# Récursivité

Des fonctions ou procédures qui s'appellent elles-mêmes

4 octobre 2017

## Table des matières



# 1 Exemples introductifs

## 1.1 Fonction factorielle

La fonction factorielle fac peut être définie ainsi :
— si $n \leq 1$ alors $\text{fac}(n) = 1$ ;
— sinon $\text{fac}(n) = n \cdot \text{fac}(n-1)$ .

Cette définition est dite **récursive** car fac fait appel à elle-même. Elle est valide car :
— il y a un cas de base qui ne génère pas d'appel récursif ;
— il n'y aura qu'un *nombre fini d'appels récursifs* avant de tomber à coup sûr sur le cas de base ;
— cette assurance est fournie par la variable $n$, qui est décrémentée de 1 à chaque appel, et donc finit par valoir 1.

Implémentation Python de la factorielle récursive :

```
def fac(n):
    if n<=1: return 1
    else: return n*fac(n-1)
```

ce qui est la traduction exacte de la formule de récurrence. Implémentation Python de la factorielle non récursive :

```
def fac2(n):
    p=1
    for i in range(2,n+1): p*=i
    return p
```

Cette version est dite **itérative** car utilise une boucle, contrairement à la précédente.

## 1.2 Suite de Fibonacci

C'est la fonction fib définie par :
— $\text{fib}(0) = 0$ ;
— $\text{fib}(1) = 1$ ;
— $\forall n \geq 2$ , $\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$ ;
C'est une définition récursive : chaque calcul de fib génère 2 appels de fib . Ce qui assure la validité de cette définition est :
— la présence de 2 cas de base ne générant pas d'appel récursif ;
— la certitude qu'il n'y aura qu'un nombre fini d'appels récursifs avant de tomber sur un cas de base ;
— certitude qui est fournie par la variable $n$, qui décroît strictement à chaque appel, et donc finit par valoir 0 ou 1.

Implémentation Python de la suite de Fibonacci récursive : traduction (presque) mot à mot !

```
def fib(n):
    if n<=1: return n
    else: return fib(n-1)+fib(n-2)
```

*Exercice*
Écrire une version itérative de la suite de Fibonacci. Est-ce plus difficile ? Est-ce plus efficace ?

Solution :

```
def fib2(n):
    if n<=1: return n
    else:
        a,b = 0,1
        for i in range(2,n+1):
            a,b = b,a+b
        return b
```

— c'est bien plus difficile à concevoir ;
— en revanche c'est plus rapide à l'exécution, pour des raisons qu'on verra.

# 2 La récursivité en général

## 2.1 Algorithmes récursifs

Un algorithme est dit **récursif** quand sa mise en oeuvre utilise ce même algorithme. Pour être valide, cet algorithme doit impérativement vérifier les 2 **contraintes de terminaison** :

— existence d'un ou plusieurs cas de base où l'algorithme est directement effectif ;
— assurance qu'il n'y aura qu'un nombre fini d'appels récursifs avant de déboucher sur un cas de base.

Cette assurance est fournie par une **variable de contrôle** : entier naturel qui doit :

— décroître strictement à chaque appel récursif ;
— valoir 0 (ou 1) pour les cas de base.

*Exemple*
Pour lister les anagrammes d'un mot, on peut :

1. exclure la dernière lettre ;
2. lister tous les anagrammes du mot restant ;
3. dans chacun de ces mots, insérer la dernière lettre à toutes les positions possibles.

Ici :

— l'utilisation récursive a lieu à la deuxième étape ;
— les cas de base sont les mots d'une lettre ;
— la variable de contrôle est la longueur du mot.

## 2.2 Fonctions et procédures récursives

Ce sont les implémentations d'algorithmes récursifs. Elles se caractérisent par le fait que le nom de la fonction (ou procédure) apparait dans sa déclaration.

***Pour déclarer une fonction récursive, vous devrez :***

— vous assurer qu'il y a une variable de contrôle ;
— commencer par traiter les cas de base.

## 2.3 Avantages et inconvénients

L'avantage principal de la récursivité est la simplicité de programmation. Pour écrire un programme récursif, il suffit de :

— trouver comment réduire le problème de taille $n$ à un ou plusieurs problèmes de taille plus petite ;
— traduire simplement la relation trouvée ;
— vérifier la terminaison de l'algorithme.

Les inconvénients sont :

— la multiplication des mémoires allouées au stockage des résultats en attente, qui peut devenir rédhibitoire ;
— le nombre de calculs effectués, souvent bien plus grand qu'en programmation itérative.

# 3 Comment fonctionne la récursivité ?

## 3.1 La pile d'appel

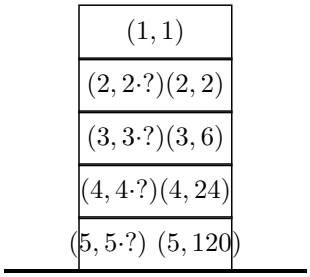
Le principe est d'utiliser une pile, dite **pile d'appel**, sur laquelle on empile les calculs en attente.

