Algorithmique et Programmation 2

2. Fonctions, récursivité et générateurs

Lionel.Moisan@parisdescartes.fr

Université Paris Descartes
http://www.mi.parisdescartes.fr/∽moisan/

Fonctions

Valeurs de retour d'une fonction

En informatique, on distingue les fonctions (qui renvoient une ou plusieurs valeurs) des procédures (qui ne renvoient rien)

Par abus de langage, on englobe parfois les procédures dans les fonctions, car dans beaucoup de langages la définition est très proche

En python, les procédures sont en réalité des fonctions qui renvoient la valeur `None` (seule valeur du type `NoneType`)

```
>>> def f(s): # procédure
...     print('Hello '+s)
>>> f('Bill')
Hello Bill
>>> y = f('Bill')
Hello Bill
>>> print(y)
None
>>> type(y)
<class 'NoneType'>
>>> y is None # expression dédiée (plutôt que y==None)
True
```

Valeurs de retour d'une fonction

Les trois fonctions suivantes sont stritement équivalentes :

```
def f(x):
    print(x)
```

```
def g(x):
    print(x)
    return
```

```
def h(x):
    print(x)
    return None
```

Néanmoins, l'usage n'est pas le même : on utilisera plutôt `return` (ou rien) dans le cas d'une procédure, et `return None` pour une fonction qui, en raison d'un cas particulier, souhaite retourner la valeur `None`

Exemple :

```
def premier_degré(a,b): # résout l'équation ax = b
    if a==0:
        return None # pas de solution
    return b/a
>>> print(premier_degré(5,3))
0.6
>>> print(premier_degré(0,3))
None
```

Valeurs de retour d'une fonction

Comme nous l'avons vu précédemment, une fonction Python peut retourner plusieurs valeurs à l'aide d'un tuple

Exemple :

```
>>> def min2(L):
...     """ renvoie m,i
...     m est la valeur minimale de L
...     i est le premier indice tel que L[i] == m
...     """
...     assert len(L)>0 # précondition: L non vide
...     for j in range(len(L)):
...         if j==0 or L[j]<m:
...             m,i = L[j],j
...     return m,i
...
>>> a,b = min2([1.1, 0.2, 3., 4.5, 0.2, 5.9, 0.2])
>>> a
0.2
>>> b
1
```

Arguments optionnels

On peut spécifier des arguments optionnels pour une fonction Python, en leur donnant des valeurs par défaut

```
def impression(fichier, taille='A4', couleur=True):
    ...
```

Lors de l'appel de la fonction, les arguments optionnels peuvent alors être spécifiés de manière positionnelle :

```
>>> impression('document.pdf')
>>> impression('document.pdf', 'A3')
>>> impression('document.pdf', 'A4', False)
```

ou alors par mot-clé (nom utilisé dans la déclaration de la fonction) :

```
>>> impression('document.pdf', couleur=False)
>>> impression(fichier='document.pdf', taille='A3')
```

Arguments optionnels

Attention, pas d'argument non-nommé après un argument nommé :

```
>>> impression(fichier='document.pdf', 'A3')
  File "<stdin>", line 1
SyntaxError: non-keyword arg after keyword arg
```

De même, dans la déclaration de la fonction, les arguments optionnels viennent toujours après les arguments non optionnels

```
def impression(taille='A4', couleur=True, fichier):
    ...
  File "<stdin>", line 1
SyntaxError: non-default argument follows default argument
```

Déballage d'un dictionnaire avec la double étoile (**)

Revenons sur l'exemple précédent :

```
# définition de la fonction
def impression(fichier, taille='A4', couleur=True):
    ...
# appel de la fonction
impression(fichier='document.pdf', couleur=False)
```

La spécification des arguments par mots-clés rappelle les dictionnaires !

Ce n'est pas un hasard : les arguments des fonctions sont en effets gérés par Python comme des dictionnaires, et la double étoile (**) permet le déballage d'un dictionnaire pour spécifier les arguments d'une fonction :

```
impression(fichier='document.pdf', couleur=False)
```

est équivalent à

```
d = dict(fichier='document.pdf', couleur=False)
impression(**d)
```

Les fonctions sont des objets Python

```
>>> def f(x):
...     return x**2

>>> type(f)
<class 'function'>
```

Les fonctions sont des objets Python, et à ce titre peuvent être passés en argument à d'autres fonctions :

```
>>> def controle(f,x):
...     y = f(x)
...     print('entrée = '+str(x)+' sortie = '+str(y))

>>> controle(f,7)
entrée = 7 sortie = 49
```

Fonctions anonymes

Il est parfois commode de définir une fonction très simple à la volée, sans avoir recours à l'instruction `def`.

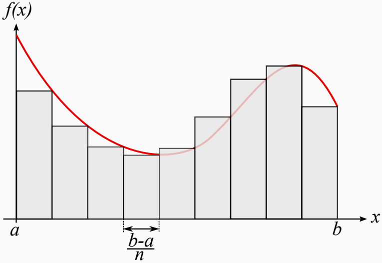
Pour cela, on dispose de l'instruction `lambda` (le nom 'lambda' vient d'une théorie informatique appelée *lambda-calcul*) :

```
f = lambda x: x**2
```

est équivalent à

```
def f(x):
    return x**2
```

Application : estimation numérique d'une intégrale



Théorème : Si $f : [a, b] \to \mathbb{R}$ est continue, alors

$$S_n = \frac{b-a}{n} \sum_{k=1}^{n} f\left(a + k\frac{b-a}{n}\right)$$

converge vers $\int_a^b f(x)\,dx$ lorsque $n \to +\infty$ (méthode des rectangles)

Implémentation du calcul de S_n à partir de f, a, b et n :

```
def rectangles(f,a,b,n=1000):
    return sum([f(a+k*(b-a)/n) for k in range(1,n+1)])*(b-a)/n
```

Application : estimation numérique d'une intégrale

Vérification : calcul de $\displaystyle\int_0^{\pi} \sin(t)\,dt = \Big[-\cos(t)\Big]_0^{\pi} = 2$

```
>>> from math import sin,pi
>>> rectangles(sin,0,pi) # argument n non précisé
1.9999983550656624
```

Estimation numérique de la constante de Wilbraham-Gibbs,

$$C = \int_0^{\pi} \frac{\sin x}{x}\,dx \simeq 1,85...$$

```
>>> rectangles(lambda x:sin(x)/x,0,pi,1000000)
1.8519354811859796
```

Récursivité

Principe d'une fonction récursive

fonction récursive

Une fonction est dite récursive si elle s'appelle elle-même.

