SUR LA SUITE DE FIBONACCI

MICHEL BOYER

RÉSUMÉ. La suite de Fibonacci permet d'illustrer plusieurs aspects du cours IFT 1063: suites et fonctions, récursivité, relation de récurrence, algorithmes, induction, analyses d'algorithmes. On s'en sert donc ici pour brosser rapidement un portrait, qui reste toutefois partiel, du cours.

1. DÉFINITION DE LA FONCTION DE FIBONACCI

La suite de Fibonacci est introduite en page 177 de Johnsonbaugh. Elle commence comme suit

$$1, 1, 2, 3, 5, 8, 13, \ldots$$

et chaque terme de la suite est la somme des deux précédents. Si on note f_n le n^{ieme} terme, la définition en français qui précède sécrit mathématiquement

$$\begin{aligned} f_1 &= 1 \\ f_2 &= 1 \\ f_n &= f_{n-1} + f_{n-2} \qquad \text{si } n \geq 3 \end{aligned}$$

Les informaticiens aiment souvent faire commencer les indices à 0 plutôt qu'à 1 et on pose alors $f_0 = 0$ pour que l'égalité $f_n = f_{n-1} + f_{n-2}$ tienne pour tout $n \geq 2$. On a alors

FIG. 1. *Suite de Fibonacci à partir du terme d'indice* 0

n	0	1	2	3	4	5	6	7	8	...
f_n	0	1	1	2	3	5	8	13	21	...

Nous allons nous intéresser au calcul du n-ième terme f_n de la suite de Fibonacci en se limitant, comme les mathématiciens, aux indices $n \geq 1$. On écrira ici F(n) au lieu de f_n; du point de vue mathématique c'est tout à fait justifié:

$$\mathrm{F}(n) = \begin{cases} 1 & \text{si } n = 1 \\ 1 & \text{si } n = 2 \\ \mathrm{F}(n-1) + \mathrm{F}(n-2) & \text{si } n \geq 3 \end{cases}$$

Cette définition est dite *récursive* car F en partie gauche est définie en termes de F en partie droite. On peut aussi la voir comme une équation dite aux différences finies, aussi appelée relation de récurrence. C'est le sujet du chapitre 7 de la sixième édition de Johnsonbaugh.

2. UNE IMPLANTATION DIRECTE DE LA FONCTION DE FIBONACCI

La majorité des langages de programmation modernes acceptent que l'on définisse une fonction en utilisant directement les trois cas ci-dessus. C'est en particulier le cas de Java, C, Pascal, Scheme, Modula, Ada, C++ et aussi Python. Pour éviter d'avoir à faire de longues déclarations (pas très utiles pour des algorithmes de trois lignes) et aussi pour profiter du fait que Python permet de manipuler des grands entiers (avec des centaines de décimales) nous allons utiliser Python. Nous mettons dans le fichier `fib0.py` les 5 lignes qui suivent (à partir de la colonne 0 et en faisant bien attention à l'indentation):

FIG. 2. *Implantation récursive simple de* F

```
def F(n):
  if n == 1:      return 1
  else:
    if n == 2:    return 1
    else:         return F(n-1) + F(n-2)
```

C'est tout! On peut maintenant en commander l'exécution (mais pas pour F(0) ni pour n négatif). Voici une trace sur Deimos; les >>> sont des "prompts" et ce qui suit à droite est tapé par l'usager:

```
deimos 2 % python
Python 2.3.3 (#1, May  7 2004, 10:31:40)
[GCC 3.3.3 20040412 (Red Hat Linux 3.3.3-7)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from fib0 import *
>>> F(1)
1
>>> F(4)
3
>>> F(5)
5
>>> F(6)
8
>>> F(30)
832040
>>>
```

La valeur F(30) prend un temps non négligeable à venir et pour $n > 30$ le temps devient rapidement insupportable. On met rapidement sa patience à épreuve en faisant calculer F(31) ou F(32).

Plusieurs questions se posent: pourquoi une telle lenteur? comment prévoir le temps d'exécution? peut-on concevoir un algorithme plus rapide pour le calcul de la fonction de Fibonacci?

