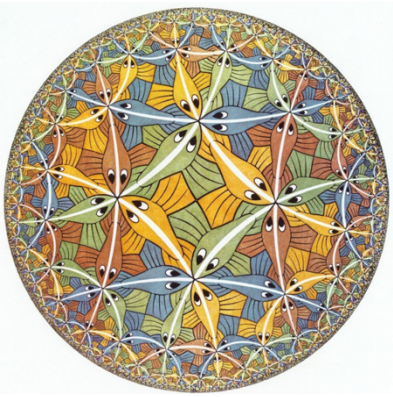
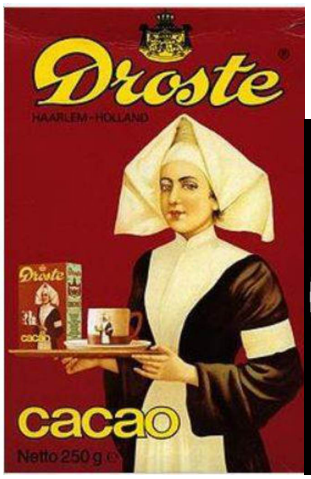
Informatique Pour Tous

Cours 19 – Découverte de la récursivité

En mathématiques, en informatique, en biologie, mais aussi dans notre quotidien, nous faisons souvent face à des situations où un problème doit être résolu en utilisant une méthode de résolution qui est répétée plusieurs fois. Dans l'itération, cette méthode est appliquée par paliers de façon séquentielle, dans la récursivité, la méthode s'appelle elle-même. La récursivité est un principe de pensée exigeant et est souvent désignée comme « trop compliquée ». La récursivité est cependant si fondamentale qu'il n'est pas possible de l'éviter.

Dans le domaine des arts, c'est l'artiste néerlandais Maurits Cornelis Escher (1898-1972) qui fait le plus grand usage de la récursivité. De son côté, la publicité a aussi utilisé la mise en abyme, rendant célèbres les fromages «La vache qui rit», le vermouth «Dubonnet», et le chocolat «Droste».

En informatique, une fonction récursive est une fonction qui s'appelle elle-même.

Pour le moment, les algorithmes que nous avons vus s'appuyaient sur des structures **ITERATIVES** (boucles `for` et `while`). On parle aussi d'algorithmes **IMPERATIFS**, ou « de bas en haut » (**bottom – up en anglais**). La logique est toujours de partir de cas simples, puis progresser vers un objectif étape par étape.

Prenons un exemple introductif d'une suite récurrente d'ordre 1 : $\begin{cases} u_0 = 1 \\ \forall n \geq 0, u_{n+1} = P(u_n) \end{cases}$ avec $P(X) = X^2 + X + 1$

On peut dès lors définir

```
def P(X):
    return X**2 + X + 1
```

Et la structure impérative permettant de déterminer le terme u_n de la suite est :

```
def u(n):
    un = 1
    for i in range(n) :
        un = P(un)
    return un
```

A / Principe de la récursivité

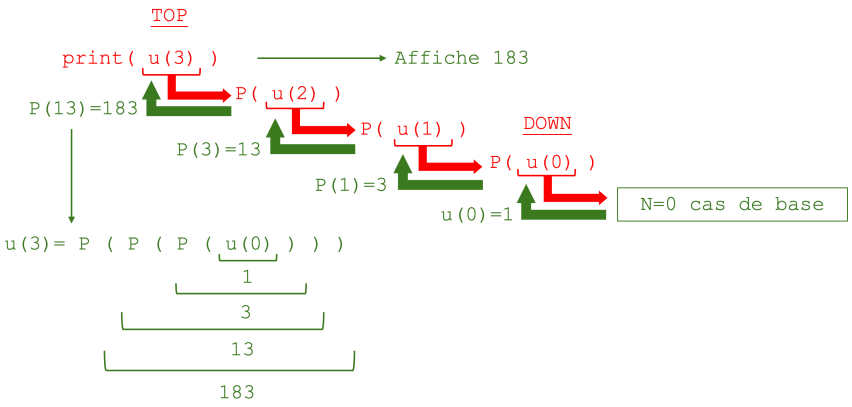
À l'inverse, penser un algorithme de façon **RECURSIVE** consiste à raisonner « de haut en bas » (**top – down**), c'est-à-dire partir d'un problème complet, et « descendre » par appels successifs d'une même fonction, jusqu'à se ramener à un **cas de base** connu.

