce qu'il est supposé faire ?**
Ici, on doit effectuer une vérification mathématique. Dans notre cas, le cœur de l'algorithme repose sur un simple produit, *resultat* ← *resultat* x *factorielle*(*n* – 1), qui retourne dans toutes les situations la valeur attendue.

Ce programme est donc conforme au principe de la récursive.

## Du point de vue programmation

*« La programmation récursive est une technique de programmation qui remplace les instructions de boucle (while, for, etc.) par des appels de fonction. »*

Source : Wikipédia

Tout algorithme peut s'écrire en itératif ou en récursif. Par contre, la pile système passe en général plus de valeurs que la pile qu'on aurait implémentée. Elle prend donc un peu plus de temps, et surtout nettement plus de place. De plus, la procédure récursive n'est jamais plus rapide que la procédure itérative correspondante : fondamentalement, le temps de calcul est surtout déterminé par l'algorithme.

Dans certains cas, on peut écrire un code très court et bien lisible en utilisant une procédure récursive. C'est commode pour prototyper un programme (faire un exemplaire d'essai, de mise au point, sur lequel on ne veut pas passer trop de temps) : on a une pile gratuitement, sans avoir besoin de l'écrire.
Cela n'a d'intérêt que pour les opérations qui utilisent une pile, c'est à dire en gros les parcours d'arbres en profondeur.

Il est très rare qu'on utilise une procédure récursive dans la version définitive d'un programme professionnel.

### Exemple avec la fonction factorielle

Programme **itératif de la fonction**

```c
int ft_iterative_factorial(int nb)
{
    int result;
    result = nb;
    if (nb < 0 || nb > 12)
        return (0);
    if (nb == 1 || nb == 0)
        return (1);
    while (nb > 1)
    {
         result = result * (nb - 1);
         nb--;
    }
    return (result);
}
```

La valeur maximum tolérée par le programme étant 2 147 483 647, on ne pourra pas calculer les termes au-delà de 12! . C'est pour cela que l'on doit limiter la valeur de ***nb*** à 12 en ligne 5

Programme **récursif de la fonction**

```
1  int ft_recursive_factorial(int nb)
2  {
3      int result;
4      if (nb < 0 || nb > 12)
5          return (0);
6      if (nb <= 1)
7          return (1);
8      result = (nb * ft_recursive_factorial(nb - 1));
9      return (result);
10 }
```

Exécution de la fonction pour ***nb = 4***

Avec ***nb = 4***, on entre dans une nouvelle itération de la fonction pour ***nb = 3***

```
1  ft_recursive_factorial(4)
2  {
…      …
10     result = (4 * ft_recursive_factorial(4 - 1));
11     return (result);
12 }
```

Avec ***nb = 3***, on entre dans une nouvelle itération de la fonction pour ***nb = 2***

```
1  ft_recursive_factorial(3)
2  {
…      …
10     result = (3 * ft_recursive_factorial(3 - 1));
11     return (result);
12 }
```

Avec ***nb = 2***, on entre dans une nouvelle itération de la fonction pour ***nb = 1***

```
1  ft_recursive_factorial(2)
2  {
…      …
10     result = (2 * ft_recursive_factorial(2 - 1));
11     return (result);
12 }
```

Avec ***nb = 1***, on n'entre pas dans une nouvelle itération de la fonction et on récupère la valeur 1.

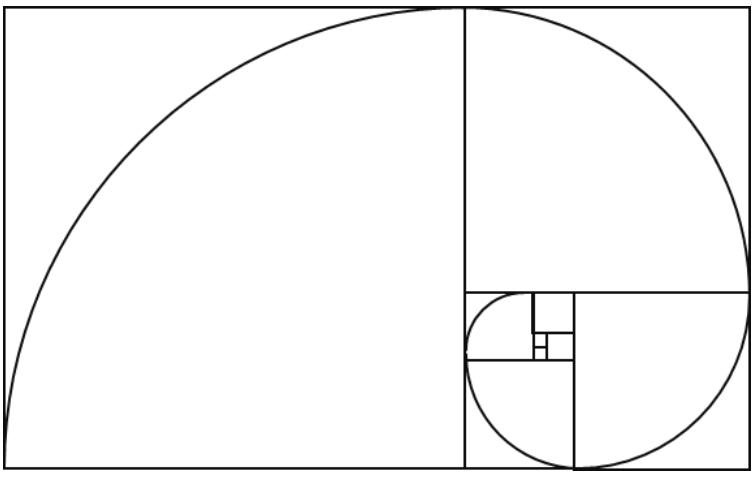
```
1  ft_recursive_factorial(1)
2  {
…      …
6      if (1 <= 1)
7          return (1);
…      …
12 }
```