La plupart des langages de programmation permettent de définir des fonctions récursives. C'est un cas bien spécifique, dans la mesure où la fonction en train d'être définie (donc pas encore définie) va apparaître dans sa propre définition.

Exemple : calcul de $n! = n \times (n-1) \times \ldots \times 2 \times 1$

```
def fact(n):
    """ calcul de n! """
    if n==0:
        return 1 # par convention, 0! = 1
    else:
        return n*fact(n-1)

>>> fact(4)
24
```

Fonctionnement d’un appel récursif

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

Que se passe-t-il lors de l’appel à `fact(4)` ?

`fact(4)` appelle `fact(3)`

`fact(3)` appelle `fact(2)`

`fact(2)` appelle `fact(1)`

`fact(1)` appelle `fact(0)`

`fact(0)` renvoie 1

$\rightarrow$ `fact(1)` renvoie $1 \times 1 = 1$

$\rightarrow$ `fact(2)` renvoie $2 \times 1 = 2$

$\rightarrow$ `fact(3)` renvoie $3 \times 2 = 6$

$\rightarrow$ `fact(4)` renvoie $4 \times 6 = 24$

Remarque : à chacun de ces appels, la variable locale `n` est différente !

Importance de la condition d'arrêt

Une fonction récursive s'appelant elle-même, il y a un risque qu'un appel à la fonction aboutisse à des appels récursifs sans fin

→ il est essentiel de vérifier que la condition d'arrêt est toujours remplie !

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

>>> fact(-1)
  ...
  File "<stdin>", line 2, in fact
RuntimeError: maximum recursion depth exceeded in comparison
```

Importance de la condition d'arrêt

```
def fact(n):
    assert type(n) is int and n>=0
    if n==0:
        return 1
    else:
        return n*fact(n-1)

>>> fact(4) # ok
24
>>> fact(-1) # interdit !
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in fact
AssertionError
>>> fact(1.1) # interdit !
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in fact
AssertionError
```

Profondeur de récursion maximale autorisée

```
def compte(n):
    if n==0:
        return 0
    else:
        return compte(n-1)+1
>>> compte(900)
900
>>> compte(1000)
  ...
  File "<stdin>", line 2, in compte
RuntimeError: maximum recursion depth exceeded in comparison
```

Remarque : La profondeur maximale de récursion (par défaut 1000) peut être modifiée avec `sys.setrecursionlimit()` :

```
import sys
>>> sys.getrecursionlimit()
1000
>>> sys.setrecursionlimit(2500)
>>> compte(2000)
2000
```

Calcul du terme d'indice n d'une suite récurrente

On souhaite définir une fonction `suite(u0,n)` qui renvoie le terme d'indice n de la suite définie par u_0 et la récurrence

$$\forall n \in \mathbb{N}, \quad u_{n+1} = \sqrt{1 + u_n^2}.$$

```
1 # terme d'indice n: version non récursive
2 def suite(u0,n):
3     u = u0
4     for k in range(1,n+1):
5         u = (1+u**2)**0.5
6     return u
```

tour de boucle	**variable `k`**	**variable `u`**
entrée (après la ligne 3)	—	u_0
tour 1 (après la ligne 5)	1	u_1
tour 2	2	u_2
...	...	...
tour n	`n`	u_n

Calcul du terme d'indice n d'une suite récurrente

On souhaite définir une fonction `suite(u0,n)` qui renvoie le terme d'indice n de la suite définie par u_0 et la récurrence

$$\forall n \in \mathbb{N}, \quad u_{n+1} = \sqrt{1 + u_n^2}.$$

```python
# terme d'indice n: version non récursive
def suite(u0,n):
    u = u0
    for k in range(1,n+1):
        u = (1+u**2)**0.5
    return u
```

Bien noter la différence entre l'écriture mathématique de la récurrence

$$u_{n+1} = \sqrt{1 + u_n^2}$$

et l'écriture Python (l'indice n est remplacé par le temps)

```python
u = (1+u**2)**0.5
```

Calcul du terme d'indice n d'une suite récurrente

On peut implémenter la calcul du terme d'indice n à l'aide d'une fonction récursive :

```python
# terme d'indice n: version récursive
def suite(u0,n):
    if n==0:
        return u0
    u = suite(u0,n-1) # calcul du terme d'indice n-1
    return (1+u**2)**0.5
```

Remarque : lorsque le test `n==0` est vérifié, l'instruction `return` fait quitter la fonction ; il est donc inutile de rajouter l'instruction `else` pour ce qui vient après

Calcul des termes d'indices 0 à n d'une suite récurrente

On a parfois besoin d'une fonction qui renvoie non pas le terme d'indice n de la suite, mais la liste des termes jusqu'à l'indice n

