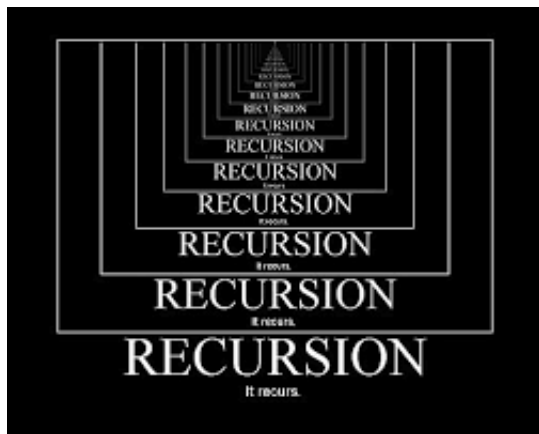


Informatique et Calcul Scientifique

Cours 7.3 : Algorithmes récursifs

13.11.2024



- ▶ Un algorithme récursif est un algorithme qui résout une instance d'un problème en résolvant une ou plusieurs instances de taille plus petite et combinant les solutions obtenues pour obtenir la solution à l'instance initiale du problème.
- ▶ On ramène donc la résolution du problème à la résolution d'un ou plusieurs cas plus simples.
- ▶ On parle du paradigme “**diviser-pour-régner**”.
- ▶ La correctitude de l'algorithme pour une instance d'une certaine taille découlera en général de la correctitude pour les instances plus petites.
- ▶ Le temps de parcours de l'algorithme sera exprimé en fonction du temps de parcours sur les instances plus petites.

- Pour $n \in \mathbb{N}$, la factorielle de n est définie comme

$$n! = \begin{cases} n \cdot (n-1)! & , n \geq 1 \\ 1 & , n = 0 \end{cases}$$

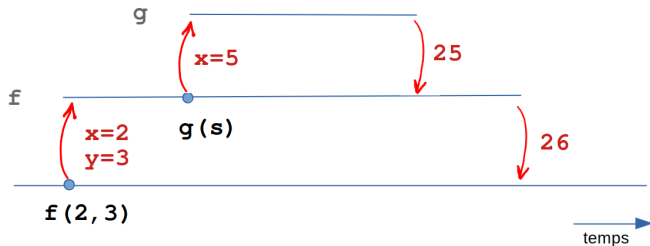
- On peut calculer la valeur de $n!$ à partir de celle de $(n-1)!$, c'est-à-dire ramener la résolution du problème pour une instance de taille n à la résolution du problème pour une instance de taille $n-1$.

```
def fact(n):  
    '''  
    Entree: entier naturel n  
    Sortie: n!  
    '''  
    if n == 0:  cas de base  
        return 1  
    f = fact(n-1)  appel récursif  
    return n*f
```

- Pour le **cas de base**, la valeur de la fonction est définie directement, sans appels récursifs.
 - On peut définir un ou plusieurs cas de base.
- On peut effectuer un ou plusieurs appels récursifs.
- fact calcule la factorielle de n avec n appels récursifs.

Appels de fonctions

```
def g(x):  
    return x**2  
  
def f(x,y):  
    s = x+y  
    return g(s)+1  
  
x = f(2,3)
```



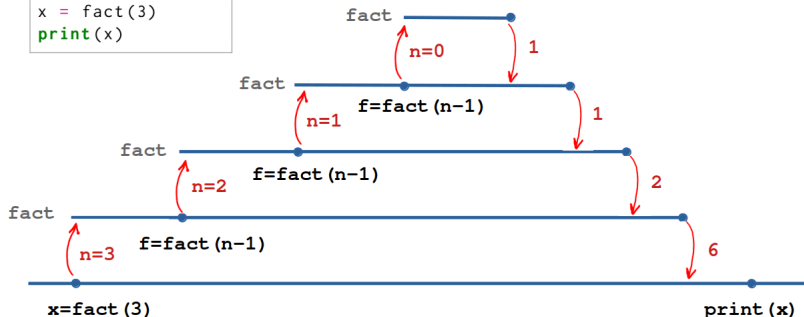
Vous pouvez suivre les appels récursifs de la fonction factorielle :

- ▶ sur [Python Tutor](#)
- ▶ sur [Recursion Visualizer](#)

Factorielle - appels récursifs

```
def fact(n):  
    if n == 0:  
        return 1  
    f = fact(n-1)  
    return n*f
```

```
x = fact(3)  
print(x)
```



