La récursivité

Lycée Blaise Pascal

Octobre 2015

L'algorithme d'Euclide

Extrait d'un cours de TS :

Proposition

Soient a et b deux entiers naturels non nuls et soit r le reste dans la division euclidienne de a par b. On a : $PGCD(a;b) = PGCD(b;r)$.

a	b	q	r
1768	1001	1	767
1001	767	1	234
767	234	3	65
234	65	3	39
65	39	1	26
39	26	1	13
26	13	2	0
13	0		

Implémentation en Python

Puisque le PGCD de a et de b et le PGCD est le reste de b et du reste r dans la division euclidienne de a par b :

```
def pgcd(a,b):
    return pgcd(b,a%b)
```


Ummagumma

Mise en abyme ou récursivité ?

Implémentation en Python

On obtient :

```
>>>pgcd(1768,1001)
ZeroDivisionError: integer division or modulo by
    zero
```

... puisque pour calculer PGCD(1768, 1001),
il faut calculer PGCD(1001, 767)
et pour cela calculer PGCD(767, 234)
...
et pour cela calculer PGCD(26, 13)
et pour cela calculer PGCD(13, 0)
et pour cela calculer le reste dans une division euclidienne par 0.

Inclure un test d'arrêt

On arrête le processus lorsque $b = 0$:

```
def pgcd(a,b):
    if b == 0:
        return a
    else:
        return pgcd(b,a%b)
```

donne comme espéré :

```
>>>pgcd(1768,1001)
13
```


Une suite récurrente

Soit (u_n) la suite définie sur $\mathbb{N}$ par :

$$\begin{cases} u_0 &= 1 \\ u_n &= n \times u_{n-1} \text{ pour } n > 0 \end{cases}$$

Cette suite s'implémente naturellement par :

```
def factorielle(n):
    if n == 0:
        return 1
    else:
        return n*factorielle(n-1)
```


La fonction factorielle

La fonction `factorielle()` donne :

```
>>>factorielle(120)
6689502913449127057588118054090372586752746333138029

8102956713523016335572449629893668741652719849813081

5763789321409055253440858940812185989848111438965000

5964960521256960000000000000000000000000000
```

ou bien (après un long moment d'attente) :

```
>>>factorielle(-2)
RuntimeError: maximum recursion depth exceeded
   in comparison
```

La fonction factorielle

Nous améliorons la fonction `factorielle()` :

```
def factorielle(n):
    assert n >= 0, 'n doit être positif'
    if n == 0:
        return 1
    else:
        return n*factorielle(n-1)
```

```
>>>>>>factorielle(-2)
AssertionError: n doit être positif
```


Terminaison d'une fonction récursive : le cas de factoriel

La fonction `factorielle()` **termine**. En effet :

- Si $n < 0$, la fonction renvoie une exception.
- Si $n = 0$, la fonction renvoie 1.
- Si $n > 0$, la fonction s'appelle elle-même pour une suite strictement décroissante de nombres entiers positifs : le nombre d'appels est donc **fini**.

Terminaison d'une fonction récursive

Pour la suite de Syracuse, on ne sait pas démontrer que la fonction termine ou non.

```
def syracuse(n):
    assert n> 0, 'n doit être strictement
       positif'
    if n == 1:
        return 1
    elif n%2 == 0:
        return syracuse(n//2)
    else:
        return syracuse(3*n+1)
```


Correction d'une fonction récursive : le cas factoriel

Afin de démontrer que la fonction `factorielle(n)` calcule bien $n!$ pour tout entier naturel n, on procède à l'aide d'une démonstration par récurrence :

Pour tout entier naturel n, on note $P(n)$ la proposition :

$P(n)$: " La fonction `factorielle(n)` retourne $n!$ "

Correction d'une fonction récursive : le cas factoriel

Démonstration par récurrence :

- **Initialisation** :
  `factorielle(0)` renvoie 1.
- **Hérédité** :
  `factorielle(n+1)` renvoie `(n+1)*factorielle(n)`.
  Or par hypothèse de récurrence, `factorielle(n)` renvoie $n!$,
  donc `factorielle(n+1)` renvoie $(n+1) \times n! = (n+1)!$.
- **Conclusion** :
  Pour tout entier naturel n, `factorielle(n)` renvoie $n!$.

La correction de l'algorithme est démontrée.

Complexité d'une fonction récursive : le cas factoriel

```
def factorielle(n):
    if n == 0:
        return 1
    else:
        return n*factorielle(n-1)
```

Pour calculer `factorielle(n)`, il faut faire une comparaison, une multiplication et calculer `factorielle(n-1)`.

Si on note c_n la complexité, on a donc :

$$\forall n \in \mathbb{N}^*, \quad c_n = 1 + 1 + c_{n-1}$$

Ainsi (c_n) est une suite arithmétique de raison 2.
La fonction factorielle est de complexité $O(n)$.

