

Récurtivité et programmation dynamique ? La récurtivité est certes attrayante, mais elle a son côté obscure... La programmation dynamique remédie à l'inconvénient en question : le coût.

### Récurtivité :

Considérons une fonction Python  $f(n)$ , d'argument entier  $n$ . On dit qu'elle est *récurtive* si elle fait appel au moins une fois à  $f(k)$ , pour  $k < n$ .

### Premier exemple

Prenons l'exemple d'une fonction Python revoyant le terme de rang  $n$  d'une suite arithmético-géométrique, par exemple la suite définie par son premier terme  $U_0=10$  et par la relation de récurrence:  $U_{n+1} = 0,7U_n + 2$ .

def u(n):

    if n == 0:

        return 10

    else:

        return 0.7 \* u(n-1) + 2

Premier constat : il faut penser à retourner une constante pour une valeur précise de  $n$ . Ici, pour  $n = 0$ , on retourne le premier terme de la suite.

Second constat : pour toute valeur de  $n$  différente de 0, on retourne une expression directement inspirée de la relation de récurrence de la suite. Ainsi, pour calculer par exemple  $u(10)$ , il faut calculer  $u(9)$ ; mais pour avoir  $u(9)$ , il faut calculer  $u(8)$ , et ceci jusqu'à  $u(0)$  qui vaut 10.

### Complexité

Regardons la complexité de l'exemple précédent, où l'ordre et le degré de récurrence ne sont que de 1 : on ne fait appel à la fonction elle-même qu'une seule fois (degré de récurrence= 1),

Notons  $C(k)$  la complexité de  $u(k)$  pour un  $k > 0$  fixé. Alors,

$$C(k) = C(k - 1) + 3.$$

En effet, il y a 3 opérations élémentaires (un test sur  $n$ , une addition et une multiplication) et il y a  $C(k-1)$  opérations lors de l'appel à  $u(k-1)$ .

Ainsi,  $C(n)$  est elle-même une suite arithmético-géométrique, et on a :

$$C(n) = C(n - 1) + 3$$

$$= [C(n - 2) + 3] + 3 = C(n - 2) + 2 \times 3$$

$$= [C(n - 3) + 3] + 2 \times 3 = C(n - 3) + 3 \times 3$$

= ...

$$= C(0) + 3n$$

$$= 3n + 1$$

car  $C(0) = 1$  (pour  $n = 0$ , il n'y a que le test sur  $n$ ).

On peut alors dire que , ce qui peut aller car ce n'est pas une classe de complexité gênante.

### **Deuxième exemple: cas général d'ordre 1**

Supposons que la fonction  $f(n)$  comporte  $k$  appels à elle-même avec un argument diminué de 1 (autrement dit, on fait appel à  $f(n-1)$   $k$  fois).

Dans ce cas, pour un entier  $n > 0$ ,

$$C(n) = k \times C(n - 1) + p$$

où  $p$  est le nombre d'opérations élémentaires faites indépendamment de  $n$ . Là encore, la complexité est une suite arithmético-géométrique, mais le coefficient devant  $C(n-1)$  étant différent de 1, la formule finale va changer.

En effet, on arrive à démontrer mathématiquement que:

$$C(n) = O(k^n).$$

La complexité est donc ici exponentielle, ce qui n'est pas bon du tout ! (imaginez pour le temps nécessaire pour arriver à la fin de l'exécution du script...)

### **Troisième exemple: récursivité d'ordre supérieur à 1**

Là, on considère une fonction  $f(n)$  faisant appel à  $f(n-1)$ , mais aussi à  $f(n-2)$ ,  $f(n-3)$ , etc. Un exemple ultra-classique est la suite de Fibonacci:

def fibonacci(n):

    if n < 2:

        return n

    else:

        return fibonacci(n-1) + fibonacci(n-2)

La suite sera définie et calculée ainsi:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

Dans cet exemple,

$$C(n) = C(n - 1) + C(n - 2) + 2$$

car quel que soit la valeur de  $n$ , il y a 1 addition et 1 test (donc 2 opérations élémentaires) auxquelles on doit ajouter la complexité de la fonction au rang précédent et au pénultième rang.