3. UNE SOLUTION "MATHÉMATIQUE"

On peut en fait trouver une formule mathématique qui calcule f_n; on résout l'*équation de récurrence* $f_n - f_{n-1} - f_{n-2} = 0$ avec *conditions initiales* $f_1 = 1$ et $f_2 = 1$ dans l'exemple 7.2.13 de Johnsongauth et l'on obtient

$$\mathrm{F}(n) = f_n = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n$$

Il est surprenant qu'une telle formule, remplie d'irrationnels[1] ($\sqrt{5}$ ne peut pas s'exprimer comme une fraction entière) donne comme réponse un nombre entier. Cette fonction peut facilement s'implanter par programme; or $\frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n$ est toujours inférieur à 0.5 et on peut donc obtenir F(n) en arrondissant simplement $\frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n$; mettons dans `fiboe.py` les trois lignes (non blanches) suivantes; la première ligne demande à Python d'importer toutes les fonctions du module `math` mais il aurait suffi d'en importer la fonction `round`; les deux autres lignes définissent F(n):

```
from math import *

def F(n):
   return round( ((1+sqrt(5))/2)**n/sqrt(5))
```

Voici maintenant une trace d'exécution sur Deimos:

```
deimos 2 % python
Python 2.3.3 (#1, May  7 2004, 10:31:40)
[GCC 3.3.3 20040412 (Red Hat Linux 3.3.3-7)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from fiboe import *
>>> F(30)
832040.0
>>> F(100)
3.542248481792618e+20
>>> F(200)
2.8057117299250997e+41
>>> F(1000)
4.3466557686938915e+208
>>> F(2000)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "fiboe.py", line 4, in fibo
    return round( ((1+sqrt(5))/2)**n/sqrt(5))
OverflowError: (34, 'Numerical result out of range')
>>>
```

Les résultats arrivent avec le retour du charriot même pour F(1000) mais alors le résultat n'est donné qu'approximativement (en notation scientifique) et pour F(2000) il y a débordement.

4. CALCUL DE DEUX VALEURS SUCCESSIVES DE LA SUITE DE FIBONACCI

Peut-on faire mieux? Il serait bien d'avoir toutes les décimales, et en un temps acceptable. On y arrive par une méthode a priori surprenante: on définit une

1. Il est à noter que le nombre $\frac{1+\sqrt{5}}{2}$ est le nombre d'or.

fonction F2 telle que F2(n) retourne les deux valeurs [F(n),F(n+1)] plutôt qu'uniquement F(n). Par exemple

F2(1) = [1,1]
F2(2) = [1,1 + 1] = [1,2]
F2(3) = [2,1 + 2] = [2,3]
F2(4) = [3,2 + 3] = [3,5]
F2(5) = [5,3 + 5] = [5,8]

et ainsi de suite; à chaque étape $n \geq 2$ on calcule F2(n) directement à partir des deux valeurs contenues dans F2($n-1$). Reste à écrire une procédure (ou un programme Python) pour le faire.

Python permet de retourner deux valeurs entre crochets. De plus si s = [5,8] alors s[0] donne 5 et s[1] donne 8 (en informatique, en général, les suites finies sont indexées à partir de 0). De plus Python connaît les grands entiers: suffit de les faire suivre d'un L majuscule. Ainsi 1L est le grand entier 1. Voici maintenant la fonction Python (fichier `fib2.py`) qui calcule F2 à partir de n = 1:

FIG. 3. *Implantation récursive de* F2

```
def F2(n):
  if n == 1: return [1L, 1L]
  else:
    s = F2(n-1)
    return  [s[1], s[0]+s[1]]
```

Et voici une trace d'exécution:

```
>>> from fib2 import *
>>> F2(30)
[832040L, 1346269L]
>>> F2(100)
[354224848179261915075L, 573147844013817084101L]
>>> F2(200)
[280571172992510140037611932413038677189525L,
 453973694165307953197296969697410619233826L]
>>>
```

Les résultats arrivement immédiatement; curieux qu'il soit plus rapide de calculer deux valeurs plutôt qu'une! On va voir plus loin que le temps que prendrait en principe la procédre récursive simple (Fig. 2) pour calculer F(200) est supérieur à l'âge de l'univers (plusieurs milliards d'années). La procédure que nous venons de voir a cependant ses limites; on évalue sans peine F2(999) mais F2(1000) fait déborder la pile d'exécution de Python.

5. ALGORITHME ITÉRATIF POUR LE CALCUL DE F(n)

Nous allons d'abord décrire un algorithme itératif pour le calcul de F2 puis en déduire F(n). Nous pourrons obtenir sans problème F(10000). L'algorithme utilise une *boucle* qui permet de répéter une opération un nombre donné de fois: en Python, `for i in range(n):` permet de répéter une opération n

fois. Comme Python permet ici d'implanter (en 5 lignes) et de tester rapidement, nous allons encore coder en Python.