Il y a plusieurs façons de penser la récursivité. Pour simplifier, on utilisera par la suite cette structure de code :

```
def fonc_recur( probleme ) :
    if probleme … :    # on teste si on est dans un cas de base
        return …       # si c'est le cas, on renvoie la solution connue
    else :
        new_probl = … # les problèmes successifs « baissent » vers le cas de base
        return fonc_recur( new_probl ) …    # appel récursif sur le nouveau pb
```

Voyons un exemple avec une structure récursive permettant de déterminer le terme u_n de la suite précédente :

```
def u(n) :
    if n == 0 :
        return 1
    else :
        new_n = n - 1
        return P( u( new_n) )
```



B / Pré-requis pratique : arguments optionnels

Il sera parfois pratique, entre autres pour des fonctions récursives, d'utiliser **des arguments optionnels** dans l'en-tête d'une fonction.

- S'il n'y a qu'un seul argument optionnel :
  - La déclaration se fait, par exemple, avec l'en-tête : `def fn(X, Y = 4) :`
  - L'appel se fera dans ce cas avec :

```
fn(5)      # X ← 5 (explicite)
           # Y ← 4 (implicite)
```

    ou bien avec :

```
fn(2,1)    # X ← 2 (explicite)
           # Y ← 1 (explicite)
```

- S'il y a plusieurs arguments optionnels (prenons ici l'exemple de 2 arguments optionnels) :
  - La déclaration se fait, par exemple, avec l'en-tête : `def fn2(X, Y = 4, Z = 0) :`
  - L'appel se fera, avec les valeurs implicites :

```
fn(5)      # X ← 5 (explicite)
           # Y ← 4 (implicite)
           # Z ← 0 (implicite)
```

La nouveauté pour vous ,c'est si on doit faire l'appel en voulant imposer les valeurs de Y et Z. Dans ce cas, on doit taper `fonction( arg_explicites, arg_implicite = valeur, ...)`

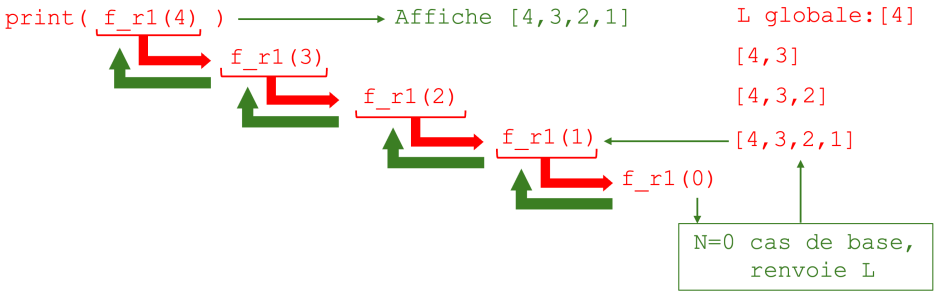
Ici notamment, on appellerait par ex. : `fn(2,Y = 1, Z = 3)`

```
# X ← 2 (exp)
# Y ← 1 (imp)
# Z ← 3 (imp)
```

Remarque : si on déclare une fonction récursive `f_rec(probleme, L = [] )`, la liste `L` en argument est globale ! → elle pourra donc être commune à tous les appels récursifs.

