Índice General

Capítulo 3

Divide y vencerás

3.1 Cálculo de una función de Fibonacci generalizada

Dada la sucesión definida como $f_n = af_{n-3} + bf_{n-2} + cf_{n-1}$ se pide diseñar un algoritmo que calcule en tiempo logarítmico el término f_n. **Sugerencia:** Utilizad la siguiente relación:

$$\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ a & b & c \end{pmatrix} \begin{pmatrix} f_{n-3} \\ f_{n-2} \\ f_{n-1} \end{pmatrix} = \begin{pmatrix} f_{n-2} \\ f_{n-1} \\ f_n \end{pmatrix}$$

Elección razonada del esquema algorítmico

Cada término de la sucesión se calcula en función de los tres anteriores. Por tanto, la forma trivial de calcular f_n consiste en calcular los n términos anteriores, lo que supone un coste lineal $\mathcal{O}(n)$. Sin embargo, la ecuación que se nos proporciona nos da una forma más eficiente de resolver el problema, recurriendo a la técnica divide y vencerás. Efectivamente, si llamamos

$$F \equiv \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ a & b & c \end{pmatrix}$$

tendremos

$$\begin{pmatrix} f_{n-2} \\ f_{n-1} \\ f_n \end{pmatrix} = F \begin{pmatrix} f_{n-3} \\ f_{n-2} \\ f_{n-1} \end{pmatrix} = F^2 \begin{pmatrix} f_{n-4} \\ f_{n-3} \\ f_{n-2} \end{pmatrix} = \ldots = F^{n-2} \begin{pmatrix} f_0 \\ f_1 \\ f_2 \end{pmatrix}$$

donde f_0, f_1 y f_2 no pueden calcularse en términos de los elementos anteriores y han de ser, por tanto, datos del problema.[1]

Para calcular f_n es suficiente, por tanto, con evaluar F^n y hacer una multiplicación de una matriz de 3×3 por un vector de 3 elementos (que tiene coste constante). Sabemos que la forma más eficiente de calcular exponenciaciones es mediante la técnica divide y vencerás, aplicada mediante la igualdad:

$$F^n = (F^{n \text{ div } 2})^2 F^{n \bmod 2}$$

(es decir, $F^n = (F^{n/2})^2$ si n es par, y $F^n = F(F^{(n-1)/2})^2$ si n es impar).

Al reducir un problema de tamaño n a otro de tamaño $n/2$, conseguiremos una eficiencia $\mathcal{O}(\log n)$ frente a $\mathcal{O}(n)$, como comprobaremos al final.

Descripción del esquema e identificación con el problema

El esquema *divide y vencerás* es una técnica recursiva que consiste en dividir un problema en varios subproblemas del mismo tipo. Las soluciones a estos subproblemas se combinan a continuación para dar la solución al problema original. Cuando los subproblemas son más pequeños que un umbral prefijado, se resuelven mediante un algoritmo específico. Si su tamaño es mayor, se vuelven a descomponer. El esquema es el siguiente:

```
fun divide_y_venceras (problema)
    si suficientemente-simple (problema)
    entonces   dev solucion-simple (problema)
    si no   hacer
      { p_1 ... p_k } ← descomposicion(problema)
      para cada p_i hacer s_i ← divide_y_venceras(p_i) fpara
      dev combinacion(s_1 ... s_k)
```

[1] Por ejemplo, la sucesión de fibonacci es un caso particular de f con $a = b = 1$, $c = 0$ y $f_0 = 1, f_1 = 1$.

fsi
ffun

Los elementos de este esquema se particularizan así a nuestro caso:

Tamaño umbral y solución simple: El preorden bien fundado para los problemas se deriva directamente del orden total entre los exponentes (n) como números naturales. Podemos tomar como tamaño umbral n=1, caso en que la exponenciación es trivial y se devuelve como resultado la matriz de entrada.

Descomposición: F^n se descompone en un único subproblema, $F^{n/2}$, de la mitad de tamaño. Se trata, pues, de un problema de reducción más que de descomposición. El convertir un problema de tamaño n en otro de tamaño $n/2$ es lo que nos proporcionará un algoritmo eficiente.

Combinación:La función de combinación es la ecuación mencionada más arriba.

Estructuras de datos

En el problema intervienen solamente enteros y matrices de 3×3. Los datos de entrada y salida del algoritmo son enteros; las matrices se utilizan como resultados intermedios. Daremos por conocida la multiplicación de matrices.