Nous allons maintenant décire plus en détails ce qui a été fait à la section précédente, mais en utilisant seulement deux cases mémoire (deux variables) que nous dénoterons `f` (pour le terme courant de la suite de Fibonacci) et `s` (pour le prochain terme qu'on précalcule). Au début `[f, s]` contient [0,1] i.e. [F(0),F(1)] c'est-à-dire que `f` contient F(0) = $f_0 = 0$ et `s` contient F(1) = $f_1 = 1$. Il est ici en effet plus facile de calculer aussi F2(0). Le tableau qui suit donne à la colonne n le contenu de `f` et `s` après n "tours de boucle" comme on dit: chaque colonne représente donc la valeur de F2(n) et s'obtient de la précédente comme suit: on met sur la ligne `f` le contenu de la ligne `s` de la colonne précédente et sur la ligne `s` la somme des contenus des deux lignes de la colonne précédente. Dans un langage plus mathématique, $[f_n,s_n] = [s_{n-1},f_{n-1}+s_{n-1}]$ où $[f_n,s_n]$ représente la n-ième colonne. Voici le tableau pour n de 0 à 8:

FIG. 4. *Table de calcul de* F(n) *pour* $n \geq 0$

n	0	1	2	3	4	5	6	7	8	...
f	0	1	1	2	3	5	8	13	21	...
s	1	1	2	3	5	8	13	21	34	...

Cet algorithme donne lieu au programme Python du fichier `fib2i.py` suivant :

FIG. 5. *Implantation itérative de la fonction* F2(n)

```
def F2(n):
  [f, s] = [0L, 1L]
  for i in range(n):          # on boucle n fois
    [f, s] = [s, f+s]         # affectation "simultanee"
  return [f,s]
```

Bien sûr, si l'on veut uniquement F(n) suffit de retourner f au lieu de $[f,s]$. On obtient ainsi l'algorithme suivant pour le calcul du n-ième terme de la suite de Fibonacci (fichier `fibi.py`):

FIG. 6. *Implantation itérative de la fonction de Fibonacci*

```
def F(n):
  [f, s] = [0L, 1L]
  for i in range(n):          # on boucle n fois
    [f, s] = [s, f+s]         # affectation "simultanee"
  return f
```

En voici une trace d'exécution:

```
>>> from fibi import *
>>> F(100)
354224848179261915075L
>>> F(200)
280571172992510140037611932413038677189525L
>>> F(1000)
43466557686937456435688527675040625802564660517371780402481729089536555417
949051 89040387984007925516929592259308032263477520968962323987332247116164
4299644090653 3187938298969649928516003704476137795166849228875L
```

La trace d'exécution de F(10000) est impressionnante (la réponse remplit un écran). On voit aussi que les dernières décimales pour F(200) dans la section 3 étaient inexactes.

6. ANALYSE DE L'ALGORITHME DE LA FIGURE 2

Un des points d'intérêt du cours IFT 1063 est la détermination de bornes sur le temps d'exécution d'algorithmes.

6.1. **Une formule pour borner inférieurement le temps d'exécution.** Soit T(n) le temps d'exécution de la fonction de la Fig. 2. Notons δ le minimum entre le temps d'exécution de F(0) et celui de F(1). On a donc $\delta \leq$ T(0) et $\delta \leq$ T(1). Ce temps est normalement très petit, à peine le temps de l'appel de la fonction. Pour $n \geq 3$, l'évaluation de F(n) se fait en évaluant F($n-1$) puis F($n-2$) et en additionnant les résultats. Le temps d'exécution est certainement supérieur ou égal à la somme des temps nécessaires pour évaluer F($n-1$) et F($n-2$). Autrement dit

$$\begin{aligned} &T(1) \geq \delta \\ &T(2) \geq \delta \\ &T(n) \geq T(n-1) + T(n-2) \qquad \text{pour } n \geq 3 \end{aligned}$$

On peut ainsi construire de proche en proche un tableau de temps inférieurs ou égaux à T(n). Par exemple T(3) $\geq$ T(1) + T(2) $\geq 2\delta$ et en faisant le même raisonnement pour T(4), T(5) etc. on obtient le tableau suivant

FIG. 7. *Borne inférieure sur le temps de calcul de* F(n)

n	1	2	3	4	5	6	7	8	...
T(n) $\geq$	δ	δ	2δ	3δ	5δ	8δ	13δ	21δ	...

En comparant ce tableau avec celui de la Fig 1 on voit, encore de proche en proche, que

$$\boxed{T(n) \geq F(n)\delta \quad \text{pour } n \geq 1}$$

On fait ici une "preuve avec des petits points" qui s'appelle plus savamment (après avoir correctement formalisé la chose) une preuve par *induction mathématique* et c'est l'une des méthodes de preuves étudiées en IFT 1226.

