Initiation à l'algorithmique

— procédures et fonctions —
2. Appel d'une fonction

Jacques TISSEAU

Enib–Cerv

Remarque (Notes de cours : couverture)

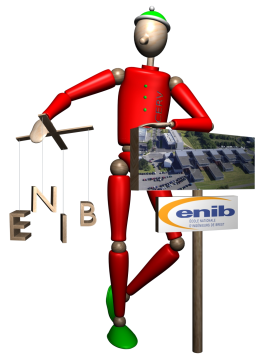
Ce support de cours accompagne le chapitre 3 des notes de cours « Initiation à l'algorithmique ».

— Cours d'Informatique S1 —

Initiation à l'algorithmique

Jacques TISSEAU

Ecole nationale d'ingénieurs de Brest
Centre européen de réalité virtuelle
tisseau@enib.fr



Ces notes de cours accompagnent les enseignements d'informatique du 1er semestre (S1) de l'Ecole Nationale d'Ingénieurs de Brest (Enib : www.enib.fr). Leur lecture ne dispense en aucun cas d'une présence attentive aux cours ni d'une participation active aux travaux dirigés.

Avec la participation de Romain Bénard, Stéphane Bonneaud, Cédric Buche, Gireg Desmeulles, Céline Jost, Sébastien Kubicki, Eric Maisel, Aléxis Nédélec, Marc Parenthoën et Cyril Septseault.

version du 21 octobre 2014

```python
def fibonacci(n):
    """
    u = fibonacci(n)
    est le nombre de Fibonacci
    a l'ordre n si n:int >= 0
    """
    assert type(n) is int
    assert n >= 0
    u, u1, u2 = 1, 1, 1
    for i in range(2,n+1):
        u = u1 + u2
        u2 = u1
        u1 = u
    return u
```


Remarque (Fonction de Fibonacci)

La fonction de Fibonacci calcule le nombre u_n à l'ordre n (dit de Fibonacci) selon la relation de récurrence :

$$u_0 = 1 \ , \ u_1 = 1 \ , \ u_n = u_{n-1} + u_{n-2} \ \forall n \in N, \ n > 1$$

Les 10 premiers nombres de Fibonacci valent donc : $u_0 = 1$, $u_1 = 1$, $u_2 = 2$, $u_3 = 3$, $u_4 = 5$, $u_5 = 8$, $u_6 = 13$, $u_7 = 21$, $u_8 = 34$ et $u_9 = 55$.
La suite de Fibonacci doit son nom au mathématicien italien Fibonacci (1175-1250). Dans un problème récréatif, Fibonacci décrit la croissance d'une population « idéale » de lapins de la manière suivante :

- *le premier mois, il y a juste une paire de lapereaux ;*
- *les lapereaux ne sont pubères qu'à partir du deuxième mois ;*
- *chaque mois, tout couple susceptible de procréer engendre effectivement un nouveau couple de lapereaux ;*
- *les lapins ne meurent jamais !*

Se pose alors le problème suivant :

> « Possédant initialement un couple de lapins, combien de couples obtient-on en douze mois si chaque couple engendre tous les mois un nouveau couple à compter du second mois de son existence ? »

APPEL DE FONCTION

PASSAGE DES PARAMÈTRES

Définition

```
def fibonacci(n):
    ...

    return u
```

paramètres formels

Appel

```
>>> x = ...
>>> y = fibonacci(x)
```

paramètres effectifs

à l'appel : copie des paramètres effectifs dans les paramètres formels (`n = x`)

à la sortie : copie des paramètres formels dans les paramètres effectifs (`y = u`)

Passage par valeur : ce ne sont pas les paramètres effectifs qui sont manipulés par la fonction mais des copies de ces paramètres

Définitions

paramètre formel paramètre d'entrée d'une fonction utilisé à l'intérieur de la fonction appelée.

paramètre effectif paramètre d'appel d'une fonction utilisé à l'extérieur de la fonction appelée.

passage par valeur action de copier la valeur du paramètre effectif dans le paramètre formel correspondant.

passage par référence action de copier la référence du paramètre effectif dans le paramètre formel correspondant.