```
1 # termes d'indices 0 à n: version non récursive avec append()
2 def suite(u0,n):
3     u = [u0]
4     for k in range(1,n+1):
5         u.append( (1+u[-1]**2)**0.5 )
6     return u
```

tour de boucle	**variable** `k`	**variable** `u`
entrée (après la ligne 3)	—	$[u_0]$
tour 1 (après la ligne 5)	1	$[u_0, u_1]$
tour 2	2	$[u_0, u_1, u_2]$
...	...	...
tour n	`n`	$[u_0, u_1, \dots, u_n]$

Calcul des termes d'indices 0 à n d'une suite récurrente

Autre solution : au lieu de compléter la liste avec `u.append()`, on part d'une liste bien dimensionnée que l'on remplit peu à peu

```
1 # termes d'indices 0 à n: version non récursive par remplissage
2 def suite(u0,n):
3     u = [0]*(n+1) # liste de taille n+1 initialisée à 0
4     u[0] = u0
5     for k in range(1,n+1):
6         u[k] = (1+u[k-1]**2)**0.5
7     return u
```

tour de boucle	**variable** `k`	**variable** `u`
entrée (après la ligne 3)	—	$[0, 0, 0, \ldots, 0]$
entrée (après la ligne 4)	—	$[u_0, 0, 0, \ldots, 0]$
tour 1 (après la ligne 6)	1	$[u_0, u_1, 0, \ldots, 0]$
tour 2	2	$[u_0, u_1, u_2, 0, \ldots, 0]$
...	...	...
tour n	`n`	$[u_0, u_1, \ldots, u_n]$

Calcul des termes d'indices 0 à n d'une suite récurrente

Version non récursive (rappel)

```
# termes d'indices 0 à n: version non récursive
def suite(u0,n):
    u = [u0]
    for k in range(1,n+1):
        u.append( (1+u[-1]**2)**0.5 )
    return u
```

et version récursive :

```
# termes d'indices 0 à n: version récursive
def suite(u0,n):
    if n==0:
        return [u0]
    u = suite(u0,n-1) # termes d'indices 0 à n-1
    u.append( (1+u[-1]**2)**0.5 ) # on complète u
    return u
```

Exemple : calcul du PGCD de deux entiers

Calcul de pgcd(a,b) par l'algorithme d'Euclide ($a, b \in \mathbb{N}$) :

> si $b = 0$
> alors
> $\quad \mathrm{pgcd}(a, b) = a$
> sinon
> $\quad$ effectuer la division euclidienne de a par b :
> $\quad\quad$ a = bq+r avec $q \in \mathbb{N}$ et $0 \leq r \leq b - 1$
> $\quad \mathrm{pgcd}(a, b) = \mathrm{pgcd}(b, r)$

Remarque : l'algorithme est formulé naturellement de manière récursive

L'algorithme termine car lors des appels successifs à la fonction pgcd, le deuxième argument décroît strictement ($b \to r \leq b - 1$). Il y a donc au plus $b + 1$ appels à la fonction pgcd()

En réalité, on montre que la complexité (en nombre de divisions) de cet algorithme est $O(\log b) \to$ c'est un algorithme très efficace !

Exemple : calcul du PGCD de deux entiers

3 implémentations différentes de l'algorithme d'Euclide :

```
# version récursive
def pgcd(a,b):
    """ retourne le plus grand diviseur commun de a et b """
    if b==0:
        return a
    return pgcd(b,a%b)
    # rappel: a%b est le reste dans la division de a par b

# version non récursive
def pgcd(a,b) :
    """ retourne le plus grand diviseur commun de a et b """
    while b!=0:
        a,b  = b,a%b
    return a

# version récursive minimaliste avec lambda (pour le fun)
pgcd = lambda a,b : a if b==0 else pgcd(b,a%b)
```

Exemple : la suite de Syracuse

La suite de Collatz (mathématicien Allemand), aussi appelée suite de Syracuse (du nom de l'université du même nom aux USA), est définie par un premier terme quelconque $u_0 \in \mathbb{N}^*$ et la relation de récurrence

$$\forall n \in \mathbb{N}, \quad u_{n+1} = \begin{cases} \dfrac{u_n}{2} & \text{si } u_n \text{ est pair,} \\ 3u_n + 1 & \text{sinon.} \end{cases}$$

```python
# calcul du terme d'indice n: version non récursive
def syracuse(u0,n):
    u = u0
    for k in range(1,n+1):
        if u%2==0:
            u = u//2
        else:
            u = 3*u+1
    return u
```

Exemple : la suite de Syracuse

On peut également implémenter le calcul du terme d'indice n à l'aide d'une fonction récursive :

```
# calcul du terme d'indice n: version récursive
def syracuse(u0,n):
    if n==0: # condition d'arrêt
        return u0
    u = syracuse(u0,n-1) # calcul du terme d'indice n-1
    if u%2==0:
        return u//2
    else:
        return 3*u+1
```

Exemple : la suite de Syracuse

On peut remarquer que si un terme de la suite vaut 1, alors les termes suivants décrivent le cycle $1, 4, 2, 1, 4, 2, 1, \ldots$

La conjecture de Syracuse (ou conjecture de Collatz) prétend que pour tout entier $u_0 \in \mathbb{N}^*$, la suite finit toujours par tomber dans ce cycle

Exercice : écrire une fonction qui renvoie la liste des termes successifs jusqu'à l'obtention du premier 1