Et avec cette valeur, on peut maintenant « rebrousser chemin » dans les itérations jusqu'à *nb = 4*.

```
1   ft_recursive_factorial(1)
2   {
…       …
6       if (1 <= 1)
7           return (1);
…       …
12  }
13
14  ft_recursive_factorial(2)
15  {
…       …
23      result = (2 * 1); //1 pour ft_recursive_factorial(2 - 1)
24      return (result); //result = 2
25  }
26
27  ft_recursive_factorial(3)
28  {
…       …
36      result = (3 * 2); //2 pour ft_recursive_factorial(3 - 1)
37      return (result); //result = 6
38  }
39
40  ft_recursive_factorial(4)
41  {
…       …
49      result = (4 * 6); //6 pour ft_recursive_factorial(4 - 1)
50      return (result); //result = 24
51  }
```

## Exemple de Fibonacci, optimisation récursive

Pour comprendre le concept de récursivité, nous allons étudier un autre cas précis : La suite de Fibonacci. Vous en avez sûrement déjà entendu parler, c'est ce truc qui fait un escargot là :



*« En mathématiques, la suite de Fibonacci est une suite d'entiers dans laquelle chaque terme est la somme des deux termes qui le précèdent. Elle commence par les termes 0 et 1. »*

Source : Wikipédia

Donc, le rapport avec la récursivité, c'est que pour avoir un nombre de cette fonction, on va devoir regarder les **autres résultats de cette fonction** !

| $F_0$ | $F_1$ | $F_2$ | $F_3$ | $F_4$ | $F_5$ | $F_6$ | $F_7$ | $F_8$ | $F_9$ | $F_{10}$ | $F_{11}$ | $F_{12}$ | ⋯ | $\boldsymbol{F_n}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | 34 | 55 | 89 | 144 | ⋯ | $\boldsymbol{F_{n-1}+F_{n-2}}$ |

**Algorithme itératif** de la suite de Fibonacci

```
// Entrée : un nombre entier n
// But : calculer le terme de rang n de la suite de Fibonacci
// Sortie : un nombre entier F
DEBUT
    si n < 0
        F ← -1
    sinon si n = 0
        F ← 0
    sinon si n = 1
        F ← 1
    sinon
        a ← 1
        b ← 1
        pour i de 2 à n faire
            c ← a + b
            a = b
            b = c
        finpour
        F ← c
    finsi
    renvoyer F
FIN
```

**Algorithme récursif** de la suite de Fibonacci

```
// Entrée : un nombre entier n
// But : calculer le terme de rang n de la suite de Fibonacci
// Sortie : un nombre entier F
DEBUT
    si n < 0
        F ← -1
    sinon si n = 0
        F ← 0
    sinon si n = 1
        F ← 1
    sinon
        F ← fibo(n - 1) + fibo(n - 2)
    finsi
    renvoyer F
FIN
```

Programme en C de l'algorithme récursif

```c
int ft_fibonacci(int index)
{
    int result;
    if (index < 0)
        return (-1);
    else if (index == 0)
        return (0);
    else if (index == 1)
        return (1);
    else if (index > 1)
    {
        result = ft_fibonacci(index - 1) + ft_fibonacci(index - 2);
        return (result);
    }
    return (0);
}
```

## Exemple de Koch, risques et avantages

Le flocon de koch est, à la base, un simple triangle. Mais il se transformant étape par étape en une forme géométrique complexe en "cassant" chacun de ses segments en 4, rajoutant 3 sommets à la forme.
Dans sa définition, la longueur du flocon de koch est infinie (à chaque étape, la longueur augmente de 4/3), mais l'aire du flocon reste intacte, elle ne change pas.

Nous constatons donc que cette fractale relève d'un paradoxe exceptionnel : son périmètre tend vers l'infini alors que son aire est finie.

Exemple de **programme itératif du flocon de Koch**

[screen programme itératif]

Source : Fefef, https://openclassrooms.com/forum/sujet/csdl-fractale-flocon-de-von-koch-55438

Ici, on utilise une liste chaînée de segments (donc de couples de points) qu'on remplace à chaque itération par les 4 autres segments calculés. Ce qui est très long, mais on peut avoir une certaine précision et contrôle de la forme que l'on veut avoir.

Exemple de **programme récursif du flocon de Koch**

[screen programme récursif]

Source : Remram44, https://openclassrooms.com/forum/sujet/csdl-fractale-flocon-de-von-koch-55438

Il est bien plus court (moins de 100 lignes) et plus clair que sa version itérative. De se fait il est tout de suite plus facile à modifier.

De plus, dans sa version itérative, ce programme est bien plus léger dans son fonctionne. En effet, un programme récursif de cette ampleur va logiquement, à chaque rappel, peser un peu plus sur la mémoire de la machine.

Cela montre que la récursivité permet de simplifier grandement un programme mais prix d’une certaine lourdeur dans son déroulement en plus du risque de boucle infinie.