La pile d'exécution : le cas factoriel

A chaque appel récursif de la fonction, il faut stocker dans une pile les résultats intermédiaires pour les restituer ensuite :

```
>>> factorielle(4)
factorielle <- 4          # Empilage de 4
 factorielle <- 3         # Empilage de 3
  factorielle <- 2        # Empilage de 2
   factorielle <- 1       # Empilage de 1
    factorielle <- 0      # Empilage de 0
    factorielle -> 1      # Dépilage et retour de
       1
   factorielle -> 1       # Dépilage et retour de
      1*1
  factorielle -> 2        # Dépilage et retour de
     2*1
 factorielle -> 6         # Dépilage et retour de
    3*2
factorielle -> 24         # Dépilage et retour de
   4*6
```

A ne pas perdre de vue : compter le nombre d'opérations est une

La pile d'exécution : faut pas pousser Mémé

Quand on remplit trop la pile d'exécution :

```
RuntimeError: maximum recursion depth exceeded
```

Certes, Python permet d'utiliser la récursivité, mais ce n'était pas la préoccupation principale de Guido Van Rossum lorsqu'il a créé ce langage.

Par défaut, le nombre d'appels récursifs est limité à 1000.
Pour changer cela :

```
>>> import sys

>>> sys.setrecursionlimit(10000)
```

La récursivité terminale : variante de la fonction factorielle

```
def factorielle(n):
    if n == 0:
        return 1
    else:
        return n*factorielle(n-1)
```

```
def factorielleTerm(n,accu=1):
    if n == 0:
        return accu
    else:
        return factorielleTerm(n-1, n * accu)
```

Cette fonction se distingue de la précédente par l'usage d'un paramètre supplémentaire et l'absence d'opération entre l'appel récursif et l'instruction `return`.

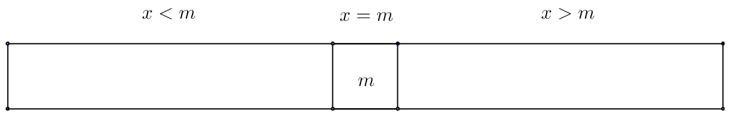
La récursivité terminale : variante de la fonction factorielle

```
>>> factorielleTerm(4)
factorielleTerm <- (4,)
 factorielleTerm <- (3, 4)       # accu = 4
  factorielleTerm <- (2, 12)     # accu = 12
   factorielleTerm <- (1, 24)    # accu = 24
    factorielleTerm <- (0, 24)   # accu = 24
    factorielleTerm -> 24
   factorielleTerm -> 24
  factorielleTerm -> 24
 factorielleTerm -> 24
factorielleTerm -> 24
```

Il n'est pas nécessaire d'empiler les valeurs puisqu'on ne les réutilise pas : pas de problème de pile d'exécution.
Cela dit, Python ne tire pas profit de cette idée et la récursivité terminale est sans intérêt dans ce langage.

Nouvel exemple : la recherche dichotomique

On dispose d'un tableau t de nombres triés dans l'ordre croissant et on veut déterminer si un nombre x appartient ou non à ce tableau.
On compare x avec la valeur centrale du tableau :



- Si x est la valeur centrale, c'est terminé !
- Si x est inférieur à la valeur centrale, on cherche dans le sous-tableau des valeurs inférieures à cette dernière.
- Si x est supérieur à la valeur centrale, on cherche dans le sous-tableau des valeurs supérieures à cette dernière.

Recherche dichotomique : une version récursive

```
def rechDicho(x, tab):
    d = 0
    f = len(tab)-1
    return rd(x, tab, d, f)

def rd(x, tab, d, f):
    m = (d + f)//2
    if tab[m] == x:
        return True
    if f == d:
        return False
    if tab[m] > x:
        return rd(x, tab, d, m-1)
    else:
        return rd(x, tab, m+1, f)
```

Complexité d'une recherche dichotomique

Notons n la longueur du tableau `tab`.

Pour effectuer une recherche dichotomique sur un tableau de longueur n, il faut faire 3 comparaisons, 2 opérations et effectuer la recherche dichotomique sur un tableau de longueur (presque) $\frac{n}{2}$.

En notant c_n la complexité pour un tableau de longueur n, on a :

$$c_n = 5 + c_{\frac{n}{2}}$$

Complexité d'une recherche dichotomique

$$\begin{aligned} c_n &= 5 + c_{\frac{n}{2}} \\ &= 10 + c_{\frac{n}{4}} \\ &= 15 + c_{\frac{n}{8}} \end{aligned}$$

$$c_n = 5p + c_{\frac{n}{2^p}}$$

où p est tel que $\frac{n}{2^p} = 1$.
C'est-à-dire :

$$p = \log_2(n)$$

Ainsi la complexité de la recherche dichotomique est en $O(\log(n))$.