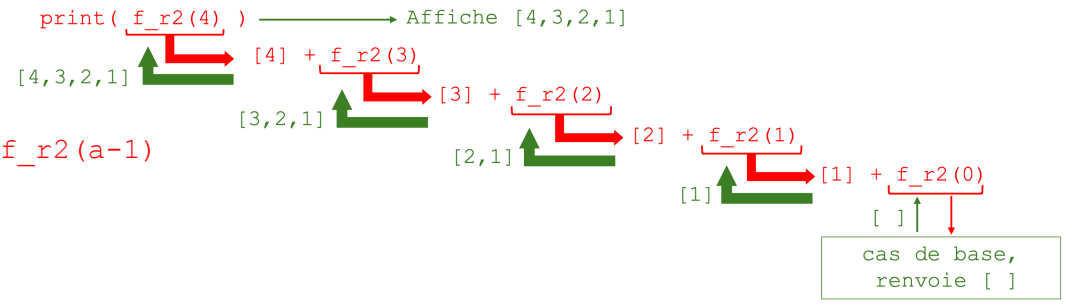
Exemple (assez haut niveau de difficulté) :

```
def f_r1(a, L = []):  # qu'affiche print(f_r1(4))?
    if a == 0:
        return L
    else :
        L.append(a)
        return f_r1(a-1)
```



On voit qu'on a ici fait « l'assemblage à la descente », et que la rencontre cas de base amorce simplement la remontée « directe » du résultat jusqu'à l'appel récursif principal. Ce code est équivalent à :

```
def f_r2(a):    # qu'affiche print(f_r2(4))?
    if a == 0:
        return []
    else :
        return [a] + f_r2(a-1)
```



Dans ce 2nd cas, on a préféré « l'assemblage à la remontée ». Dans la suite du cours / TD, **on privilégiera ce genre d'écriture.**

C / Conversion de quelques algorithmes impératifs

Dans cette partie, nous allons revoir trois algorithmes impératifs que nous avons déjà vus, s'appuyant sur une boucle `while`, et les repenser en version récursive.

C.1 / Division Euclidienne

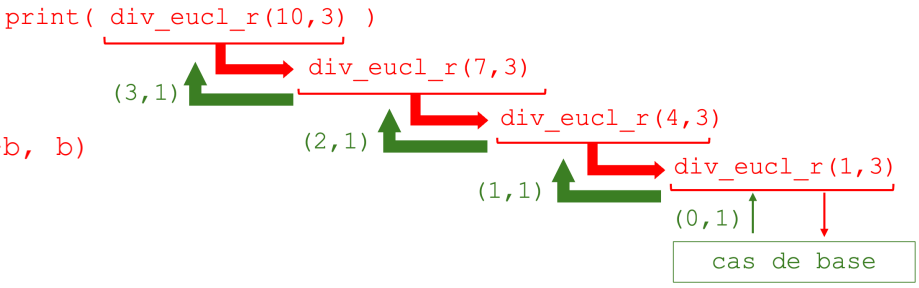
↦ Version **ITERATIVE**

```
def div_eucl(a, b):   # a et b entiers, a ≥ b, ( a = b * q + r )
    q, r = 0, a       # initialisation q et r
    while r >= b :    # à la sortie de boucle, on aura r < b
        r -= b        # r, initialisé à a, est décrémenté de b à chaque étape
        q += 1        # et q est donc un compteur du nombre de fois qu'on a décrémenté a
    return q,r        # quotient, reste
```

```
print(div_eucl(9, 3))     # affiche (3,0)
print(div_eucl(10, 3))    # affiche (3,1)
print(div_eucl(11, 3))    # affiche (3,2)
print(div_eucl(12, 3))    # affiche (4,0)
```

↦ Version **RECURSIVE**

```
def div_eucl_r(a, b):  # ex : print(div_eucl_r(10,3))
    if a < b :
        return 0, a
    else :
        q, r = div_eucl_r(a-b, b)
        return q+1, r
```