- Tout appel à une fonction crée d'abord 2 mémoires :
  - 1 mémoire pour stocker l'argument, noté ici x ;
  - 1 mémoire pour stocker le résultat, noté ici y.
- Si le calcul de y n'est pas encore possible, ce qui est le cas s'il y a un appel récursif, le couple (x,y) est empilé. La pile augmente tant qu'il y a des calculs en attente.
- Dès qu'un calcul de y est possible, (x,y) est dépilé ; on récupère ainsi y, et on utilise généralement sa valeur pour faire le calcul qui est en attente au sommet de la pile.
- Quand la pile est vide, le dernier y qui a été dépilé fournit le résultat.

## 3.2 Exemple de la factorielle

```
def fac(n):
    if n<=1: return 1
    else: return n*fac(n-1)
```

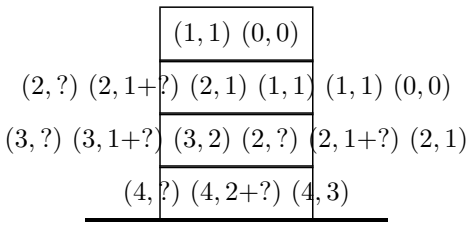
On va visualiser l'évolution de la pile d'appel pour `fac(5)` :




## 3.3 Exemple de la suite de Fibonacci

```
def fib(n):
    if n<=1: return n
    else: return fib(n-1)+fib(n-2)
```

On va visualiser l'évolution de la pile d'appel pour `fib(4)` :



# 4 Complexité d'un algorithme récursif

## 4.1 Définitions

- La **complexité temporelle** d'un algorithme est l'*ordre de grandeur* du nombre d'opérations élémentaires qu'il va nécessiter pour résoudre un problème de taille $n$. Ce nombre est à peu près proportionnel au temps effectif de calcul.
- La **complexité spatiale** d'un algorithme est l'*ordre de grandeur* du nombre de mémoires qu'il va utiliser pour résoudre un problème de taille $n$.
- Dans les deux cas on distingue la complexité :
  - en moyenne ;
  - dans le meilleur des cas ;
  - dans le pire des cas.

La plus utile est la première, la plus discriminante est la dernière.

## 4.2 Exemple de la factorielle

```
def fac(n):
    if n<=1: return 1
    else: return n*fac(n-1)
```

```
def fac2(n):
    p=1
    for i in range(2,n+1):
        p*=i
    return p
```

- Pour calculer $n!$ par la méthode itérative, il faut :
  - $n-1$ multiplications : complexité temporelle $O(n)$ ;
  - 4 mémoires : complexité spatiale $O(1)$.
- Pour calculer $n!$ par la méthode récursive, il faut :
  - $n$ tests, $n-1$ multiplications : complexité temporelle $O(n)$ ;
  - $2n$ mémoires (2 à chaque appel) : complexité spatiale $O(n)$.

Les deux méthodes s'exécutent en à peu près la même durée, mais la deuxième est plus gourmande en mémoire.

## 4.3 Exemple de la suite de Fibonacci

```
def fib2(n):
    if n<=1: return n
    else:
        a,b = 0,1
        for i in range(2,n+1): a,b = b,a+b
        return b
```

- Nombre d'opérations : $1(+2) + (n-1)(1(+2))$ soit une complexité temporelle $O(n)$ : raisonnable.
- Nombre de mémoires : 5 soit une complexité spatiale $O(1)$ : excellente.

```
def fib(n):
    if n<=1: return n
    else: return fib(n-1)+fib(n-2)
```

Je note $M(n)$ le nombre de mémoires utilisée par `fib(n)` :

$$\forall n \geq 2\ ,\ M(n) = 2 + M(n-1)$$

car contrairement au temps, *la mémoire est recyclable* : on peut réutiliser celles du calcul de `fib(n-1)` pour `fib(n-2)` . On a directement :

$$M(n) = M(0) + 2n$$

donc $M(n) = O(n)$ : la complexité spatiale est raisonnable, mais moins bonne qu'en itératif.

Je note $C(n)$ le nombre de calculs faits pour `fib(n)` :

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

On pose $U(n) = C(n) + 2$ car -2 est le point fixe de la récurrence :

$$\forall n \geq 2\ ,\ U(n) = U(n-1) + U(n-2)$$

C'est une récurrence classique qu'on résout par la méthode de l'équation caractéristique ; on trouve :

$$U(n) = \alpha r^n + \beta s^n \ \text{ où } \ r = \frac{1+\sqrt{5}}{2} \simeq 1.6\ ,\ s = \frac{1-\sqrt{5}}{2} \simeq -0.6$$

On en déduit $U(n) = O\left(r^n\right)$ puis $C(n) = O\left(r^n\right)$ . La complexité temporelle est *très mauvaise car exponentielle.*

## 4.4 Complexité exponentielle

Pour calculer `fib(100)`, la machine fera de l'ordre de $r^{100}$ opérations. Or :

$$\log_{10}(r^{100}) = 100 \log_{10}(r) \simeq 20$$

ce qui signifie qu'il y aura environ $10^{20}$ opérations. À raison de $10^9$ opérations par seconde, le calcul prendra de l'ordre de $10^{11}$ secondes, soit environ 3000 ans ! D'une manière générale, les algorithmes qui ont une complexité temporelle du type $O(q^n)$ avec $q > 1$ ne peuvent pas être exécutés en temps acceptable pour $n$ grand, contrairement à ceux en $O(n^p)$ qui peuvent l'être (surtout si $p$ n'est pas trop grand). Ces derniers sont dits de **type P**.