TP2 : SUITE DE FIBONACCI

Raphaël Alexandre

SUITE DE FIBONACCI

Une suite, $(F_n)_{n\in\mathbf{N}}$, de Fibonacci est décrite par la relation de récurrence :

$$\forall n \in \mathbf{N},\ F_{n+2} = F_{n+1} + F_n.$$

On prendra pour conditions initiales dans la suite :

$$F_0 = 0\ ;\ F_1 = 1.$$

1. ALGORITHME NAÏF

On écrit un premier algorithme naïf à partir de la définition de la suite par la relation de récurrence.

```
def fib1(n):
        if n <2:
                return n
        return fib1(n-1)+fib1(n-2)
```

Soit $n \in \mathbf{N}$, établissons le nombre d'appels, $A(n)$, à F_1 pour le calcul de F_n par cet algorithme :

— pour $n = 0$, $A(0) = 0$ et pour $n = 1$, $A(1) = 1$;

— soit $n \geq 2$ fixé, $A(n+2) = A(n+1) + A(n)$ d'après l'appel récursif.

C'est donc aussi une suite de Fibonacci et les conditions initiales sont encore une fois 0 et 1. On a donc un coût exponentiel selon n.

Si on s'intéresse au nombre d'appels, $A_k(n)$ à la valeur F_k pour $k < n$ fixé on a :

— $A_k(p) = 0$ pour $p < k$, $A_k(k) = 1$;

— $A_k(n+2) = A_k(n+1) + A_k(n)$.

On en déduit naturellement $A_k(n) = A(n-k+1)$ pour $n > k$ (on peut naturellement l'étendre à $n \geq k$).

Une mesure pour évaluer le temps d'exécution de l'algorithme pour $F(18)$ et $F(30)$ donne :

```
print time.clock()
print fib1(18)
print time.clock()
print fib1(30)
print time.clock()
```

```
air-de-raphael:Info Raphael$ python TP2.py
0.020602
2584
0.022091
832040
0.377324
```

On mesure donc un temps de 0.001489 secondes pour $F(18)$ et pour $F(30)$: 0.355233 secondes.

2. ALGORITHME LINÉAIRE

L'écriture de ce second algorithme repose sur « l'astuce » de garder en mémoire les termes déjà calculés. On fait donc suivre d'un appel récursif à l'autre la liste grandissante des $F_0, F_1, \ldots, F_k$ termes.

```
def fib2(n,liste = [0,1]):
        a = len(liste)
        if n == a-1 or n == a-2:
                return liste[n]
        liste.append(liste[a-1]+liste[a-2])
        return fib2(n,liste)
```

Le passage à la récursivité se fait en deux étapes : on allonge la liste avec le terme nouvellement calculé avec les deux derniers et on effectue l'appel récursif.

Calculons le nombre d'appels récursifs, $A(n)$:

— pour $n = 0, 1$ il est clair que le nombre d'appel est de 0 : le retour est immédiat ;

— pour $n > 1$, on a la relation : $A(n) = 1 + A(n-1)$.

On a clairement $A(n) = n-1$ pour $n > 1$ et $A(0) = A(1) = 0$. C'est bien une progression linéaire.

Mesurons le temps pour $F(18)$ et $F(30)$:

```
print time.clock()
print fib2(18)
print time.clock()
print fib2(30)
print time.clock()
```

```
air-de-raphael:Info Raphael$ python TP2.py
0.017101
2584
0.017151
832040
0.017162
```

Soit des durées de 0.000051 secondes et 0.000011 secondes. Cependant à ce niveau ce n'est plus très représentatif (selon les divers processus en exécutions parallèles).

3. ALGORITHME LOGARITHMIQUE

Pour ce dernier algorithme qui s'avèrera le plus efficace des trois, on utilise les relations suivantes :

$$\forall k \geq 1,\ F_{2k} = (2F_{k-1} + F_k)F_k$$
$$\forall k \geq 1,\ F_{2k+1} = F_{k+1}^2 + F_k^2$$

Un premier algorithme « naïf » avec ces informations supplémentaires serait :

```
def fib3(n):
        if n<2:
                return n
        fk = fib3(n/2)
        if n%2==0:
                return (2*fib3(n/2-1)+fk)*fk
        return fib3(n/2+1)**2+fk**2
```

Cependant les appels récursifs s'accumulent du fait de la nécessité d'au moins deux termes pour calculer. On utilise donc une autre astuce qui est de transmettre un couple (F_{k-1}, F_k) à chaque itération pour économiser ces doubles appels.

On construit donc l'algorithme :

```
def fib3Aux(n):
        if n<2:
                return (0,1)
        a = fib3Aux(n/2)
```

```
        if n%2==0:
                return (a[0]**2+a[1]**2,(2*a[0]+a[1])*a[1])
        b = a[0]+a[1]
        return ((2*a[0]+a[1])*a[1],b**2+a[1]**2)
```

Décrivons un peu plus la manière dont l'algorithme calcule le terme $F(n)$.

Soit $n \geq 2$ (le cas $n = 0$ ou 1 étant aisé à comprendre) :

1. On fait un appel récursif pour récupérer le couple $(F(n \div 2 - 1), F(n \div 2))$ où $n \div 2$ est la division entière de n par 2. Dans le code : a[0]= $F(n \div 2 - 1)$ (notons ce nombre x) et a[1]= $F(n \div 2)$ (notons ce nombre y).
2. Si n est pair, c'est-à-dire : $n = 2k$ avec $k \geq 1$. Alors $x = F(k-1)$ et $y = F(k)$. Par la relation on a :
$$(F(n-1), F(n)) = (F(2k-1), F(2k)) = (x^2 + y^2, (2x + y)y)$$
ce qui justifie cette partie du code.
3. Enfin, si n est impair, c'est-à-dire $n = 2k + 1$ avec $k \geq 1$. On a $x = F(k-1)$ et $y = F(k)$. Alors on calcule tout d'abord $z = x+y = F(k+1)$. On calcule ensuite :
$$(F(n-1), F(n)) = (F(2k), F(2k+1)) = ((2x + y)y, z^2 + y^2)$$
ce qui justifie la fin du code.

Remarquons que cet algorithme renvoie le couple $(F(n-1), F(n))$ ce qui demande donc un traitement final si l'on souhaite uniquement $F(n)$.

Comparons les deux derniers algorithme pour une grande valeur $n = 999$:

```
print time.clock()
print fib2(999)
print time.clock()
print fib3Aux(999)[1]
print time.clock()
```

```
air-de-raphael:Info Raphael$ python TP2.py
0.018006
26863810024485[...]
0.019236
26863810024485[...]
0.019283
```

Le second algorithme met 0.001230 secondes et le troisième 0.000047 secondes.