> Version pdf

La récursivité (Bac 🎯)

Définition d'une fonction récursive

> **"Point de cours 1"**

Un **algorithme récursif** est un algorithme qui résout un problème en se ramenant à la résolution de un ou plusieurs sous-problèmes similaires mais plus petits jusqu'à la résolution d'un cas de base dont la réponse est directe.

On distingue deux parties dans un algorithme récursif :

- la *réduction* consiste à réduire la résolution du problème initial de taille n à celle de un ou plusieurs sous-problèmes de même type et de taille inférieure ($n-1$ ou $n/2$ etc ...) : l'algorithme s'appelle alors lui-même sur ces sous-problèmes : on parle d'*appels récursifs*.
- la résolution directe pour un *cas de base*

Un algorithme récursif est implémenté sous la forme d'une **fonction récursive**.

> 💡**Une *fonction récursive* est une fonction qui s'appelle elle-même.**

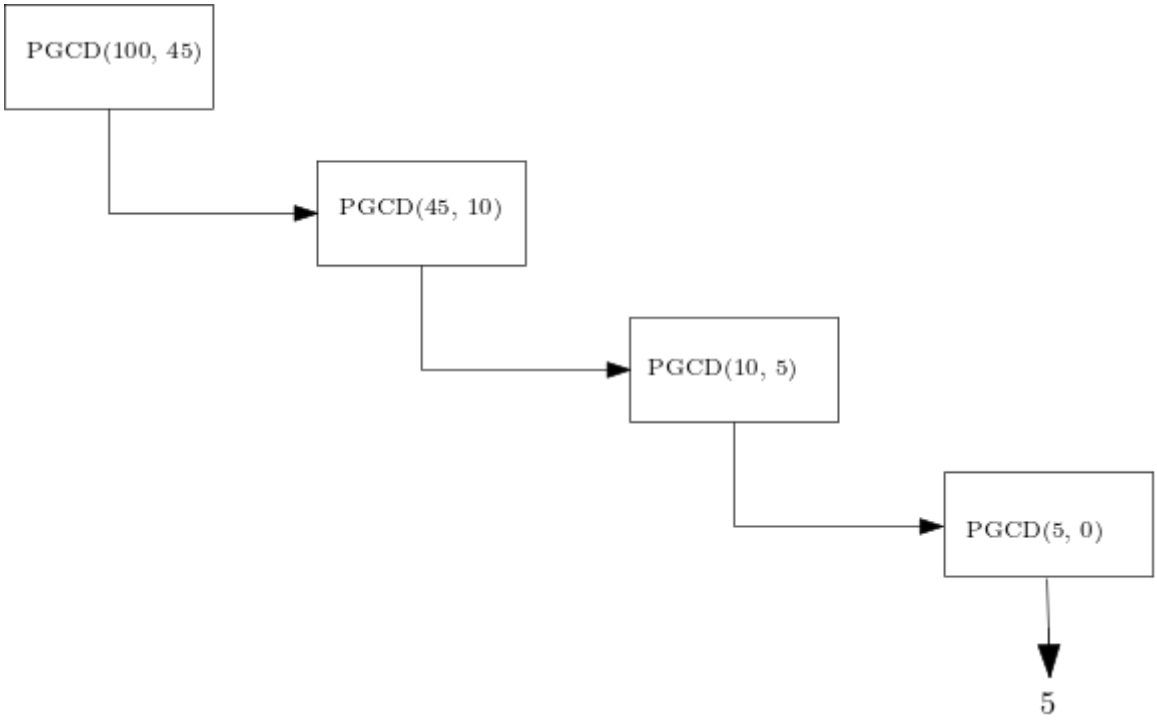
Le code d'une **fonction récursive** suit donc en général un schéma avec un `if ... else` :

```
def fonction_rec(n):
    # n entier naturel pour simplifier
    if n in cas_de_base:  # cas de base
        return solution_directe
    else:
        return fonction_rec(reduction(n)) #avec reduction(n) < n
```