APPEL DE FONCTION

APPEL ÉQUIVALENT

Appel :

```
>>> x = 9
>>> y = fibonacci(x)
>>> y
55
```

Appel équivalent :

```
>>> x = 9
>>> n = x
>>> u, u1, u2 = 1, 1, 1
>>> for i in range(2,n+1):
...     u = u1 + u2
...     u2 = u1
...     u1 = u
...
>>> tmp = u
>>> del n, u, u1, u2, i
>>> y = tmp
>>> del tmp
>>> y
55
```


TD (Passage par valeur)

On considère les codes suivants :

```
>>> x, y
(1, 2)
>>> tmp = x
>>> x = y
>>> y = tmp
>>> x, y
(2, 1)
```

```
def swap(x,y):
    tmp = x
    x = y
    y = tmp
    return
```

```
>>> x, y
(1, 2)
>>> swap(x,y)
>>> x, y
(1, 2)
```

Expliquer la différence entre l'exécution de gauche et l'exécution de droite en explicitant l'appel équivalent à l'appel `swap(x,y)` *dans l'exécution de droite.*

APPEL DE FONCTION

PORTÉE DES VARIABLES

```
def f(x):
    y = 3
    x = x + y
    print('liste :', dir())
    print('intérieur :', locals())
    print('extérieur :', globals())
    return x

>>> y = 6
>>> f(6)
liste : ['x', 'y']
intérieur : {'y': 3, 'x': 9}
extérieur : {'f': <function f at 0x822841c>,
             'y': 6,
             '__name__': '__main__',
             '__doc__': None}
9
```


TD (Portée des variables)

On considère les fonctions f, g *et* h *suivantes :*

```
def f(x):
  x = 2*x
  print('f', x)
  return x
```

```
def g(x):
  x = 2*f(x)
  print('g', x)
  return x
```

```
def h(x):
  x = 2*g(f(x))
  print('h', x)
  return x
```

Qu'affichent les appels suivants ?

1.
```
>>> x = 5
>>> print(x)
?
>>> y = f(x)
>>> print(x)
?
>>> z = g(x)
>>> print(x)
?
>>> t = h(x)
>>> print(x)
?
```

2.
```
>>> x = 5
>>> print(x)
?
>>> x = f(x)
>>> print(x)
?
>>> x = g(x)
>>> print(x)
?
>>> x = h(x)
>>> print(x)
?
```

APPELS RÉCURSIFS
FONCTION DE FIBONACCI

$$u_0 = 1\ ,\ u_1 = 1\ ,\ u_n = u_{n-1} + u_{n-2}\ \forall n \in N,\ n > 1$$

Version itérative :

```
def fibonacci(n):
    u, u1, u2 = 1, 1, 1
    for i in range(2,n+1):
        u = u1 + u2
        u2 = u1
        u1 = u
    return u
```

Version récursive :

```
def fibonacci(n):
    u = 1
    if n > 1:
        u = fibonacci(n-1) +
            fibonacci(n-2)
    return u
```


TD (Puissance entière)

Définir une fonction récursive qui calcule la puissance entière $p = x^n$ d'un nombre entier x.

TD (Coefficients du binôme)

Définir une fonction récursive qui calcule les coefficients du binôme

$$(a+b)^n = \sum_{k=0}^{n} \frac{n!}{k!(n-k)!} a^{n-k} b^k.$$

TD (Fonction d'Ackerman)

Définir une fonction récursive qui calcule la fonction d'Ackerman :

$$f : N^2 \rightarrow N \left\{ \begin{array}{lcl} f(0,n) & = & n+1 \\ f(m,0) & = & f(m-1,1)\ si\ m > 0 \\ f(m,n) & = & f(m-1, f(m,n-1))\ si\ m > 0, n > 0 \end{array} \right.$$

APPELS RÉCURSIFS

FONCTION DE FIBONACCI

```
>>> fibonacci(5)
8
```

```
fibonacci(5)
 ├─fibonacci(4)
 │  ├─fibonacci(3)
 │  │  ├─fibonacci(2)
 │  │  │  ├─fibonacci(1)
 │  │  │  └─fibonacci(0)
 │  │  └─fibonacci(1)
 │  └─fibonacci(2)
 │     ├─fibonacci(1)
 │     └─fibonacci(0)
 └─fibonacci(3)
    ├─fibonacci(2)
    │  ├─fibonacci(1)
    │  └─fibonacci(0)
    └─fibonacci(1)
```