```
def syracuse_stop1(u0):
    u = [u0]
    while u[-1]!=1: # tant que le dernier terme ne vaut pas 1
        if u[-1]%2==0:
            u.append(u[-1]//2)
        else:
            u.append(3*u[-1]+1)
    return u
>>> syracuse_stop1(7)
[7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1]
```

Remarque : On ne sait pas prouver que cette fonction termine !

La suite de Fibonacci

Définition

La suite de Fibonacci est la suite $(\mathcal{F}_n)_{n\in\mathbb{N}}$ définie par $\mathcal{F}_0 = 0$, $\mathcal{F}_1 = 1$, et la récurrence d'ordre 2 suivante :

$$\forall n \in \mathbb{N}, \quad \mathcal{F}_{n+2} = \mathcal{F}_{n+1} + \mathcal{F}_n.$$

Pour calculer $\mathcal{F}_n$ de manière explicite, on cherche des suites géométriques solutions de l'équation de récurrence.

Si $u_n = q^n$ et $u_{n+2} = u_{n+1} + u_n$, alors $q^{n+2} = q^{n+1} + q^n$, donc $q^2 = q + 1$, i.e. q est solution de l'équation du second degré

$$x^2 - x - 1 = 0,$$

dont les solutions sont $\varphi = \frac{1+\sqrt{5}}{2}$ et $\varphi' = \frac{1-\sqrt{5}}{2}$.

On cherche alors deux coefficients α, α' tels que $\mathcal{F}_n = \alpha\varphi^n + \alpha'\varphi'^n$.

En identifiant $\mathcal{F}_0$ et $\mathcal{F}_1$, on trouve $\alpha = \frac{1}{\sqrt{5}}$ et $\alpha' = \frac{-1}{\sqrt{5}}$.

Calcul explicite de la suite de Fibonacci

Les calculs précédents aboutissent donc à l'expression explicite de $\mathcal{F}_n$:

Forme explicite de la suite de Fibonacci

$$\forall n \in \mathbb{N}, \quad \mathcal{F}_n = \frac{1}{\sqrt{5}}\left(\varphi^n - \varphi'^n\right), \quad \text{avec } \varphi = \frac{1+\sqrt{5}}{2} \text{ et } \varphi' = \frac{1-\sqrt{5}}{2}.$$

Cette suite (étroitement liée au nombre d'or φ), revient souvent en mathématiques, mais joue également un rôle en informatique !

De la forme explicite on déduit notamment la croissance exponentielle de $\mathcal{F}_n$, puisque

$$\mathcal{F}_n \underset{n\to+\infty}{\sim} \frac{\varphi^n}{\sqrt{5}} \quad \text{avec} \quad \varphi \simeq 1{,}618\ldots$$

Remarque : L'équivalent est très précis puisque $\mathcal{F}_n$ est en réalité l'entier le plus proche de $\frac{\varphi^n}{\sqrt{5}}$ (conséquence du fait que $0 < \left|\frac{\varphi'^n}{\sqrt{5}}\right| < \frac{1}{2}$)

Calcul numérique de la suite de Fibonacci

Le calcul du terme général de la suite de Fibonacci peut se faire de plusieurs manières :

- **itérativement** (le plus simple)
- à l'aide de la **formule explicite** (seulement pour $n \leq 70$)

```
round( (((1+5**0.5)/2)**n)/5**0.5 )
```

  Pour $n = 71$ on obtient (avec la précision limitée du type `float`)

$$\frac{\varphi^n}{\sqrt{5}} = 308061521170129.7, \text{ alors que } \mathcal{F}_{71} = 308061521170129$$

- **récursivement** (très inefficace avec une implémentation naïve)
- par **exponentiation matricielle** (très efficace pour n très grand)

Calcul itératif de la suite de Fibonacci

Implémentation itérative :

```
def fib1(n):
    "Renvoie le terme d'indice n de la suite de Fibonacci"
    if n==0:
        return 0
    a,b = 0,1 # F(0) et F(1)
    for k in range(2,n+1):
        a,b = b,a+b # F(k-2),F(k-1) -> F(k-1),F(k)
    return b
```

n	100	10000	1000000
`fib1(n)`	$\simeq 3.10^{20}$	$\simeq 10^{2000}$	$\simeq 10^{200000}$
temps de calcul	7 μs	2 ms	11s

Complexité : $O(n)$ additions pour le calcul de $\mathcal{F}_n$

Calcul récursif de la suite de Fibonacci

Implémentation récursive :

```
def fib2(n):
    "Renvoie le terme d'indice n de la suite de Fibonacci"
    if n<=1:
        return n # valeurs explicites pour n=0 et n=1
    return fib2(n-2)+fib2(n-1)
```

n	8	18	32	40	50
`fib2(n)`	21	2584	2178309	102334155	$\simeq 10^{10}$
temps de calcul avec `fib2`	1 μs	1 ms	1 s	1 min	$>$ 1h
temps de calcul avec `fib1`	3 μs	3 μs	3 μs	3 μs	4 μs

À cause de la double récursion, la version récursive considérablement plus lente que la version itérative

Double récursion : une mauvaise idée !