6.2. **Borne inférieure pour le temps de calcul de** F(200)**.** On a vu plus haud que $F(200) > 2.805 \times 10^{41}$. Quel serait le temps de calcul de F(200) avec la procédure récursive?; si δ est un nanoseconde (i.e. 10^{-9} sec, ce qui est très petit) alors

$$T(200) \geq 10^{-9} \times 2.805 \times 10^{41} \text{sec} = 2.805 \times 10^{32} \text{sec}$$

Combient ca fait de temps? Notons qu'une année est égale à

$$365.25 \times 24 \times 60 \times 60 \text{ sec} = 3.156 \times 10^{7} \text{sec}$$

(ou encore 3.156×10^{16} nanosecondes). On en déduit que 3.156×10^{8}sec font 10 ans, que 3.156×10^{9} sec font cent ans, 3.156×10^{10}sec font mille ans. Mais combien donc font 2.805×10^{32}sec? C'est énorme.

En fait, selon les données de la NASA l'âge de l'univers est de 13.7×10^{9} ans (13.7 milliards d'années) avec une erreur d'au plus 1%. À raison de 365.25 jours par an, 24 heures par jour, 60 minutes par heure et 60 secondes par minute, celà donne

$$13.7 \times 10^{9} \times 365.25 \times 24 \times 60 \times 60 \text{ sec} = 4.32 \times 10^{17} \text{ sec}$$

Ce temps est largement dépassé par le temps nécessaire pour calculer F(200). Par étonnant que l'algorithme de la section 2 se faisait sentir lent. Imaginez un décideur qui, sans comprendre ce qui se passe et tenant à utiliser l'algorithme de la Fig. 2, ferait acheter un ordinateur superpuissant coûtant une fortune pour n'obtenir que quelques valeurs additionnelles de la suite de Fibonacci.

7. Peut-on faire encore mieux que la Fig 6?

La procédure itérative implantée par le programem Python de la Fig. 6 est amplement satisfaisante pour des inputs inférieurs à quelques dizaines de milliers. Elle est lente sur le calcul de F(100000). On peut se demander s'il y a moyen de faire mieux et la réponse est que c'est relativement facile (mais plus en 4 ou 5 lignes)[2] si l'on connaît les matrices. De fait, la solution mathématique avait en principe un avantage en terme de vitesse car on peut calculer la puissance n-ième d'un nombre en beaucoup moins que n étapes alors que notre boucle de la Fig. 6 nécessite n étapes. Par exemple 2^{16} se calcule avec 4 multiplications: on calcule d'abord $2^2 = 2 \times 2 = 4$ puis on calcule $2^4 = 4 \times 4 = 16$ puis on calcule $2^8 = 16 \times 16 = 256$ et enfin $2^{16} = 256 \times 256$, ce qui est notre quatrième multiplication. Cette idée se généralise pour des exposants qui ne sont pas des puissances de 2.

Comment maintenant les matrices interviennent-elles? L'opération qui consiste à remplacer f par s et s par $f+s$ peut s'implanter par un produit matriciel; en effet

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} f \\ s \end{bmatrix} = \begin{bmatrix} s \\ f+s \end{bmatrix}$$

2. Il y a une solution futée qui s'écrit en 12 lignes de Python et qui fait disparaître les matrices mais si on fait l'implantation soignée avec les matrices, ca prend près de 60 lignes.

et donc une simple multiplication par $\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$ fait passer d'une colonne à la suivante dans le tableau de la Fig. 4. Par exemple la colonne 3 est $\begin{bmatrix} 2 \\ 3 \end{bmatrix}$ et la colonne 2 est $\begin{bmatrix} 1 \\ 2 \end{bmatrix}$ et

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 2 \\ 3 \end{bmatrix}$$

De proche en proche, on a

$$\begin{bmatrix} 2 \\ 3 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

et, de façon générale, pour retrouver la colonne n, suffit de multiplier la colonne 0 par $\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$ un nombre n fois: on obtient donc, en mettant en colonne la valeur retournée par F2(n) c'est-à-dire en posant $F2(n) = \begin{bmatrix} F(n) \\ F(n+1) \end{bmatrix} = \begin{bmatrix} f_n \\ s_n \end{bmatrix}$:

$$F2(n) = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Le même principe qui permet d'implanter une exponentiation rapide pour les nombres s'applique aussi pour les matrices et celà nous donne une implantation en principe beaucoup plus efficace pour le calcul de la fonction de Fibonacci; en fait, le gain se fait sentir pour de grandes valeurs de n. Reste bien sûr à implanter le produit de matrices mais ça n'est pas compliqué du tout si on se limite à des matrices 2×2. Si l'on essaie, on se rend compte que le calcul de F(100000) est sensiblement plus rapide qu'avec la procédure de la section 5 (sur Deimos, F(100000) est calculé en 2 secondes au lieu de 17 secondes). La réponse contient 20 899 décimales.

Références

Richard Johnsonbaugh. *Discrete mathematics.* Pearson, Prentice-Hall, 6th edition, 2005.