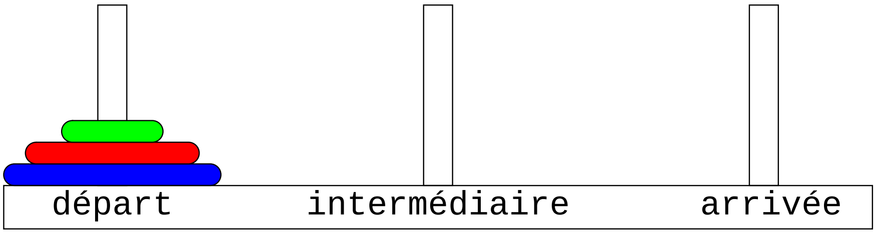

Remarque (Récursivité en arbre : `fibonacci`)

Dans la version récursive, pour calculer `fibonacci(5)`*, on calcule d'abord* `fibonacci(4)` *et* `fibonacci(3)`*. Pour calculer* `fibonacci(4)`*, on calcule* `fibonacci(3)` *et* `fibonacci(2)`*. Pour calculer* `fibonacci(3)`*, on calcule* `fibonacci(2)` *et* `fibonacci(1)`*... Le déroulement du processus ressemble ainsi à un arbre*

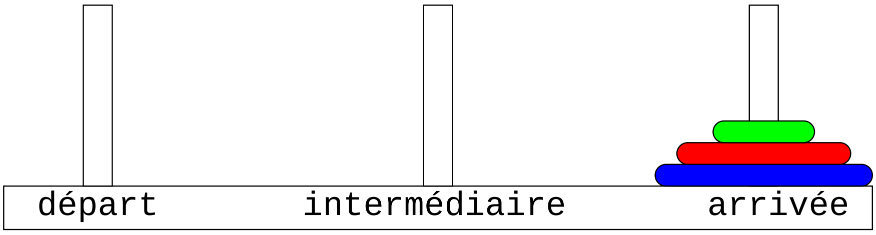
On remarque que les branches de l'arbre se divise en deux à chaque niveau (sauf en bas de l'arbre, ie *à droite sur la figure), ce qui traduit le fait que la fonction* `fibonacci` *s'appelle elle-même deux fois à chaque fois qu'elle est invoquée avec* `n > 1`*.*

Le nombre de feuilles dans l'arbre est précisément u_n (`fibonacci(n)`*). Or la valeur de u_n croît de manière exponentielle avec n ; ainsi, avec cette version récursive, le processus de calcul de* `fibonacci(n)` *prend un temps qui croît de façon exponentielle avec* `n`*.*

Etat initial :



Etat final :



Remarque (Tours de Hanoï)

Les « tours de Hanoï » est un jeu imaginé par le mathématicien français Édouard Lucas (1842-1891). Il consiste à déplacer n disques de diamètres différents d'une tour de « départ » à une tour d'« arrivée » en passant par une tour « intermédiaire » et ceci en un minimum de coups, tout en respectant les règles suivantes :

- *on ne peut déplacer qu'un disque à la fois,*
- *on ne peut placer un disque que sur un autre disque plus grand que lui ou sur une tour vide.*

TD (Tours de Hanoï *à la main*)

Résoudre à la main *le problème des tours de Hanoï à n disques successivement pour* $n = 1$, $n = 2$, $n = 3$ *et* $n = 4$.