```python
def fib2(n):
    if n<=1:
        return n
    return fib2(n-2)+fib2(n-1)
```

Calculons la complexité de la fonction `fib2` en termes de nombre d'appels récursifs (on pourrait aussi compter le nombre d'additions, ce qui aboutirait à la même conclusion)

Soit c_n le nombre d'appels à la fonction `fib2` lors de l'appel `fib2(n)`

On a $c_0 = 1$, $c_1 = 1$ et $c_n = 1 + c_{n-1} + c_{n-2}$ pour tout $n \geq 2$.

Donc en posant $u_n = c_n + 1$,
on a $u_0 = 2$, $u_1 = 2$ et $u_n = u_{n-1} + u_{n-2}$ pour tout $n \geq 2$.

La suite (u_n) vérifie la même relation de récurrence que $(\mathcal{F}_n)$, et les relations $u_0 = 2\mathcal{F}_1$, $u_1 = 2\mathcal{F}_2$.

Double récursion : une mauvaise idée !

La suite $(u_n/2)$ vérifie la même relation de récurrence que $(\mathcal{F}_n)$, et les relations $u_0/2 = \mathcal{F}_1$, $u_1/2 = \mathcal{F}_2$.

On en déduit que $u_n/2 = \mathcal{F}_{n+1}$ pour tout $n \in \mathbb{N}$, c'est-à-dire

$$\forall n \in \mathbb{N}, \qquad c_n = -1 + \frac{2}{\sqrt{5}}\left(\varphi^{n+1} - \varphi'^{n+1}\right) \underset{n\to+\infty}{\sim} \frac{2\varphi}{\sqrt{5}}.\varphi^n$$

La complexité de `fib2(n)` est donc $O(\varphi^n)$ (complexité exponentielle)

(rappel : on a $u_n = O(v_n)$ si la suite $(\frac{u_n}{v_n})$ est bornée)

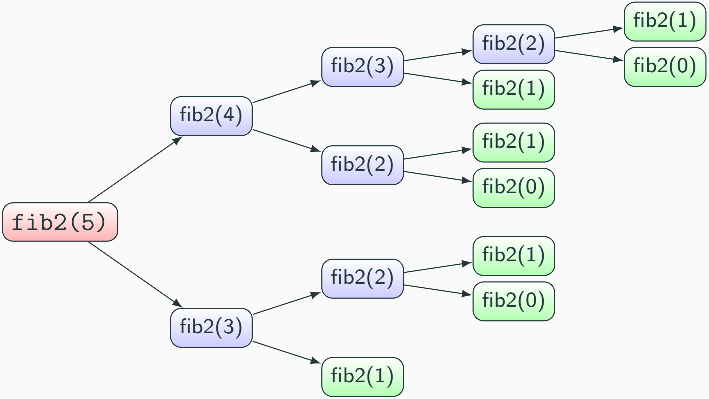
Remarque : Ce résultat n'est pas spécifique à la suite de Fibonacci, mais seulement à l'odre de la récurrence. La complexité est la même pour tout calcul utilisant une double récursion de type

$f(n)$, pour $n \geq 2$, produit un appel récursif à $f(n-1)$ et $f(n-2)$

Mémoïsation

Pourquoi la double récursion est-elle si inefficace ?

Examinons l'arbre des appels récursifs lors de l'appel de `fib2(5)` :



On voit que plusieurs termes sont recalculés plusieurs fois !

Cet écueil peut être évité en utilisant un principe de mémoïsation

Mémoïsation

Principe de la mémoïsation

La mémoïsation consiste, pour chaque calcul d'une fonction donnée, à stocker la valeur calculée dans un dictionnaire pour ne pas avoir à la recalculer lorsque la fonction sera rappelée avec les mêmes arguments.

```python
# suite de Fibonacci: version récursive avec mémoisation
F = {0:0,1:1} # dictionnaire de memoisation (variable globale)
def fib3(n):
    if n not in F: # si fib3(n) n'est pas déjà stockée dans F
        F[n] = fib3(n-2)+fib3(n-1)
    return F[n]
```

n	8	18	32	40	50
temps de calcul avec `fib2`	1 μs	1 ms	1 s	1 min	$>$ 1h
temps de calcul avec `fib3`	3 μs	7 μs	11 μs	14 μs	17 μs

Diviser pour régner

Le principe “diviser pour régner”

Définition

“Diviser pour régner” (*divide and conquer* en anglais) est un principe d’algorithmique récursive, qui consiste à résoudre un problème en le découpant en sous-problèmes plus petits de même nature.

Ce principe repose typiquement sur le schéma suivant pour résoudre un problème X :

fonction résoudre(X)

- découper X en deux sous-problèmesx X_1 et X_2
- $Y_1 \leftarrow$ résoudre(X_1)
- $Y_2 \leftarrow$ résoudre(X_2)
- calculer la solution Y associée à X à partir de Y_1 et Y_2
- retourner Y

Application : tri rapide (*quicksort*)

L'algorithme de tri rapide repose sur le principe "diviser pour régner"

Il permet de trier une liste L selon le schéma suivant :

> **fonction tri(L)**
> - si L est vide, renvoyer L
> - retirer un élément x de la liste L
> - construire la liste L_1 de tous les éléments de L qui sont $< x$
> - construire la liste L_2 de tous les éléments de L qui sont $\geq x$
> - retourner la liste obtenue en concaténant tri(L_1), x, et tri(L_2)

En général, l'élément x choisi est le premier de la liste, ou le dernier, ou bien même un élément pris au hasard

Application : tri rapide (*quicksort*)

fonction tri(L)

- si L est vide, renvoyer L
- retirer un élément x de la liste L
- construire la liste L_1 de tous les éléments de L qui sont $< x$
- construire la liste L_2 de tous les éléments de L qui sont $\geq x$
- retourner la liste obtenue en concaténant tri(L_1), x, et tri(L_2)

Implémentation Python du tri rapide :

```python
def tri(L):
    if len(L)==0:
        return L
    x = L[0]
    L1 = [a for a in L[1:] if a<x]
    L2 = [a for a in L[1:] if a>=x]
    return tri(L1) + [x] + tri(L2)
```

Application : tri rapide (*quicksort*)

Vérification de la fonction `tri` :

```
>>> L = [5,1,10,3,-2,4,1] # une liste
>>> tri(L) # résultat obtenu avec la fonction tri
[-2, 1, 1, 3, 4, 5, 10]