Autant donc dire que la complexité est exponentielle encore une fois.

Il va donc de soit que trouver une autre façon de coder s'impose dans ce cas...

Des solutions miracles pour baisser la complexité de la récursivité: vers la programmation dynamique

### **Diviser pour régner**

J'ajoute tout de même un point sur la complexité. Imaginons que notre fonction puisse être coupée en deux sous-problèmes de taille  $n/2$  et que:

$$C(n) = 2C(n/2) + f(n)$$

Alors,  $C(n) = n \log(n)$

La complexité est donc moindre que celle du script original (ce dernier cas est par exemple celui du tri fusion).

C'est un cas particulier du *Master Theorem*.

### **Alternative à la récursivité : programmation dynamique**

Bon, tout ça, c'est bien beau, mais pour notre "petit" problème concernant la suite de Fibonacci, ça ne nous aide pas... On va donc voir les choses autrement: pour calculer  $f(9)$ , nous avons vu qu'il nous fallait calculer  $f(8)$  et  $f(7)$  mais pour calculer ces derniers, il nous faut calculer:

pour  $f(8)$ ,  $f(7)$  [encore une fois] et  $f(6)$ ;

pour  $f(7)$ ,  $f(6)$  [calculé précédemment... donc ça fait doublon ...

On voit bien qu'il y a des termes qui sont calculés plusieurs fois... ce qui n'est pas "finot"... Et comme nous voyons que nous avons plusieurs fois besoin d'un même terme, autant le stocker... C'est le principe de la programmation dynamique.

### **Approche Top Down (mémoïsation)**

C'est une approche récursive pour laquelle le principe est:

d'utiliser la formule de récurrence; avant tout calcul, de regarder si le résultat est stocké dans une liste.

Cette approche nécessite donc deux fonctions: la première, admettant pour arguments un entier  $n$  et une liste  $L$ , servira à retourner le dernier élément de la liste, celui que l'on vient de calculer.

```
def fibonacci(n , L):
```

```
    if n == 0 or n == 1:
```

```
        return n
```

```
    else:
```

```
        L[n] = fibonacci(n-1 , L) + fibonacci(n-2 , L)
```

```
        return L[n]
```

La seconde fonction servira à calculer le terme souhaité:

```
def F(n):
```

```
    L = [0] * (n+1)
```

```
    return fibonacci(n , L)
```

```
>>> F(10)
```

```
55
```

De façon intuitive, on comprend que la complexité d'une telle approche est largement meilleure que la solution récursive initiale car il y a bien moins de calculs.

Le terme "Top Down" prend alors tout son sens: on part du haut ("top") du problème (le problème initial) et on le découpe pour arriver à résoudre des problèmes de tailles plus petites ("down").

## Approche Bottom Up

C'est une approche itérative pour laquelle:

on commence par résoudre les sous-problèmes de la taille la plus petite à ceux de la taille la plus grande; on stocke les résultats au fil de la résolution des sous-problèmes.

Le terme "Bottom Up" prend alors tout son sens : on part du plus petit problème ("bottom") pour aller au plus grand ("up").

Cela donne pour la suite de Fibonacci:

```
def fibonacci(n):  
    L = [0] * (n+1)  
    L[1] = 1  
    for i in range(2, n+1):  
        L[i] = L[i-1] + L[i-2]  
    return L[n]
```

Cette approche est-elle plus intuitive que l'approche "Top Down".

Une autre façon de voir que la récursivité ou la programmation dynamique

Je me suis servi de la suite de Fibonacci pour exposer le principe de la programmation dynamique, mais il va de soit que l'on peut calculer les termes de cette suite autrement. Voici un autre exemple parmi tant d'autres:

```
def fibonacci(n, a = 0, b = 1):  
    if n == 0:  
        return a  
    elif n == 1:  
        return b  
    else:  
        for i in range(n-1):  
            tmp = a  
            a = b  
            b = b + tmp  
        return b
```

La complexité de ce dernier est en  $O(n)$  donc aussi bien que l'approche dynamique!

Il faudra donc retenir une chose: la récursivité, c'est peut-être beau mais pas toujours optimal donc, pour améliorer les performances, on peut opter pour la programmation dynamique !