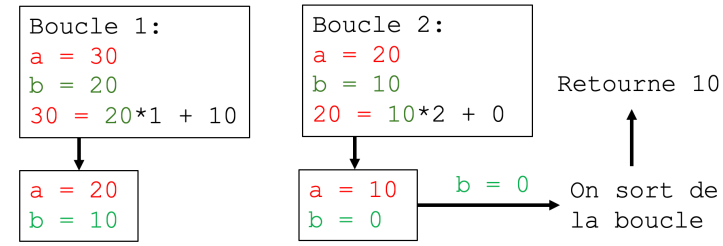


C.2 / Algorithme d'Euclide

Il s'agit de déterminer le PGCD entre deux entiers *a* et *b*. On rappelle (découverte d'Euclide) que PGCD (a,b) = PGCD (b,r). Voici la version **ITERATIVE**.

↦ Version **ITERATIVE**

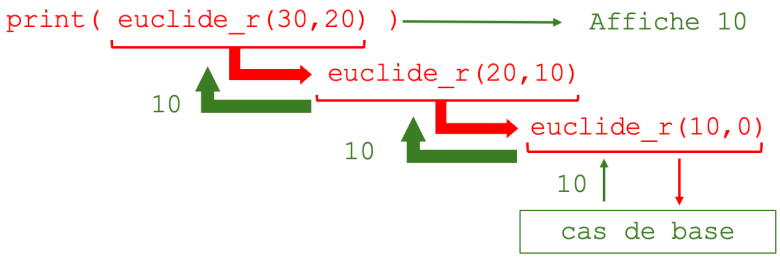
```
def euclide(a, b):          # a et b entiers, a ≥ b
    while b != 0 :          # ou tout simplement : while b
        a, b = b, a%b       # ou a, b = b, reste de div_eucl(a,b)
    return a

print(euclide(30, 20))     # affiche 10 →
print(euclide(30, 15))     # affiche 15
print(euclide(30, 14))     # affiche 2
print(euclide(30, 13))     # affiche 1
```



↦ Version **RECURSIVE**

Réécrire une fonction faisant la même chose, mais récursivement. On prendra pour cas de base que le PGCD entre tout nombre entier *a* et *0* est *a* lui-même. *a* est en effet multiple de lui-même, et de 0 puisque le reste de la division Euclidienne est nul. Version **RECURSIVE.**

```
def euclide_r(a, b):  # ex : print(euclide_r(30,20))
    if b == 0 :
        return a
    else :
        return euclide_r(b, a%b)
```



C.3 / Recherche par dichotomie d'un élément dans une liste triée.

Algorithme de recherche dichotomique de l'index (position) d'un élément e dans une liste triée L.

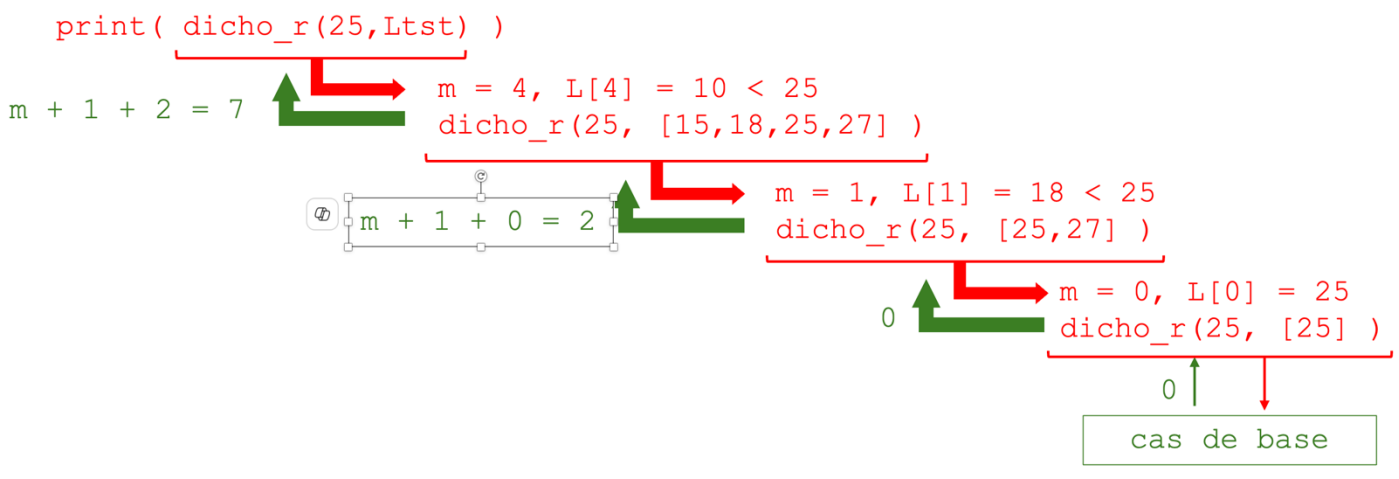
↦ Version **ITERATIVE**

```
def dicho(e, L):
    a, b = 0, len(L)     # On cherche sur l'intervalle [L[a], … L[b]]
    m = (a+b)//2
    while a < b :
        if e <= L[m] :     # on cherche sur l'intervalle [L[a],... L[m]]
            a, b = a, m-1
        else :             # on cherche sur l'intervalle [L[m+1],... L[b]]
            a, b = m+1, b
        m = (a+b)//2
    return m

Ltst = [1,5,7,9,10,15,18,25,27]
print(dicho(1, Ltst))     # affiche 0
print(dicho(10, Ltst))    # affiche 4
print(dicho(27, Ltst))    # affiche 8
```