>>> from random import random # nombres aléatoires dans [0,1]
>>> L = [random() for i in range(1000)] # 1000 nombres
>>> T = tri(L) # tri des éléments de L
>>> T==sorted(L) # comparaison au résultat obtenu avec sorted()
True
```

Complexité du tri rapide (pour N éléments) :

- en moyenne : $O(N \log N)$
- dans le cas le pire : $O(N^2)$

Application : tri fusion (*merge sort*)

L'algorithme de tri fusion repose également sur le principe "diviser pour régner"

Il permet de trier une liste L selon le schéma suivant :

> **fonction tri_fusion(L)**
> - si L est de taille 0 ou 1, renvoyer L
> - séparer (en coupant à l'indice moitié) L en L_1 et L_2
> - $T_1 \leftarrow$ tri_fusion(L_1)
> - $T_2 \leftarrow$ tri_fusion(L_2)
> - fusionner T_1 et T_2 en une seule liste triée T
> - retourner T

Application : tri fusion (*merge sort*)

Fusion de deux listes triées (T_1 **de taille** n_1 **et** T_2 **de taille** n_2**) :**

- $i_1 \leftarrow 0$, $i_2 \leftarrow 0$
- $T \leftarrow [\,]$
- **tant que** $i_1 < n_1$ **et** $i_2 < n_2$

  si $T_1[i_1] < T_2[i_2]$

      ajouter $T_1[i_1]$ à la fin de T et incrémenter i_1

  sinon

      ajouter $T_2[i_2]$ à la fin de T et incrémenter i_2

- compléter T avec $(T_1[i])_{i \geq i_1}$ et $(T_2[i])_{i \geq i_2}$

Remarque : à la dernière étape, on a soit $i_1 = n_1$ soit $i_2 = n_2$, donc on ne complète en fait T que par la fin de T_1 ou de T_2

Application : tri fusion (*merge sort*)

Implémentation Python :

```python
def tri_fusion(L):
    n = len(L)
    if n<=1:
        return L
    n1,n2 = n//2,n-n//2
    T1 = tri_fusion(L[:n1])
    T2 = tri_fusion(L[n1:])
    # fusion de T1 et T2
    i1,i2,T = 0,0,[]
    while i1<n1 and i2<n2:
        if T1[i1]<T2[i2]:
            T.append(T1[i1])
            i1 = i1+1
        else:
            T.append(T2[i2])
            i2 = i2+1
    return T+T1[i1:]+T2[i2:]
```

Application : tri fusion (*merge sort*)

Complexité du tri fusion (pour N éléments) :

- en moyenne : $O(N \log N)$
- dans le cas le pire : $O(N \log N)$

Remarque : Python utilise, pour la fonction standard `sorted()` et la méthode `sort()` du type `list`, une variante du tri fusion (un mélange de tri par insertion et de tri fusion)

Complexité du tri fusion

On se place tout d'abord dans le cas où $N = 2^n$

Soit $c(N)$ le nombre maximum de comparaisons (entre éléments de la liste initiale) effectuées pour le tri fusion de N éléments

On a $c(N) = 2c(N/2) + N - 1$ donc en posant $d(n) = c(N)/N$,

$$d(N) \leq d\left(\frac{N}{2}\right) + 1 \quad \text{avec} \quad d(1) = c(1) = 0.$$

On en déduit

$$d(2^n) \leq d(2^{n-1}) + 1 \leq d(2^{n-2}) + 2 \leq \ldots \leq d(2^{n-n}) + n = n.$$

D'où, quand $N \to +\infty$, $d(2^n) = O(n)$, $c(2^n) = O(n2^n)$ et finalement $c(N) = O(N \log_2 N)$ puisque $n = \log_2 N = \frac{\ln N}{\ln 2}$.

Conclusion : $c(N) = O(N \log N)$ dans le cas le pire

Cette formule reste valable lorsque N n'est pas une puissance de 2 (on majore alors N par la puissance de 2 immédiatement supérieure)

Application : exponentiation rapide

On souhaite calculer la puissance n-ième d'une matrice (pour $n \in \mathbb{N}$).

On suppose que le produit matriciel est déjà implémenté et s'écrit $A \times B$

Version itérative : on écrit $A^n = I \times A \times A \times \ldots \times A$

> **fonction** puissance(A,n)
> $B \leftarrow I$ (matrice identité de même taille que A)
>   pour k allant de 1 à n
> $B \leftarrow B \times A$
>   retourner B

Complexité (en nombre de multiplications matricielles) : $O(n)$

Application : exponentiation rapide

Version selon le principe "diviser pour régner", récursive :

On remarque que

- si n est pair, alors $A^n = B \times B$ avec $B = A^{n/2}$,
- si n est impair, alors $A^n = A^{n-1} \times A$, avec $n - 1$ pair.

> **fonction** puissance(A,n)
>   si $n = 0$
>   retourner I (matrice identité de même taille que A)
>   si n est impair
>   retourner puissance(A,$n-1$) $\times A$
> $B \leftarrow$ puissance(A,$n/2$)
>   retourner $B \times B$

Complexité (en nombre de multiplications matricielles) : $O(\log n)$

fonction puissance(A,n)
  si $n = 0$
    retourner I (matrice identité de même taille que A)
  si n est impair
    retourner puissance(A,$n-1$) $\times A$
  $B \leftarrow$ puissance(A,$n/2$)
  retourner $B \times B$

Exemple : calcul de A^{102}

$$A^{102} = (A^{51})^2 \qquad A^{51} = A^{50} \times A \qquad A^{50} = (A^{25})^2$$
$$A^{25} = A^{24} \times A \qquad A^{24} = (A^{12})^2 \qquad A^{12} = (A^{6})^2$$
$$A^{6} = (A^{3})^2 \qquad A^{3} = A^{2} \times A \qquad A^{2} = A \times A$$

coût total : 9 multiplications

Application : Pour $N = 10^{100}$, calculer modulo N la matrice $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^N$, et en déduire les 100 derniers chiffres de $\mathcal{F}_N$ (suite de Fibonacci).

Générateurs

Types séquence pour une boucle `for`

Pour réaliser une boucle `for`, nous avons besoin d'un objet de type *séquence* (`list`, `tuple`, mais aussi `set` par exemple)

Les codes suivants sont ainsi équivalents (à l'ordre près pour `set`) :

```
# list
for i in [0,1,2]:
    print(i)
0
1
2
```

```
# tuple
for i in (0,1,2):
    print(i)
0
1
2
```

```
# set (non ordonné !)
for i in {0,1,2}:
    print(i)
0
1
2
```

Ce type de boucle est d'ailleurs plus communément effectué à l'aide de la fonction `range` en général :

```
# avec range
for i in range(3): # pour i allant de 0 à 2
    print(i)
0
1
2
```

La fonction `range`

Une question naturelle est donc : que renvoie la fonction `range()` ?

