RÉCURSIVITÉ

Chapitre 2

jeudi 20 septembre 2018

Lycée Chaptal – PT*

http://www.mickaelprost.fr

On cherche à écrire une fonction `u(n)` qui retourne 2^n en utilisant seulement des multiplications ; l'opérateur `**` est donc proscrit.

$$\forall n \in \mathbb{N}^* \quad u(n) = 2^n = 2 \times \cdots \times 2$$

```
>>> def puis_it(n):
...     res = 1
...     for i in range(1,n+1):
...         res *= 2
...     return res
...
>>> puis_it(5)
32
```

VERSION ITÉRATIVE

$$\forall n \in \mathbb{N}^* \quad u(n) = 2^n = 2 \times 2^{n-1} = 2 \times u(n-1)$$

```
>>> def puis_rec(n):
...     if n == 0:
...         return 1
...     return 2*puis_rec(n-1)
...
>>> puis_rec(5)
32
```

VERSION RÉCURSIVE

- Principe général de la récursivité :
  Découper un problème en sous-problèmes de même nature jusqu'à obtenir des sous-problèmes dont la résolution sera triviale.
- Une fonction est dite récursive si le corps de la fonction fait appel à elle-même.
- Le recours à une fonction récursive est naturelle lorsqu'apparaît une relation de récurrence de la forme :

$$u(n) = f(n, u(n-1))$$

ou même,

$$u(n) = f(n, u(n-1), u(n-2), \ldots, u_0)$$

En pratique :

- Un algorithme récursif fera *toujours* appel à lui-même.
- On prévoira *toujours* un ou plusieurs cas initiaux.
- L'algorithme récursif changera d'état, *toujours* en direction des cas initiaux.

```
>>> def puis_rec(n):
...     if n == 0:
...         return 1
...     return 2*puis_rec(n-1)
...

>>> puis_rec(5)
32
```

VERSION RÉCURSIVE

- On prouve généralement la correction et la terminaison d'une fonction récursive conjointement, à l'aide d'une récurrence.
- Exemple :

```
>>> def puis_rec(n):
...     if n == 0:
...         return 1
...     return 2*puis_rec(n-1)
...
>>> puis_rec(5)
32
```

VERSION RÉCURSIVE

Revenons sur le fonctionnement du programme `puis_rec(n)`.

- Lorsqu'on appelle la fonction pour `n = 3`, la machine calcule successivement :

$$\texttt{puis\_rec}(3) = 2 * \texttt{puis\_rec}(2)$$
$$\texttt{puis\_rec}(2) = 2 * \texttt{puis\_rec}(1)$$
$$\texttt{puis\_rec}(1) = 2 * \texttt{puis\_rec}(0)$$

- Pour `n = 0`, l'instruction `return 1` permet de poursuivre les calculs, mais cette fois-ci dans le sens inverse :

$$\texttt{puis\_rec}(0) = 1$$
$$\texttt{puis\_rec}(1) = 2 * \texttt{puis\_rec}(0) = 2$$
$$\texttt{puis\_rec}(2) = 2 * \texttt{puis\_rec}(1) = 4$$
$$\texttt{puis\_rec}(3) = 2 * \texttt{puis\_rec}(2) = 8$$

- Attention à ne pas dépasser la capacité de la pile de récursivité.

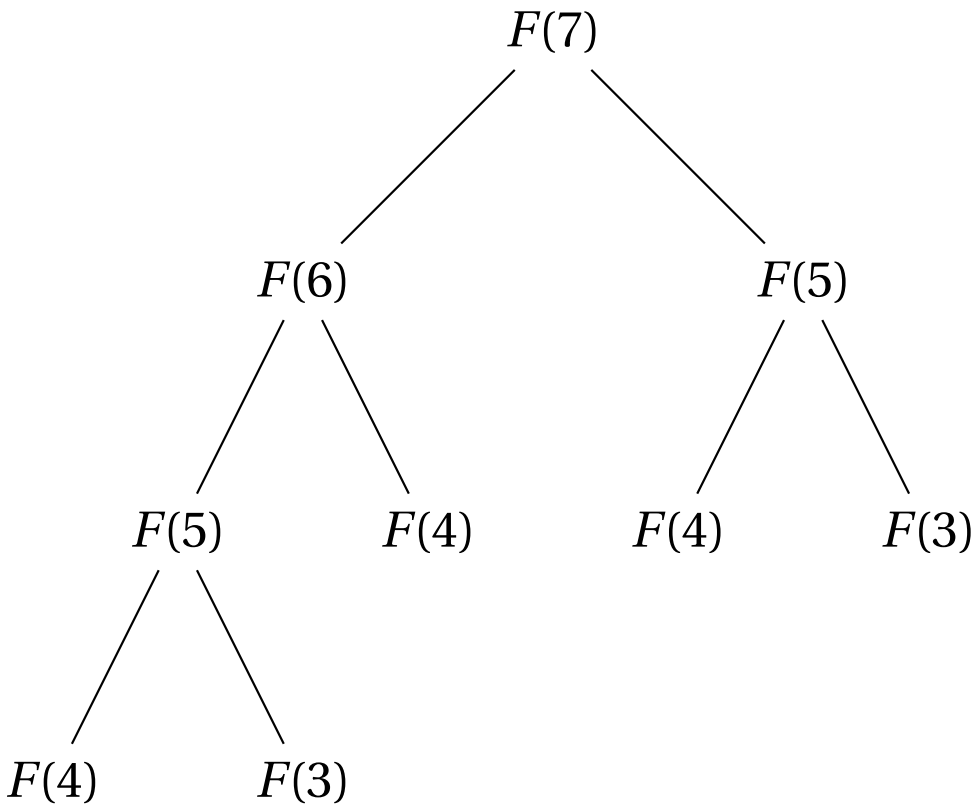
```
>>> RuntimeError: maximum recursion depth exceeded in comparison
```

Intéressons-nous maintenant à la suite de Fibonacci définie par :

$$F_0 = 0, \quad F_1 = 1 \quad \text{et} \quad \forall n \in \mathbb{N} \quad F_{n+2} = F_{n+1} + F_n$$

```
>>> def F(n):
...     if n == 0:
...         return 0
...     elif n == 1:
...         return 1
...     return F(n-1)+F(n-2)
...
>>> F(7)
13
```

Quel est le nombre d'appels récursifs nécessaires pour calculer $F(7)$?



Ce qui explique que pour $n = 42$:

```
>>> F(42)
RuntimeError: maximum recursion depth exceeded in comparison
```

On peut cependant modifier la capacité de la pile de récursivité :

```
>>> from sys import *
>>> getrecursionlimit()
1000
>>> setrecursionlimit(10000)
```

Vaut-il mieux travailler de façon récursive ou itérative ?

1. Approche naturelle pour résoudre certains problèmes.
2. Souvent plus coûteux en occupation mémoire.
3. Attention au dépassement de la capacité.
4. Principe de dérécursification : traduction d'un algorithme récursif en algorithme itératif.
   Mais en pratique...

Conclusion ?


ASK
Alice & Bob