↦ Version **RECURSIVE**

Réécrire une fonction faisant la même chose, mais récursivement.

```
def dicho_r(e, L):     # trouver l'index de e dans L sachant que e appartient à L
    if len(L) == 1 :   # cas de base : la liste ne compte qu'un seul élément
        return 0
    else :
        m = (len(L)-1) // 2
        if e <= L[m] :           # on cherche à gauche, sur [L[0], ..., L[m]]
            return dicho_r(e, L[:m+1])
        else :                   # on cherche à droite, sur [L[m+1], ..., L[N-1]]
            return m + 1 + dicho_r(e, L[m+1:])
                                 # Attention au décalage d'indice

print(dicho_r(25, Ltst))                    # Ltst = [1,5,7,9,10,15,18,25,27]
```



D / Exemples supplémentaires

D.1 / Calcul de la factorielle

On rappelle que $n! \equiv n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1$ avec $0! \equiv 1$.

↦ Version **ITERATIVE**

```
def factorielle(n):
    resultat = 1
    for i in range(1,n+1):
        resultat *= i
    return resultat
```

↦ Version **RECURSIVE**

Cas de base, cas le plus simple.

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

La fonction s'auto appelle pour une entrée plus simple.

Il est possible de calculer la factorielle par une méthode itérative avec une boucle for :

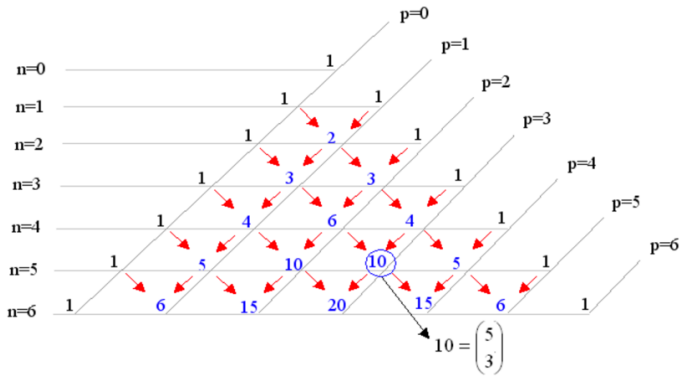
D.2 / Le calcul des coefficients binomiaux

La fonction $C(n,p)$ donne le nombre de façon de choisir p (un entier) objets parmi n (un entier) avec évidemment $p \leq n$. Vous avez déjà dû rencontrer les relations suivantes :

$$C(n,0) = C(n,n) = 1$$

$$C(n,p) = C(n-1,p-1) + C(n-1,p) \text{ (triangle de Pascal, cf. figure ci-dessous)}$$

↦ Version **RECURSIVE**



La fonction s'auto appelle (2 fois) pour deux entrées plus simples.

```
def C(n,p):
    if (n>p) and (p>0):
        return C(n-1,p-1)+C(n-1,p)
    else:
        return 1
```

Cas de base, cas le plus simple.