APPELS RÉCURSIFS

TOURS DE HANOÏ

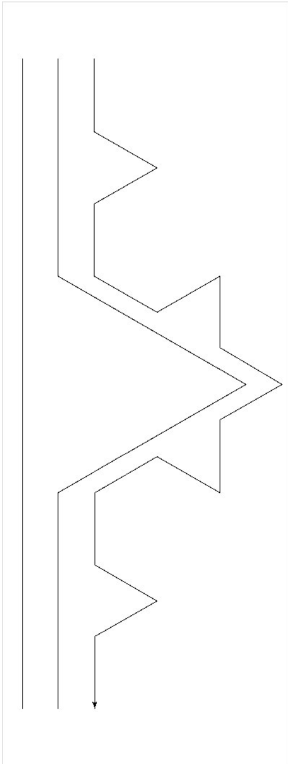
```
def hanoi(n,gauche,milieu,droit):
    assert type(n) is int
    assert n >= 0
    if n > 0:
        hanoi(n-1,gauche,droit,milieu)
        deplacer(n,gauche,droit)
        hanoi(n-1,milieu,droit,gauche)
    return

def deplacer(n,gauche,droit):
    print('déplacer disque', n,
          'de la tour', gauche,
          'à la tour', droit)
    return
```

```
>>> hanoi(3,'d','i','a')
```

```
hanoi(3,'d','i','a')
├─ hanoi(2,'d','a','i')
│   ├─ hanoi(1,'d','i','a')
│   │   ├─ hanoi(0,'d','a','i')
│   │   ├─ deplacer(1,'d','a')        1
│   │   └─ hanoi(0,'i','d','a')
│   ├─ deplacer(2,'d','i')            2
│   └─ hanoi(1,'a','d','i')
│       ├─ hanoi(0,'a','i','d')
│       ├─ deplacer(1,'a','i')        3
│       └─ hanoi(0,'d','a','i')
├─ deplacer(3,'d','a')                4
└─ hanoi(2,'i','d','a')
    ├─ hanoi(1,'i','a','d')
    │   ├─ hanoi(0,'i','d','a')
    │   ├─ deplacer(1,'i','d')        5
    │   └─ hanoi(0,'a','i','d')
    ├─ deplacer(2,'i','a')            6
    └─ hanoi(1,'d','i','a')
        ├─ hanoi(0,'d','a','i')
        ├─ deplacer(1,'d','a')        7
        └─ hanoi(0,'i','d','a')
```


Remarque (Récursivité en arbre : `hanoi`)

L'exécution d'un appel à la procédure `hanoi` *s'apparente ici encore à un processus récursif en arbre : les 7 déplacements effectués lors de l'appel* `hanoi(3,'d','i','a')` *sont numérotés dans leur ordre d'apparition sur la figure (les appels à la fonction* `hanoi` *pour* `n = 0` *ne font rien).*
Mais toutes les fonctions récursives ne conduisent pas nécessairement à un processus récursif en arbre comme l'exemple de la fonction `factorielle` *le montrera.*



```
def kock(n,d):
    if n == 0: forward(d)
    else:
        kock(n-1,d/3.)
        left(60)
        kock(n-1,d/3.)
        right(120)
        kock(n-1,d/3.)
        left(60)
        kock(n-1,d/3.)
    return
```

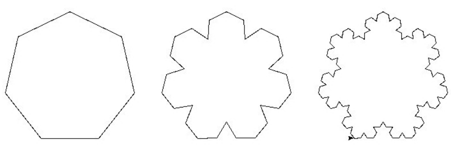

Remarque (Courbes de Koch)

La courbe de von Koch est l'une des premières courbes fractales à avoir été décrite par le mathématicien suédois Helge von Koch (1870-1924). On peut la créer à partir d'un segment de droite, en modifiant récursivement chaque segment de droite de la façon suivante :

1. *on divise le segment de droite en trois segments de longueurs égales,*
2. *on construit un triangle équilatéral ayant pour base le segment médian de la première étape,*
3. *on supprime le segment de droite qui était la base du triangle de la deuxième étape.*

TD (Flocons de Koch)

Définir une fonction qui dessine les flocons de Koch heptagonaux ci-dessous. Généraliser à des polygones réguliers quelconques (triangle équilatéral, carré, pentagone régulier. . .).



APPELS RÉCURSIFS

FONCTION FACTORIELLE

$$\begin{cases} 0! = 1 \\ n! = n \cdot (n-1)! \quad \forall n \in N^* \end{cases}$$

Version itérative :

```
def factorielle(n):
    u = 1
    for i in range(2,n+1):
        u = u * i
    return u
```

Version récursive :

```
def factorielle(n):
    u = 1
    if n > 1:
        u = n * factorielle(n-1)
    return u
```


TD (Pgcd et ppcm de 2 entiers)

1. *Définir une fonction récursive qui calcule le plus grand commun diviseur d de 2 entiers a et b :*
   $\mathrm{pgcd}(a, b) = \mathrm{pgcd}(b, a \bmod b) = \ldots\ldots = \mathrm{pgcd}(d, 0) = d.$
2. *En déduire une fonction qui calcule le plus petit commun multiple m de 2 entiers a et b.*

TD (Somme arithmétique)