```
>>> a = range(10**30)
>>> a
range(0, 1000000000000000000000000000000)
>>> type(a)
<class 'range'>
```

`range(10**30)` n'est pas la séquence $0, 1, \ldots 10^{30} - 1$ (qui ne pourrait tenir dans la mémoire d'un ordinateur), mais un objet Python capable de générer la séquence $0, 1, \ldots 10^{30} - 1$.

On dit que `range(10**30)` est un itérable (de même qu'une liste, un tuple ou un ensemble). À partir d'un itérable, on peut fabriquer un itérateur, qui permet de générer les valeurs successives.

Intérêt des itérables

L'utilisation d'un itérable obtenu avec `range()` ou avec un générateur (plutôt qu'une liste des valeurs correspondantes par exemple) a plusieurs avantages.

Le principal avantage est l'économie de mémoire, et même la possibilité de manipuler "à la volée" des séquences qui ne tiendraient pas en mémoire.

Exemple : calculer la somme des éléments d'un itérable

```
>>> a = range(10**8) # itérable (entiers de 0 à 10^8-1)
>>> sum(a)
4999999950000000
>>> b = list(a) # liste associée (1 Go en mémoire !)
>>> sum(b)
4999999950000000
>>> a = range(10**9) # jusqu'à 1 milliard maintenant
>>> sum(a)
499999999500000000
>>> b = list(a) # ne tient pas en mémoire
Killed
```

Expression génératrice

Pour calculer la somme des carrés des entiers de 1 à 100, nous pouvons utiliser une compréhension de liste, et lui appliquer la fonction `sum()` :

```
>>> [x**2 for x in range(1,101)] # compréhension de liste
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196,
...
8649, 8836, 9025, 9216, 9409, 9604, 9801, 10000]
>>> sum([x**2 for x in range(1,101)])
338350
```

Mais il est beaucoup plus économique (en termes de mémoire, et donc aussi de temps de calcul) d'utiliser une expression génératrice, qui s'obtient en remplaçant les crochets par des parenthèses dans la compréhension de liste précédente :

```
>>> (x**2 for x in range(1,101)) # expression génératrice
<generator object <genexpr> at 0x7f30dc633870>
>>> sum(x**2 for x in range(1,101))
338350
```

Expression génératrice contre compréhension de liste

Nous pouvons reprendre l'exemple précédent, qui illustre ici l'intérêt d'une expression génératrice sur la compréhension de liste équivalente (qui, cette fois encore, ne tient pas en mémoire) :

```
>>> sum(x**2 for x in range(10**9)) # expression génératrice
333333332833333333500000000
>>> sum([x**2 for x in range(10**9)]) # compréhension de liste
Killed
```

Nous venons de voir comment construire un générateur à l'aide d'une expression génératrice.

Si l'on souhaite fabriquer un générateur plus général qui dépend d'arguments, on peut utiliser une fonction génératrice.

Définition d'une fonction génératrice

Une fonction génératrice est semblable à une fonction usuelle, mais elle renvoie un générateur capable de délivrer successivement plusieurs valeurs. La définition se fait avec la même instruction `def` que pour les fonctions, mais l'instruction `return` des fonctions est, pour une fonction génératrice, remplacée par `yield` (*yield* = donner en anglais)

```
def f(a): # exemple de fonction génératrice
    yield a
    yield a*2
    yield a+1

>>> f(10) # générateur produit par f
<generator object f at 0x7f30dc6337e0>
>>> list(f(10)) # réalisation de f(10) en liste
[10, 20, 11]
>>> for x in f(10): # boucle sur f(10)
...     print(x)
10
20
11
```

Expression génératrice et fonction génératrice

Nous avons vu précédemment comment constuire le générateur des carrés des entiers de 1 à 100 à l'aide d'une expression génératrice :

```
>>> g = (x**2 for x in range(1,101))
>>> g
<generator object <genexpr> at 0x7ff4035c08b8>
```

Il serait équivalent de définir g à l'aide de la fonction génératrice `G` suivante :

```
def G(n):
    for x in range(1,n+1):
        yield x**2

>>> G
<function G at 0x7ff4035c7510>
>>> g = G(100)
>>> g
<generator object G at 0x7ff405524288>
```

Épuisement d'un générateur

Nous avons vu qu'un générateur renvoie successivement plusieurs valeurs. Lorsque la liste est épuisée, le générateur ne renvoie plus rien !

```
>>> g = (x for x in range(10)) # générateur de 0,1,...,9
>>> list(g) # réalisation de g dans une liste
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(g) # g est maintenant épuisé !
[]
>>> g = (x for x in range(10)) # on redéfinit g
>>> for i in g:
...     if i==3: # si i = 3,
...         break # on force la sortie de la boucle
...
>>> list(g) # réalisation du reste de g dans une liste
[4, 5, 6, 7, 8, 9]
>>> list(g) # le générateur g est épuisé
[]
```

Exemples de générateurs

Générateur de tous les entiers naturels :

```
def entiers():
    n = 0 # initialisation
    while True:
        yield n # envoie n
        n = n+1 # incrémente n
```

Attention à ce générateur, il ne termine pas spontanément !