Le calcul de $C(n,p)$ par récursivité semble magique puisque que cela ne nécessite jamais l'utilisation de la relation bien connue $C(n,p) = \frac{n!}{(n-p)!p!}$. On utilise encore une fois le lien entre le cas à entrée plus complexe et les deux cas avec des entrées plus simples: $C(n,p) = C(n-1,p-1) + C(n-1,p)$.

E / Efficacité de la récursivité, exemple de la suite de Fibonacci

La récursivité peut être un outil puissant pour la mise en œuvre d'algorithme complexe comme vous le verrez dans votre cours d'informatique. Mais d'un autre côté, la récursivité peut conduire à un algorithme de piètre performance en terme de temps de calcul. Nous allons entrevoir une partie de ce problème sur l'exemple de la fameuse suite de Fibonacci. Cette dernière est définie de la façon suivante :

$$f_1 = f_2 = 1$$
$$f_n = f_{n-1} + f_{n-2}$$

La valeur de chaque terme est la somme des deux précédents. Les dix premiers termes de la suite sont :

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

Les termes de la suite sont très faciles à calculer ; l'entrée suivante est 34+55=89.

↦ Version **RECURSIVE**

Voici un programme qui calcule les termes de la suite par récursivité et les résultats pour les 35 premiers termes :

```
1  ##
2  #  This program computes Fibonacci numbers using a recursive function.
3  #
4
5  def main() :
6     n = int(input("Enter n: "))
7     for i in range(1, n + 1) :
8        f = fib(i)
9        print("fib(%d) = %d" % (i, f))
10
11 ## Computes a Fibonacci number.
12 #  @param n an integer
13 #  @return the nth Fibonacci number
14 #
15 def fib(n) :
16    if n <= 2 :
17       return 1
18    else :
19       return fib(n - 1) + fib(n - 2)
20
21 # Start the program.
22 main()
```

Cas de base, cas le plus simple.

La fonction s'auto appelle (2 fois) pour deux entrées plus simples.

Program Run

```
Enter n: 35
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
fib(6) = 8
fib(7) = 13
. . .
fib(35) = 9227465
```

Le programme semble fonctionner correctement. Cependant vous pouvez constater qu'en exécutant le programme, le temps de calcul des termes augmente de façon dramatique avec n. Cela n'a pas de sens, il est plus rapide de calculer les termes avec un crayon et une calculatrice !!

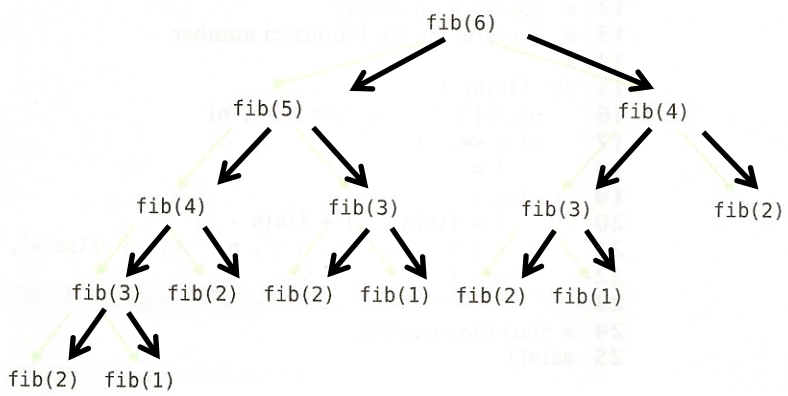
Pour comprendre ce qu'il se passe, le programme suivant « print » chaque étape pour avoir une trace de ce que fait le programme :

```
6  def main() :
7     n = int(input("Enter n: "))
8     f = fib(n)
9     print("fib(%d) = %d" % (n, f))
10
11 ## Computes a Fibonacci number.
12 #  @param n an integer
13 #  @return the nth Fibonacci number
14 #
15 def fib(n) :
16    print("Entering fib: n =", n)
17    if n <= 2 :
18       f = 1
19    else :
20       f = fib(n - 1) + fib(n - 2)
21    print("Exiting fib: n =", n, "return value =", f)
22    return f
23
24 # Start the program.
25 main()
```