1. *Définir une fonction récursive qui calcule la somme des n premiers nombres entiers.*
   $$s = \sum_{k=0}^{n} k = \frac{n(n+1)}{2}$$
2. *Comparer la complexité de cette version avec les versions constante et itérative.*

```
>> factorielle(5)
        (5*factorielle(4))
        (5*(4*factorielle(3)))
        (5*(4*(3*factorielle(2))))
        (5*(4*(3*(2*factorielle(1)))))
        (5*(4*(3*(2*1))))
        (5*(4*(3*2)))
        (5*(4*6))
        (5*24)
120
```


Remarque (Récursivité linéaire : `factorielle`)

La version récursive est la traduction directe de la formulation mathématique. Dans la version récursive, le processus nécessite que l'interpréteur garde une trace des multiplications à réaliser plus tard. Le processus croît puis décroît : la croissance se produit lorsque le processus construit une chaîne d'opérations différées (ici, une chaîne de multiplications différées) et la décroissance intervient lorsqu'on peut évaluer les multiplications.
Ainsi, la quantité d'information qu'il faut mémoriser pour effectuer plus tard les opérations différées croît linéairement avec n : on parle de processus récursif linéaire.

APPELS RÉCURSIFS

FONCTION FACTORIELLE

```
def factorielle(n):
    u = factIter(n,1,1)
    return u

def factIter(n,i,fact):
    u = fact
    if i < n:
        u = factIter(n,i+1,fact*(i+1))
    return u
```

```
>>> factorielle(5)
        (factIter(5,1,1))
        (factIter(5,2,2))
        (factIter(5,3,6))
        (factIter(5,4,24))
        (factIter(5,5,120))
120
```


Définitions

récursivité terminale Un appel récursif terminal est un appel récursif dont le résultat est celui retourné par la fonction.

récursivité non terminale Un appel récursif non terminal est un appel récursif dont le résultat n'est pas celui retourné par la fonction.

Remarque

La nouvelle fonction `factorielle` *appelle une fonction auxiliaire* `factIter` *dont la définition est syntaxiquement récursive (*`factIter` *s'appelle elle-même). Cette fonction à 3 arguments : l'entier* `n` *dont il faut calculer la factorielle, un compteur* `i` *initialisé à* `1` *au premier appel de* `factIter` *par* `factorielle` *et incrémenté à chaque nouvel appel, et un nombre* `fact` *initialisé à* `1` *et multiplié par la nouvelle valeur du compteur à chaque nouvel appel.*
Le déroulement d'un appel à `factIter` *montre qu'ainsi, à chaque étape, la relation* `(i! == fact)` *est toujours vérifiée. La fonction* `factIter` *arrête de s'appeler elle-même lorsque* `(i == n)` *et on a alors* `(fact == i! == n!)` *qui est la valeur recherchée.*
Ainsi, à chaque étape, nous n'avons besoin que des valeurs courantes du compteur `i` *et du produit* `fact`*, exactement comme dans la version itérative de la fonction* `factorielle` *: il n'y a plus de chaîne d'opérations différées comme dans la version récursive de* `factorielle`*. Le processus mis en jeu ici est un processus itératif, bien que la définition de* `factIter` *soit récursive.*

APPELS RÉCURSIFS

RÉCURSIVITÉ TERMINALE → BOUCLE

```
def f(x):
    if cond: arret
    else:
        instructions
        f(g(x))
    return
```

⇓

```
def f(x):
    while not cond:
        instructions
        x = g(x)
    arret
    return
```

```
def factIter(n,i,fact):
    if i >= n: u = fact
    else:
        pass
        u = factIter(n,i+1,fact*(i+1))
    return u
```

⇓

```
def factIter(n,i,fact):
    while i < n:
        pass
        n,i,fact = n,i+1,fact*(i+1)
    u = fact
    return u
```


Remarque (Elimination de la récursivité)

La méthode précédente ne s'applique qu'à la récursivité terminale.
Une méthode générale existe pour transformer une fonction récursive quelconque en une fonction itérative équivalente. En particulier, elle est mise en œuvre dans les compilateurs car le langage machine n'admet pas la récursivité.
Cette méthode générale fait appel à la notion de pile pour sauvegarder le contexte des appels récursifs.

TD (Pgcd)

Transformer la fonction récursive ci-dessous en une fonction itérative.

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