Trace d'une fonction récursive

> **"Méthode 1 : trace d'une fonction récursive"**

On peut représenter l'évaluation d'une fonction récursive par un **arbre d'appels**. Par exemple pour une fonction récursive de calcul du PGCD de deux entiers, on peut représenter ainsi les appels récursifs dans l'évaluation de `PGCD(100, 45)` lors de la *phase de descente* qui se termine par un **cas de base** :



La *phase de remontée* depuis un cas de base s'effectue en remplaçant *à rebours* chaque appel récursif par sa valeur :

Correction et terminaison

"Méthode 2 : preuve de correction par récurrence"

La correction d'une fonction récursive se démontre à l'aide d'un *raisonnement* par récurrence comme en Mathématiques :

- *Initialisation* : on démontre que la propriété est vraie pour le ou les cas de base (sans en oublier), ici pour $n = 0$.
- *Hérédité* : on démontre que pour tout entier naturel n, si on suppose que $P(n)$ est vraie alors $P(n+1)$ est encore vraie.

"Récursivité et terminaison"

Une fonction récursive ne se termine pas si les étapes de réduction ne font pas converger les appels récursifs vers un cas de base.

Dans l'exercice 1, on a vu que celà peut se produire :

- Si la fonction ne propose pas de cas de base :

```
def factorielle(n):
    """Fausse"""
    # réduction mais pas de cas de base
    return n * factorielle(n-1)
```

- Si la fonction implémentée est appelée sur une valeur en dehors du domaine de définition de l'algorithme récursif :

```
def factorielle(n):
    """Renvoie n! = 1 *2 * ... (n-1) * n pour tout entier n >= 0"""
    # cas de base
    if n == 0:
        return 1
    # réduction
    return n * factorielle(n-1)

# Appel sur un nombre négatif => descente infinie
factorielle(-1)
```

On peut prévenir ces problèmes en vérifiant des préconditions (**programmation défensive**).

```python
def factorielle(n):
    """Renvoie n! = 1 *2 * ... (n-1) * n pour tout entier n >= 0"""
    assert isinstance(n, int) and n >= 0
    # cas de base
    if n == 0:
        return 1
    # réduction
    return n * factorielle(n-1)

# Appel sur un nombre négatif => descente infinie
factorielle(-1)
```

- Si dans la fonction implémentée, la réduction ne fait pas converger les appels récursifs vers un cas de base. Par exemple dans le cas ci-dessous l'appel `factorielle(1)` ne se termine pas car `factorielle(1)` appelle `factorielle(-1)` qui appelle `factorielle(-3)` etc ... Donc le cas de base 0 a été sauté et ne sera jamais atteint.

```python
def factorielle(n):
    """Fausse"""
    # cas de base
    if n == 0:
        return 1
    # réduction fausse
    return n * factorielle(n - 2)
```

> 💡 *Comme pour une boucle, on peut démontrer que fonction récursive se termine, à l'aide d'un **variant**, une quantité entière positive dont on démontre qu'elle décroît lors de chaque appel récursif jusqu'à ce que le cas de base soit atteint.*

Avantages et inconvénients de la récursivité

"Point de cours 3"

Dans le problème du dénombrement des murs de longueur n construits avec des briques de longueur 2 ou 3, on a vu un exemple où la récursivité permet d'exprimer simplement et lisiblement une solution à un problème en ramenant sa résolution à la résolution de problèmes similaires mais plus petits. Cette approche sera déclinée dans les méthodes de résolution :

- *Diviser pour régner* où les sous-problèmes sont indépendants
- par *Programmation dynamique* où les sous-problèmes peuvent se chevaucher, comme dans le problème du mur.

"Exemple 6"

La suite de Fibonacci, est définie récursivement : chaque nouvelle valeur est la somme des deux dernières valeurs calculées. Évidemment il faut partir de valeurs initiales qui sont 0 et 1.

Mathématiquement si on note (f_n) la suite alors on peut la définir pour tout entier naturel n par :

- $f_0 = 0$ et $f_1 = 1$
- $f_n = f_{n-1} + f_{n-2}$ pour tout entier $n \geqslant 2$

Il est naturel de traduire cette définition sous la forme d'une fonction récursive, pour calculer le terme de rang n de la suite.

Les deux cas de base $n = 0$ et $n = 1$ peuvent être regroupés dans une seule conditionnelle mais la partie réduction doit comporter deux appels récursifs. On parle dans ce cas de **récursivité multiple**.