Program Run

```
Enter n: 6
Entering fib: n = 6
Entering fib: n = 5
Entering fib: n = 4
Entering fib: n = 3
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Entering fib: n = 1
Exiting fib: n = 1 return value = 1
Exiting fib: n = 3 return value = 2
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Exiting fib: n = 4 return value = 3
Entering fib: n = 3
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Entering fib: n = 1
Exiting fib: n = 1 return value = 1
Exiting fib: n = 3 return value = 2
Exiting fib: n = 5 return value = 5
Entering fib: n = 4
Entering fib: n = 3
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Entering fib: n = 1
Exiting fib: n = 1 return value = 1
Exiting fib: n = 3 return value = 2
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Exiting fib: n = 4 return value = 3
Exiting fib: n = 6 return value = 8
fib(6) = 8
```

On peut aussi représenter les appels de fonction dans un arbre.



On voit que ce n'est pas efficace : par exemple, fib(3) est appelé trois fois, ce qui est une perte de temps. De plus on imagine bien que l'arbre va devenir très vite gigantesque, avec un très grand nombre d'appels inutiles. Cela est très différent d'un calcul avec un crayon et un papier où chaque terme n'est calculé qu'une fois.

↦ Version **ITERATIVE**

```
1  ##
2  #  This program computes Fibonacci numbers using an iterative function.
3  #
4
5  def main() :
6     n = int(input("Enter n: "))
7     for i in range(1, n + 1) :
8        f = fib(i)
9        print("fib(%d) = %d" % (i, f))
10
11 ## Computes a Fibonacci number.
12 #  @param n an integer
13 #  @return the nth Fibonacci number
14 #
15 def fib(n) :
16    if n <= 2 :
17       return 1
18    else :
19       olderValue = 1
20       oldValue = 1
21       newValue = 1
22       for i in range(3, n + 1) :
23          newValue = oldValue + olderValue
24          olderValue = oldValue
25          oldValue = newValue
26
27       return newValue
28
29 # Start the program.
30 main()
```

Program Run

```
Enter n: 50
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
fib(6) = 8
fib(7) = 13
. . .
fib(50) = 12586269025
```

Vous pouvez constater que ce programme est beaucoup plus rapide.
Le tableau ci-après donne les ordres de grandeur du temps de calcul du programme par itération et du programme par **récursivité**.

	Fonction récursive (fib1)	Fonction itérative (fib2)
k	Temps (en secondes)	Temps (en secondes)
10	$7 \cdot 10^{-5}$	$3 \cdot 10^{-6}$
20	$3 \cdot 10^{-3}$	$3 \cdot 10^{-6}$
30	0.5	$3 \cdot 10^{-6}$
40	58	$4 \cdot 10^{-6}$
50	7264	$4 \cdot 10^{-6}$

Pour conclure, on peut retenir que, de façon occasionnelle, la solution par récursivité est beaucoup plus lente que la solution par itération. Cependant, dans la plupart des cas, la récursivité est simplement un peu plus lente que l'itération. Mais, dans de très nombreux cas, la solution par récursivité est plus simple à comprendre et à mettre en œuvre qu'une solution par itération.
Selon l'informaticien (et créateur de l'interpréteur GhostScript pour le langage graphique PostScript) L. Peter Deutsh :

« L'itération est humaine, la récursivité est divine »