Algoritmo completo

Suponemos conocidos a, b, c, f_0, f_1, f_2.

fun *f (n:entero)* ***dev*** *entero*

$$F \leftarrow \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ a & b & c \end{pmatrix}.$$

caso $n = 0$: ***dev*** f_0 ;
[] $n = 1$: ***dev*** f_1 ;
[] $n = 2$: ***dev*** f_2 ;
[] *verdadero* : ***hacer***

$$S \leftarrow exp\_mat(F, n-2)$$

$$s \leftarrow S \begin{pmatrix} f_0 \\ f_1 \\ f_2 \end{pmatrix}$$

dev $s[3]$

fcaso

```
fun exp_mat (M:vector[3,3]; n:entero) dev vector[3,3]
  si n ≤ 1
  entonces dev solucion_simple(M,n)
  si no hacer
   p ← n div 2
   r ← n mod 2
   T ← exp_mat(M,p)
     dev combinacion(T,r,M)
  fsi
ffun
```

En los argumentos de *combinacion*, sólo el primero es un subproblema. Los otros dos son parámetros auxiliares que utiliza la función de combinación.

```
fun combinacion (T:vector[3,3]; r:entero; M:vector[3,3]) dev vector[3,3]
  dev TT * M^r
ffun
```

donde $M^1 = M, M^0 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$.

```
fun solucion_simple (M:vector[3,3], n:entero) dev vector[3,3]
  si n = 1 dev M
  si no dev ( 1 0 0 )
            ( 0 1 0 )
            ( 0 0 1 )
  fsi
ffun
```

Estudio del coste

El coste del algoritmo recae sobre la exponenciación de la matriz F. Como se reduce un problema de tamaño n a otro de tamaño $n/2$, el coste cumple la ecuación de recurrencia $T(n) = T(n/2) + cte$, donde cte hace referencia al tiempo necesario para hacer una o dos multiplicaciones de matrices de 3×3. Utilizando

$$T(n) = \begin{cases} cn^k & , \text{si } 1 \leq n < b \\ aT(n/b) + cn^k & , \text{si } n \geq b \end{cases}$$

fsi
ffun

Los elementos de este esquema se particularizan así a nuestro caso:

Tamaño umbral y solución simple: El preorden bien fundado para los problemas se deriva directamente del orden total entre los exponentes (n) como números naturales. Podemos tomar como tamaño umbral n=1, caso en que la exponenciación es trivial y se devuelve como resultado la matriz de entrada.

Descomposición: F^n se descompone en un único subproblema, $F^{n/2}$, de la mitad de tamaño. Se trata, pues, de un problema de reducción más que de descomposición. El convertir un problema de tamaño n en otro de tamaño $n/2$ es lo que nos proporcionará un algoritmo eficiente.

Combinación:La función de combinación es la ecuación mencionada más arriba.

Estructuras de datos

En el problema intervienen solamente enteros y matrices de 3×3. Los datos de entrada y salida del algoritmo son enteros; las matrices se utilizan como resultados intermedios. Daremos por conocida la multiplicación de matrices.

Algoritmo completo

Suponemos conocidos a, b, c, f_0, f_1, f_2.

fun *f (n:entero)* ***dev*** *entero*
$F \leftarrow \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ a & b & c \end{pmatrix}$.
caso $n = 0$: ***dev*** f_0 ;
[] $n = 1$: ***dev*** f_1 ;
[] $n = 2$: ***dev*** f_2 ;
[] *verdadero* : ***hacer***
$S \leftarrow exp\_mat(F, n-2)$
$s \leftarrow S \begin{pmatrix} f_0 \\ f_1 \\ f_2 \end{pmatrix}$
dev $s[3]$
fcaso

```
fun exp_mat (M:vector[3,3]; n:entero) dev vector[3,3]
  si n ≤ 1
  entonces dev solucion_simple(M,n)
  si no hacer
  p ← n div 2
  r ← n mod 2
  T ← exp_mat(M,p)
    dev combinacion(T,r,M)
  fsi
ffun
```

En los argumentos de *combinacion*, sólo el primero es un subproblema. Los otros dos son parámetros auxiliares que utiliza la función de combinación.

```
fun combinacion (T:vector[3,3]; r:entero; M:vector[3,3]) dev vector[3,3]
  dev TT * M^r
ffun
```

donde $M^1 = M, M^0 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$.

```
fun solucion_simple (M:vector[3,3], n:entero) dev vector[3,3]
  si n = 1 dev M
  si no dev ( 1 0 0 )
            ( 0 1 0 )
            ( 0 0 1 )
  fsi
ffun
```

Estudio del coste

El coste del algoritmo recae sobre la exponenciación de la matriz F. Como se reduce un problema de tamaño n a otro de tamaño $n/2$, el coste cumple la ecuación de recurrencia $T(n) = T(n/2) + cte$, donde cte hace referencia al tiempo necesario para hacer una o dos multiplicaciones de matrices de 3×3. Utilizando

$$T(n) = \begin{cases} cn^k & , \text{si } 1 \leq n < b \\ aT(n/b) + cn^k & , \text{si } n \geq b \end{cases}$$

$$\Rightarrow$$

$$T(n) \in \begin{cases} \Theta(n^k) & , \text{si } a < b^k \\ \Theta(n^k \log n) & , \text{si } a = b^k \\ \Theta(n^{\log_b a}) & , \text{si } a > b^k \end{cases}$$

con $k = 0, b = 2, a = 1$, estamos en el caso $a = b^k$ y, por lo tanto, el coste del algoritmo es $\Theta(\log n)$. Una implementación directa del cálculo de la sucesión hubiera necesitado un tiempo lineal con $\mathcal{O}(n)$.