

Programación Dinámica

Andrés Becerra Sandoval

3 de septiembre de 2007

Resumen

Esta es una técnica asombrosa que permite diseñar algoritmos muy eficientes para problemas de optimización, que, a primera vista, parecen requerir algoritmos de orden exponencial para su solución.

Dividir y Conquistar

Esta técnica de diseño de algoritmos se basa en:

- Dividir un problema en subproblemas del mismo tipo
- Solucionar cada subproblema recursivamente
- Combinar las soluciones a los subproblemas para obtener la solución global (conquistar)

Ejemplos de ella son los algoritmos:

Quicksort

- Divide el arreglo de entrada en dos subarreglos a la izquierda y a la derecha del pivote (partition)
- Ordena recursivamente las dos mitades

MergeSort

- Divide el arreglo de entrada de tamaño en dos subarreglos de tamaño aproximado $n/2$
- Ordena recursivamente las dos mitades
- Mezcla las dos mitades, ya ordenadas, para obtener el resultado

Nuestro objetivo en ésta lectura es construir otra técnica de diseño basada en dividir y conquistar: la programación dinámica.

1. Motivación

El factorial de un número puede calcularse recursivamente:

```
FACT( $n$ )
  if  $n = 0$ 
    then return 1
    else return  $n \times \text{fact}(n - 1)$ 
```

Con una complejidad dada por la recurrencia $T(n) = T(n - 1) + \Theta(1)$ que tiene como solución $\Theta(n)$. Verifiquelo!

Los números de fibonacci también pueden calcularse recursivamente siguiendo una estrategia de *dividir y conquistar*:

```
FIB( $n$ )
  if  $n = 0$  or  $n = 1$ 
    then return 1
    else return FIB( $n - 1$ ) + FIB( $n - 2$ )
```

Pero la ecuación de recurrencia $T(n) = T(n - 1) + T(n - 2) + \Theta(1)$ tiene una solución exponencial $\Theta(\phi^n)$. Verifiquelo!

Una forma de hacerlo mejor consiste en llevar una tabla de los números fibonacci ya calculados para no repetir trabajo:

FIB(n)

▷ PRE: $C[0]$ y $C[1]$ contienen 1, $C[k] = \infty$ para $k \geq 2$

if $C[n] \neq \infty$

then return $C[n]$

else $fibn \leftarrow FIB(n-1) + FIB(n-2)$

$C[n] \leftarrow fibn$

return $fibn$

Este cambio, aunque pequeño, invalida la recurrencia $T(n) = T(n-1) + T(n-2) + \Theta(1)$, porque los llamados recursivos a FIB pueden tardar tiempo constante si los resultados ya se han calculado previamente. Considere como transcurre la ejecución para un llamado como FIB(5), grafique los llamados recursivos y descubrirá que el tiempo de ejecución se ha disminuido ahora, con respecto al tiempo de ejecución de la primera versión. Esta técnica de almacenar soluciones previamente calculadas para evitar repetir trabajo recibe el nombre de *memoization*.

Cuando un algoritmo construido mediante dividir y conquistar utiliza una arreglo o tabla para evitar repetir trabajo, decimos que estamos aplicando *programación dinámica*.

2. Un ejemplo mas elaborado: $\binom{n}{r}$

El número de combinaciones puede definirse recursivamente. Para escoger r items de n posibles:

- Se escoge el primer item, y luego los $r-1$ restantes de $n-1$ posibles
- Ó, no se escoge el primer item. Entonces debemos escoger los r items de $n-1$ posibles (todos, menos el primero)

Esto se puede expresar así:

$$\binom{n}{r} = \binom{n-1}{r-1} + \binom{n-1}{r}$$

Lo que nos permite rápidamente diseñar un algoritmo recursivo basado en dividir y conquistar que calcule el número de combinaciones:

```

COMB( $n, r$ )
  if  $r = 0$  or  $n = r$ 
    then return 1
  else return COMB( $n-1, r-1$ )+COMB( $n-1, r$ )

```

Que tiene la ecuación de recurrencia $T(n) = 2T(n-1) + \Theta(1)$, cuya solución es $\Theta(2^n)$. Demuestre que es así iterando la recurrencia 3 veces con un árbol de recursión que le permita adivinar la forma de la solución.

El genial Blaise Pascal se inventó una forma extremadamente eficiente de calcular el número de combinaciones mediante programación dinámica (sin siquiera existir la programación dinámica, ni siquiera la programación, ni los computadores!). Hoy en día podemos decir que lo que hizo Pascal es una aplicación de programación dinámica, *el triángulo de Pascal*. Primero añadamos una tabla para almacenar las soluciones $\binom{n}{r}$:

```

COMB( $n, r, C$ )
  ▷ PRE:  $C[k, 1] = C[k, k] = 1$  para todo  $k$ ,  $C[k, l] = \infty$  para todo  $l \neq 1, l \neq k$ 
  if  $C[n, r] \neq \infty$ 
    then return  $C[n, r]$ 
  else  $comb\_nr \leftarrow$  COMB( $n-1, r-1$ ) + COMB( $n-1, r$ )
     $C[n, r] = comb\_nr$ 
  return  $comb\_nr$ 

```

Observe que la ecuación de recurrencia también se invalida aquí porque en algunos casos COMB(n, r, C) va a calcular su resultado en tiempo constante.