Nouvel exemple : la suite de Fibonacci

On considère la suite définie pour tout entier naturel n par :

$$\begin{cases} f_0 &= 1 \\ f_1 &= 1 \\ f_{n+2} &= f_{n+1} + f_n \end{cases}$$

dont on rappelle pour mémoire les premiers termes :

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584$$

La suite de Fibonacci en Python

Une version naturelle :

```
def fibo(n):
    # Pour n>=1
    if n <= 2:
        return n
    else:
        return fibo(n-1)+fibo(n-2)
```

Complexité de la suite de Fibonacci

Notons à nouveau c_n la complexité. On a :

$$\forall n \geqslant 3, \qquad c_n = 1 + c_{n-1} + c_{n-2} \qquad \text{[E]}$$

On définit alors d_n tel que : $c_n = 2d_{n+1} - 1$.
[E] donne alors :

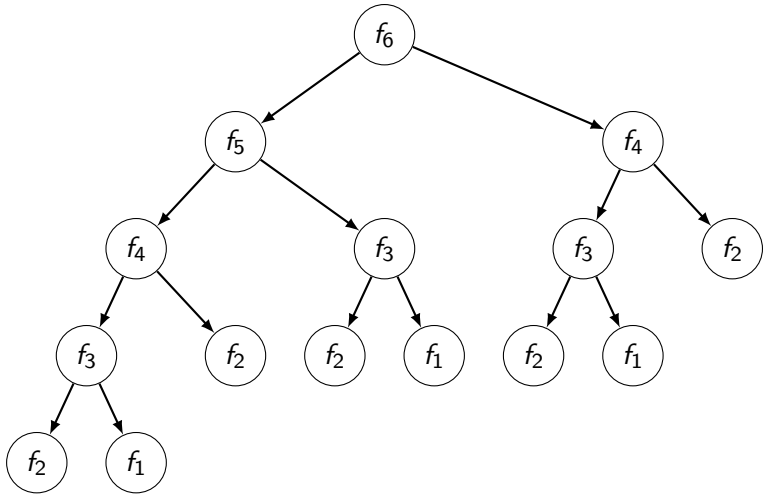
$$2d_{n+1} - 1 = 1 + 2d_n - 1 + 2d_{n-1} - 1$$

c'est-à-dire :

$$d_{n+1} = d_n + d_{n-1}$$

La suite (d_n) est la suite de Fibonacci (f_n). Comme on sait que en $+\infty$, $f_n \sim \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2}\right)^n$, alors on en déduit que la complexité de la suite de Fibonacci est en $O(\phi^n)$.

La suite de Fibonacci : arbre des appels



On constate que certains termes sont calculés plusieurs fois : cette méthode n'est pas efficace.

La suite de Fibonacci en Python : version efficace

```
def fiboL(n):
    listeFibo = [0]*(n+1)
    listeFibo[0], listeFibo[1], listeFibo[2] =
       1, 1, 2
    return f(n,listeFibo)

def f(n,lF):
    if n <= 2:
        return n
    elif lF[n-2] != 0:
        if lF[n-1] != 0:
            lF[n] = lF[n-2]+lF[n-1]
            return lF[n]
        else:
            lF[n] = lF[n-2]+f(n-1,lF)
            return lF[n]
    else:
        lF[n] = f(n-2,lF)+f(n-1,lF)
        return lF[n]
```


La suite de Fibonacci en Python : version efficace

Chaque terme de la suite n'est calculé qu'une fois !

```
>>> fiboL(8)
f <- (8, [1, 1, 2, 0, 0, 0, 0, 0, 0])
 f <- (6, [1, 1, 2, 0, 0, 0, 0, 0, 0])
  f <- (4, [1, 1, 2, 0, 0, 0, 0, 0, 0])
   f <- (3, [1, 1, 2, 0, 0, 0, 0, 0, 0])
   f -> 3
  f -> 5
  f <- (5, [1, 1, 2, 3, 5, 0, 0, 0, 0])
  f -> 8
 f -> 13
 f <- (7, [1, 1, 2, 3, 5, 8, 13, 0, 0])
 f -> 21
f -> 34

34
```

La suite de Fibonacci en Python : version efficace

```
>>> fiboL(3000)

 66439046036696007228021784786602838424416351245278325

94055797655426212141612192573964498109829998203911322

68028094651324463493319944094349260190453427237491885

03169946784735513206351010996193829731816225856873369

97843735278975554894868417261317338143401291756224504

16051010258971732359906627702037564387865175305471011

37488491402526861201040326470251455989566759021350105

69097831249594364698255583142897013542271517846028657

07806246751070565698228205428466603218138388962758197
```