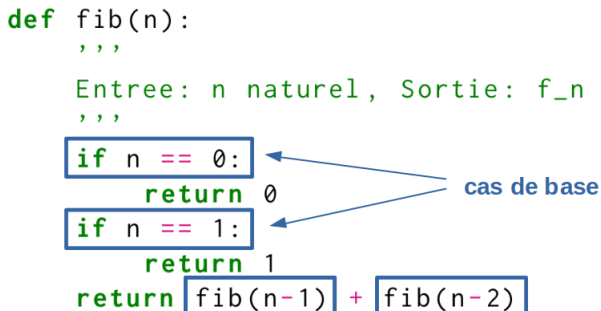
- Pour $n \in \mathbb{N}$, le n ème nombre de Fibonacci est défini comme

$$f_n = \begin{cases} f_{n-1} + f_{n-2} & , n \geq 2 \\ 1 & , n = 1 \\ 0 & , n = 0 \end{cases}$$

- On peut ramener la résolution du problème pour une instance de valeur n à la résolution du problème pour deux instances de valeur $n - 1$ et $n - 2$.

Suite de Fibonacci

```
def fib(n):  
    '''  
    Entree: n naturel, Sortie: f_n  
    '''  
    if n == 0:  
        return 0  
    if n == 1:  
        return 1  
    return fib(n-1) + fib(n-2)
```

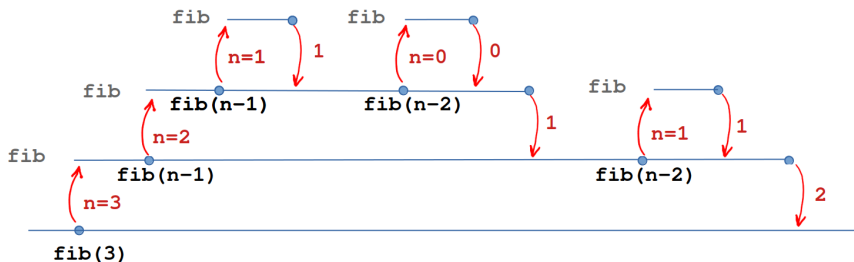


cas de base

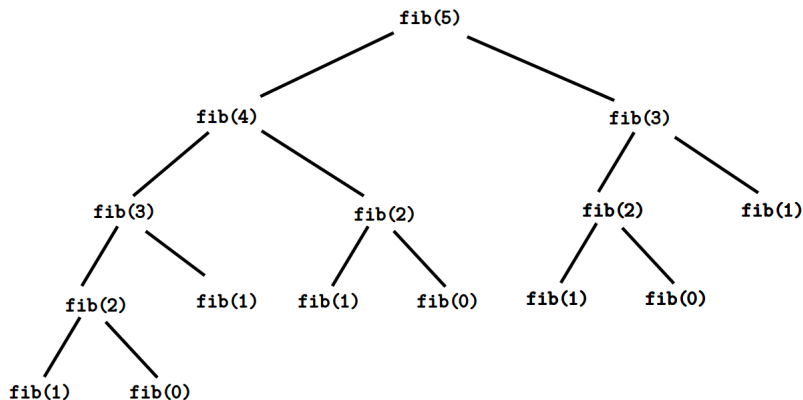
appels récursifs

Fibonacci - appels récursifs

```
def fib(n):  
    if n == 0:  
        return 0  
    if n == 1:  
        return 1  
    return fib(n-1) + fib(n-2)  
  
print(fib(3))
```



Fibonacci - appels récursifs



. On peut prouver que pour l'entrée n le nombre d'appels récursifs de `fib_rec` et son temps de parcours sont **exponentiels** en n .

On a déjà vu un algorithme **itératif** (contenant une boucle) qui prend en entrée un entier n strictement positif et calcule la somme des nombres de 1 à n .

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

Peut-on écrire un algorithme **récuratif** (sans boucle) `somme_rec` qui calcule la même somme ?

- Pour n entier strictement positif,

$$\sum_1^n i = \begin{cases} 1 & , n = 1 \\ \sum_1^{n-1} i + n & , n > 1 \end{cases}$$

- `somme_rec(n)` doit donc retourner `1` si $n = 1$ (cas de base),
et `somme_rec(n-1) + n` sinon (appel récursif).

Somme récursive

```
def somme_rec(n):  
    '''  
    Entree: n entier strictement positif  
    Sortie: somme des nombres entiers de 1 a n inclus  
    '''  
    if n == 1:  
        return 1  
    return somme_rec(n-1) + n
```

Algorithmes récursifs vs algorithmes itératifs

- ▶ On peut toujours remplacer un algorithme récursif par un algorithme itératif qui fait le même travail. Alors pourquoi écrire des algorithmes récursifs ?
- ▶ Certains problèmes ont naturellement une structure récursive (ex. calcul de la factorielle) et il est donc plus facile d'y réfléchir récursivement.
- ▶ Il est plus facile de garantir la correctitude d'une implémentation récursive que celle d'une implémentation itérative¹.
- ▶ Mais les algorithmes récursifs sont souvent moins efficaces...

1. C'est la prémisse de la *programmation fonctionnelle*.