3. Obteniendo algoritmos iterativos

Hasta este punto parece que hemos obtenido dos algoritmos para calcular números de fibonacci y números de combinaciones de una forma mas eficiente (aunque esto solo es una sospecha) que las versiones recursivas por medio del uso de un arreglo y una tabla o matriz que guardan valores previamente calculados. La idea consiste en que cada vez que se calcule un valor se almacene en la tabla de forma que cualquier llamado posterior que necesite el valor sea $\Theta(1)$.

Sin embargo, los algoritmos recursivos que hemos obtenido son muy difíciles de analizar, así que no lo haremos. El curso de acción mas sencillo consiste en transformarlos en algoritmos iterativos, de esta forma:

- obtenemos algoritmos mas fáciles de analizar
- podemos ganar eficiencia, al liberarnos de la memoria que se gasta en la recursión

Para transformar estos algoritmos recursivos a iterativos es conveniente comenzar con las definiciones recursivas. Primero la de los números de fibonacci:

$$F(n) = F(n-1) + F(n-2)$$

Esta nos permite ver que el número de fibonacci depende unicamente de los dos números anteriores, lo que nos permite escribir un simple ciclo que va llenando el arreglo con los fibonaccis calculados en orden:

```
FIB(n)
  C[0] ← 1
  C[1] ← 1
  for j ← 2 to n
    do
      C[j] ← C[j-1] + C[j-2]
  return C[n]
```

Lo que se puede hacer mas eficiente aún al notar que no necesitamos ocupar todo un arreglo, solo 2 variables temporales:

```
FIB(n)
  fn-2 ← 1
  fn-1 ← 1
  for j ← 2 to n
    do
      fn ← fn-1 + fn-2
      fn-2 ← fn-1
      fn-1 ← fn
  return fn
```

En donde podemos hacer un sencillo análisis de complejidad temporal ($\Theta(n)$), y espacial ($\Theta(1)$).

Regresemos a la definición recursiva del número de combinaciones:

$$\binom{n}{r} = \binom{n-1}{r-1} + \binom{n-1}{r}$$

Para obtener $\binom{n}{r}$ podemos pensar en llenar una matriz con n filas y r columnas de una forma iterativa que respete la recurrencia anterior. La situación puede visualizarse así para el rincón *mas inferior y mas a la derecha* de la matriz:

$$\begin{array}{ccc} & r-1 & r \\ n-1 & \square & \square \\ n & & \square \end{array}$$

La casilla (n,r) (de la fila n, columna r) depende de las casillas (n-1,r-1) y (n-1,r). Si generalizamos a toda la matriz, la situación sería:

$$\begin{array}{ccc} & j-1 & j \\ i-1 & \square & \square \\ i & & \square \end{array}$$

Para cualesquier i,j que indiquen entradas válidas de la matriz.

Esta observación nos permite escribir un algoritmo iterativo que llene las entradas de la matriz respetando la situación anterior. *Una forma de hacerlo* consiste en llenar las columnas de izquierda a derecha, y los elementos de cada columna de arriba hacia abajo, por ejemplo, el orden parcial de llenado de las casillas para n=10, r=5 sería:

1	11
2	12	⋮		
3	13			
4	14			
5	15			
6	16			
7	17			
8	18			
9	19			
10	20			

Este orden nos sugiere el siguiente algoritmo:

```

COMB( $n, r$ )
1  for  $i \leftarrow 0$  to  $n$ 
2      do  $C[i, 0] \leftarrow 1$ 
3  for  $j \leftarrow 1$  to  $r$ 
4      do for  $i \leftarrow 0$  to  $n$ 
5          do if  $i = j$ 
6              then  $C[i, j] \leftarrow 1$ 
7              else  $C[i, j] \leftarrow C[i - 1, j - 1] + C[i - 1, j]$ 
8  return  $C[n, r]$ 

```

Que luce mucho mas fácil de analizar que el recursivo!. Y además, permite realizar algunas mejoras. Para comprenderlo observe que en las líneas 1 a 3 se aprovecha la identidad $\binom{n}{0} = 1$, en las líneas 5 a 6 se utiliza la otra identidad $\binom{n}{n} = 1$ que permiten llenar la primera columna de la tabla, y la diagonal principal. Los dos ciclos for hacen un recorrido por columnas de izquierda a derecha, y en cada columna se recorren los elementos de arriba a abajo, respetando la recurrencia para el número de combinaciones. Observe que en este problema de llenado de la matriz si nos conviene contar las posiciones de la matriz desde cero.