```
for n in entiers():
    print(n)
    if n==1000: # si n = 1000,
        break # on force la sortie de la boucle
```

```
1
2
3
...
999
1000
```

Exemples de générateurs

Générateur (non optimisé) des nombres premiers inférieur ou égaux à N :

```
def premiers(N):
    "renvoie un générateur des nombres premiers entre 1 et N"
    p = 2 # initialisation
    while p<=N:
        yield p # envoie p (premier)
        # cherche le nombre premier suivant
        p = p+1
        while any(p%x==0 for x in range(2,p)):
            p = p+1

>>> list(premiers(50))
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

Remarque : `any(L)` effectue un **ou logique** entre tous les éléments d'une liste (ou d'un itérable) de booléens (donc renvoie `True` si au moins l'un des éléments est `True`)

Exemples de générateurs

```python
def fib(N):
    "générateur des termes 0 à N de la suite de Fibonacci"
    yield 0 # F(0)
    if N>=1:
        yield 1 # F(1)
    a,b = 0,1
    for n in range(2,N+1):
        a,b = b,a+b
        yield b # F(n)

>>> x = fib(10)
>>> x
<generator object fib at 0x7f74e47f5240>
>>> list(x)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

Générateurs et récursivité

Comme une fonction classique, une fonction génératrice peut être récursive.

Exemple : générateur de tous les sous-ensembles d'un ensemble donné

```
def sous_ensembles(A):
    "générateur de tous les sous-ensembles de A"
    if len(A)==0: # si A est vide
        yield set() # ensemble vide
    else:
        x = A.pop() # extrait un élément x de A
        for B in sous_ensembles(A): # appel récursif
            yield B
            yield B|{x}

>>> list(sous_ensembles({1,2,3}))
[set(), {1}, {2}, {1, 2}, {3}, {1, 3}, {2, 3}, {1, 2, 3}]
```

Générateurs et récursivité

Générateur de toutes les permutations d'un tuple ou d'une chaîne :

```
def permutations(T):
    "générateur de toutes les permutations de T"
    if len(T)==1: # un seul élément
        yield T
    elif len(T)>=2: # au moins 2 éléments
        n = len(T)
        for t in permutations(T[1:]): # appel récursif
            for i in range(n): # insère T[0] en position i
                yield t[:i]+T[:1]+t[i:]

>>> list(permutations('abcd'))
['abcd', 'bacd', 'bcad', 'bcda', 'acbd', 'cabd', 'cbad',
 'cbda', 'acdb', 'cadb', 'cdab', 'cdba', 'abdc', 'badc',
 'bdac', 'bdca', 'adbc', 'dabc', 'dbac', 'dbca', 'adcb',
 'dacb', 'dcab', 'dcba']
```

Le module `itertools`

Le module `itertools` contient plusieurs fonctions génératrices très utiles.

- `permutations(I)` : génère toutes les permutations de l'itérable `I`
- `permutations(I,k)` : arrangements à `k` éléments de `I`
- `combinations(I,k)` : combinaisons à `k` éléments de `I`
- `product(A1,A2,...,An)` : produit cartésien $A_1 \times A_2 \times \ldots \times A_n$
- `product(A, repeat=n)` : produit cartésien A^n
- etc.

Le module itertools

```
>>> from itertools import permutations,combinations,product

>>> list(permutations('abc')) # permutations
[('a', 'b', 'c'), ('a', 'c', 'b'), ('b', 'a', 'c'),
 ('b', 'c', 'a'), ('c', 'a', 'b'), ('c', 'b', 'a')]

>>> list(permutations([1,'a',set()])) # permutations
[(1, 'a', set()), (1, set(), 'a'), ('a', 1, set()),
('a', set(), 1), (set(), 1, 'a'), (set(), 'a', 1)]

>>> list(permutations('abc',2)) # arrangements
[('a', 'b'), ('a', 'c'), ('b', 'a'), ('b', 'c'),
 ('c', 'a'), ('c', 'b')]

>>> list(combinations('abc',2)) # combinaisons
[('a', 'b'), ('a', 'c'), ('b', 'c')]

>>> list(combinations(range(1,6),3)) # combinaisons
[(1, 2, 3), (1, 2, 4), (1, 2, 5), (1, 3, 4), (1, 3, 5),
(1, 4, 5), (2, 3, 4), (2, 3, 5), (2, 4, 5), (3, 4, 5)]
```

Le module `itertools`

```
>>> list(product(['rouge','vert','bleu'],['clair','foncé']))
[('rouge', 'clair'), ('rouge', 'foncé'), ('vert', 'clair'),
('vert', 'foncé'), ('bleu', 'clair'), ('bleu', 'foncé')]

>>> list(product((0,1),repeat=4))
[(0, 0, 0, 0), (0, 0, 0, 1), (0, 0, 1, 0), (0, 0, 1, 1),
 (0, 1, 0, 0), (0, 1, 0, 1), (0, 1, 1, 0), (0, 1, 1, 1),
 (1, 0, 0, 0), (1, 0, 0, 1), (1, 0, 1, 0), (1, 0, 1, 1),
 (1, 1, 0, 0), (1, 1, 0, 1), (1, 1, 1, 0), (1, 1, 1, 1)]
```

La fonction `product()` permet de simuler, avec une seule boucle, plusieurs boucles imbriquées :

```
for i in (1,3,9):
    for j in range(20):
        for k in 'abcd':
            ....
```

est équivalent à

```
for i,j,k in product((1,3,9),range(20),'abcd'):
    ...
```

Le module `itertools` : exemples

- Combien y a-t-il de façons d'écrire 37 comme somme de 5 entiers strictement positifs ?

```
>>> sum(sum(x)==37 for x in product(range(1,38),repeat=5))
58905
```

- Quelle est la probabilité de tirer 7 fois "pile" en 10 lancers de pièce ?

```
>>> 2**-10*sum(sum(x)==7 for x in product((0,1),repeat=10))
0.171875
```

- Résoudre le cryptarithme $\text{SIX}^2 = \text{TROIS}$
(chaque lettre représente un chiffre différent) :

```
>>> for s,i,x,t,r,o in permutations('0123456789',6):
...     if int(s+i+x)**2 == int(t+r+o+i+s):
...         print('trois='+t+r+o+i+s+' six='+s+i+x)
...
trois=28561 six=169
```