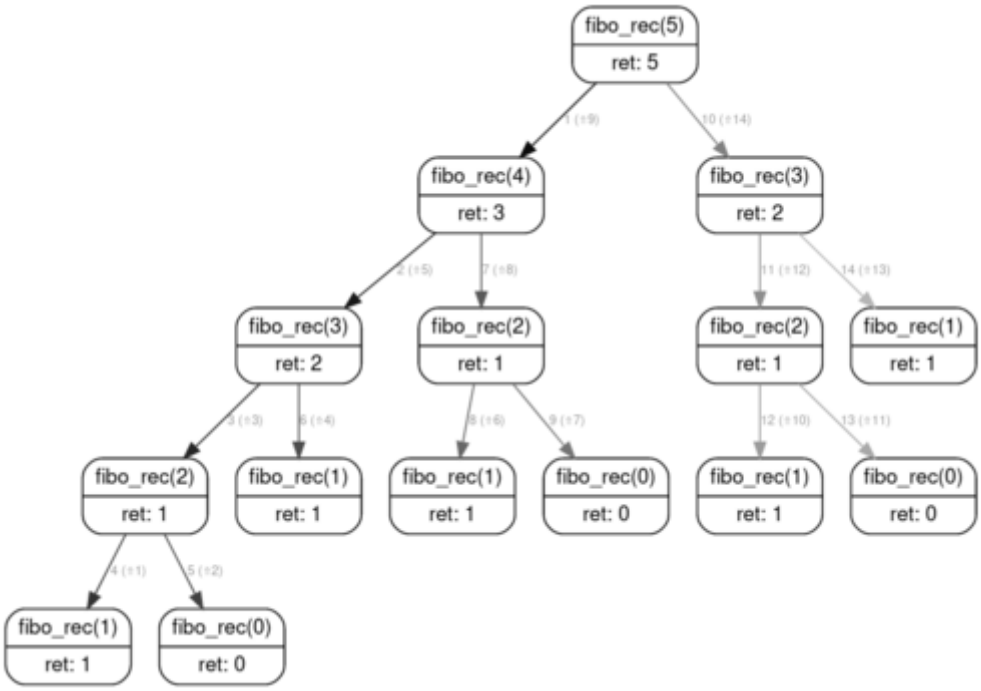
```
def fibo(n):
    """Fonction enveloppe pour ne vérifier la précondition sur n qu'une seule fo
    assert isinstance(n, int) and n >= 0
    return fibo_rec(n)

def fibo_rec(n):
    if n <= 1:  # 2 cas de bases n = 0 ou n = 1
        return n
    return fibo_rec(n - 1) + fibo_rec(n - 2)
```

"Point de cours 4"

Si une fonction récursive permet d'exprimer une solution à un problème de façon plus lisible et élégante qu'une fonction itérative, il faut se méfier de la **complexité** cachée.
Par exemple l'arbre d'appels du calcul de `fibo_rec(5)` met en évidence qu'on effectue plusieurs fois les mêmes calculs. En effet les sous-problèmes obtenus par réduction se

chevauchent. Nous verrons plus tard comment enregistrer les calculs effectués pour être plus efficace.



"Point de cours 5"

Lorsqu'une fonction est appelée, un **contexte** est créé qui va contenir les variables locales de cette fonction.

Les contextes des appels récursifs sont sauvegardés dans une **pile d'appels**.

> **Chaque appel récursif consomme de la mémoire dans la pile d'appels**. Une boucle réalisant le même nombre de répétitions n'a pas besoin d'enregistrer des états antérieurs pour y revenir. Une version récursive d'un algorithme est donc en général plus gourmande en mémoire que la version itérative, c'est pourquoi les langages de programmation comme Python prévoient un mécanisme de *limitation de la taille de la pile d'appels*. On peut la lire et la modifier avec des `getter` et `setter`.

```
>>> import sys
>>> sys.getrecursionlimit()  # lecture de la taille  pile d'appels
1000
>>> sys.setrecursionlimit(3000)  # modification de sa taille
>>> sys.getrecursionlimit()
3000
```