Para mejorar el algoritmo podemos observar que no es necesario llenar toda la matriz. Lo único que nos interesa para calcular $\binom{n}{r}$ es llenar la casilla (n, r) y sus dependencias. El algoritmo recursivo que desarrollamos en la sección anterior es muy eficiente desde el punto de vista de ahorro de espacio, solo llena las casillas de la matriz que se necesitan llenar mediante la magia de la recursión. Nuestro algoritmo iterativo podría mejorarse para llenar menos casillas:

```

COMB( $n, r$ )
1  for  $i \leftarrow 0$  to  $n$ 
2      do  $C[i, 0] \leftarrow 1$ 
3  for  $i \leftarrow 0$  to  $n$ 
4      do  $C[i, i] \leftarrow 1$ 
5  for  $j \leftarrow 1$  to  $r$ 
6      do for  $i \leftarrow j + 1$  to  $n$ 
7          do  $C[i, j] \leftarrow C[i - 1, j - 1] + C[i - 1, j]$ 
8  return  $C[n, r]$ 

```

Ahora, en las líneas 3 a 4 se llena la diagonal principal. Luego se hace un recorrido por columnas de izquierda a derecha, y en cada columna se recorren los

elementos de arriba a abajo, empezando desde la posición $j+1$, esto es, siempre por debajo de la diagonal principal, que ya había sido calculada.

Con ésta mejora el orden parcial de llenado de las casillas *en los ciclos for anidados* (líneas 5 a la 7) luce así:

1				
1	1			
1	1	1		
1	2	11	1	
1	3	12		1
1	4	13		
1	5	14		
1	6	15		
1	7	16		
1	8	17		

Note que la primera columna ya había sido llenada previamente, al igual que la diagonal principal. Entendiendo éste orden podemos pensar con que valores reales se va llenando la tabla mediante el último algoritmo. Recuerde que el valor de cada casilla (p. ej. 21) es la suma de la casilla al noroccidente (p. ej. 6) y de la casilla al norte (p. ej. 15).

1				
1	1			
1	2	1		
1	3	3	1	
1	4	6	4	1
1	5	10	10	5
1	6	15	20	15
1	7	21	35	35
1	8	28	56	70
1	9	36	84	126

Y *voilà*, tenemos el triángulo de Pascal, con una complejidad espacial y temporal $O(n^2)$ como usted puede chequear. Una mejora adicional consiste en llenar solo $n-r$ números en cada columna:

```

COMB( $n, r$ )
1  for  $i \leftarrow 0$  to  $n$ 
2      do  $C[i, 0] \leftarrow 1$ 
3  for  $i \leftarrow 0$  to  $n$ 
4      do  $C[i, i] \leftarrow 1$ 
5  for  $j \leftarrow 1$  to  $r$ 
6      do for  $i \leftarrow j + 1$  to  $n - r + j$ 
7          do  $C[i, j] \leftarrow C[i - 1, j - 1] + C[i - 1, j]$ 
8  return  $C[n, r]$ 

```

Lo que causaría el siguiente comportamiento:

1				
1	1			
1	2	1		
1	3	3	1	
1	4	6	4	1
1	5	10	10	5
1	6	15	20	15
1		21	35	35
1			56	70
1				126

La pregunta es, ¿porqué puede ahorrarse el llenado de estas casillas?

¿Puede usted decir cuales son las casillas que hay que llenar obligatoriamente tachandolas sobre la tabla anterior?

4. Descripción *metodológica*

Para resolver un problema por medio de programación dinámica es conveniente seguir la siguiente secuencia:

- Diseñar un algoritmo recursivo que divida (en subproblemas) y conquiste (construya la solución general).
- Si la complejidad de éste algoritmo es muy alta como en el cálculo de números de fibonacci y número de combinaciones, entonces:

- Agregue una tabla que almacene resultados a subproblemas
- Modifique el algoritmo para que consulte la tabla antes de realizar el trabajo recursivo, y para que vaya llenando la tabla
- Si el algoritmo obtenido en el punto anterior es muy difícil de analizar, conviértalo a iterativo observando la definición recursiva de la solución para determinar que dependencias deben respetarse al llenar la tabla iterativamente.

En este punto es conveniente que lea el capítulo 15 de CLRS [1], para adquirir un punto de vista más formal sobre la programación dinámica. Siga en detalle la solución de los problemas de optimización:

1. Longest common subsequence
2. Matrix-chain multiplication
3. Optimal binary search trees
4. Assembly-line scheduling

A mi me parece conveniente el orden sugerido, aunque en el libro las soluciones a estos problemas se presentan en otra secuencia.

Referencias

- [1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms*. MIT Press, Cambridge, MA, USA, 2001.