Resolución de Recurrencias Lineales en Análisis de Algoritmos

Juan G. Lalinde-Pulido
jlalinde@eafit.edu.co

Alberto Restrepo Vásquez
arestrep@eafit.edu.co

Departamento de Informática y Sistemas
Universidad EAFIT
Medellín - Colombia

29 de marzo de 2001

Resumen

Uno de los principales problemas en análisis de algoritmos surge al tratar de determinar la complejidad de un algoritmo recursivo. Para llevar a cabo esta tarea se formula la ecuación de recurrencia asociada con el algoritmo. Dicha ecuación se puede solucionar al realizar la expansión de la recurrencia [2] o mediante la técnica de resolución de recurrencias lineales [1].

1 Introducción

El objetivo principal del análisis de algoritmos es proporcionar una métrica objetiva que permita comparar dos algoritmos para determinar cual de ellos es mejor. Para llevar a cabo su tarea, se debe definir claramente cuando dos algoritmos son comparables y la naturaleza de la métrica.

En primer lugar, decimos que dos algoritmos son comparables cuando son indistinguibles en una prueba de caja negra. La prueba de caja negra consiste en presentarle entradas al algoritmo y ver que produce a la salida. Con ella se evalua *qué* hace el algoritmo sin preocuparse por el *cómo* lo hace. Dos algoritmos son indistinguibles en una prueba de caja negra cuando para toda entrada x la respuesta y de ambos algoritmos es identica.

En cuanto a la métrica, se define de manera que permita minimizar los recursos utilizados para resolver el problema. En computación hay dos recursos que son escasos: memoria (espacio) y tiempo. Se puede utilizar indistintamente el consumo de uno de estos recursos como métrica para evaluar la calidad del algoritmo.

En la práctica el recurso tiempo es mucho más valioso que el recurso espacio. Por esta razón lo convencional es utilizar como métrica la complejidad temporal

de una aplicación. Sin embargo, hay que resaltar que cuando el costo del recurso memoria sea más significativo que el del tiempo, se debe utilizar la complejidad espacial como criterio de comparación.

Para efectos del análisis de un algoritmo se define la complejidad como la relación que existe entre el consumo de recursos y el número de datos que se procesan. En sentido estricto, la complejidad define el consumo de recursos como una función del número de datos a procesar. Así, la complejidad temporal se define como la relación que existe entre el tiempo de ejecución de un algoritmo y el número de datos que procesa. Por otra parte la complejidad espacial se define como la relación que existe entre la cantidad de memoria necesaria para la ejecución del algoritmo y el número de datos que procesa.

Si bien la complejidad usualmente se calcula con respecto al número de datos que se procesan, hay problemas en los cuales la variación no se da con respecto al número de datos sino con respecto al valor númerico del dato a procesar. Un ejemplo de este tipo de apliaciones es la factorización de un número. Las funciones utilizadas en criptografía presentan, generalmente, este tipo de comportamiento.

En el resto del artículo nos vamos a referir a la complejidad temporal de un algoritmo, si bien las técnicas desarrolladas son aplicables también al estudio de la complejidad espacial. La función que relaciona el tiempo de ejecución del algortimo con el número de datos la denominaremos $T(n)$, donde n es el número de datos a procesar.

2 La notación O

Como se dijo anteriormente, el análisis de algoritmos busca una métrica objetiva para comparar algoritmos. Sean f y g algoritmos comparables entre si y sean $T_f(n)$ y $T_g(n)$ sus complejidades temporales respectivas. Cómo pueden compararse entre sí? El criterio va a ser el comportamiento de $T(n)$ cuando n es muy grande. Es decir, $\lim_{n\to\infty} T(n)$.

Tal como se ha visto hasta el momento, para cada algoritmo va a existir una función $T(n)$ particular. Con el fin de construir un cuerpo teórico que pueda ser util, es necesario introducir algún tipo de notación que permita agrupar todos aquellos algoritmos que tienen comportamiento similar. Esta es la función de la notación O.

Definición 1 *Se dice que $T_f(n) = O(T_g(n))$ si existen constantes n_o y c tales que $T_f(n) \leq cT_g(n)$ para todo $n > n_o$. En este caso decimos que $T_f(n)$ es de orden $T_g(n)$.*

Informalmente, un algoritmo es de orden $O(T(n))$ si el tiempo requerido para la ejecución del mismo crece proporcionalmente a $T(n)$. También es importante tener presente que la notación de orden define como es el comportamiento asintóticamente.

Algunas propiedades interesantes son:

- Suma: $\sum_{j=1}^{n} O(T(n_j)) = \max_{j=1}^{n}(O(T(n_j)))$
- Multiplicación: $O(T(n_1)T(n_2) = O(T(n_1))O(T(n_2))$

Para simplificar $O(T(n))$ será expresado en términos de funciones simples, de manera que el tiempo constante se dirá que es $O(1)$, el lineal es $O(n)$, etc.

3 Calculando la complejidad de un algoritmo simple

Cuando se desea calcular la complejidad de un algoritmo, hay que identificar la complejidad asociada con cada uno de sus componentes. El primer paso para construir la ecuación de recurrencia de un algoritmo es descomponerlo en sus componentes fundamentales.

La programación estructurada dice que todo algoritmo puede ser reescrito de manera que sólo utilice tres estructuras de control básicas: las secuencias, los ciclos y las decisiones. A cada una de ellas podemos asociar una complejidad así:

- Secuencias: Se calcula como la suma de la complejidad de cada paso. Si el paso involucra una llamada a una función su complejidad es la misma de la función invocada. De lo contrario, su complejidad es constante.
- Decisiones: Se analiza por aparte el caso cuando la condición es verdadera del caso cuando la condición es falsa. Se toma como complejidad el máximo de las dos complejidades.
- Ciclos: Se calcula la complejidad del bloque interno. La complejidad del ciclo es el resultado de multiplicar el número de iteraciones por la complejidad interna.

Cuando se realiza el análisis de un algoritmo en el cual estas estructuras están anidadas, se debe realizar el análisis de adentro hacia afuera.

Analicemos un ejemplo sencillo. El algoritmo 1 presenta el algoritmo clásico conocido como *sort* burbuja. Este algoritmo recibe como entrada un vector de n elementos todos del mismo tipo. La única condición es que para ese tipo de dato debe existir una relación de orden total que permita comparar dos datos y ordenarlos.

De acuerdo con lo planteado, los pasos 4, 7, 8, 9, 11 y 13 tienen un tiempo de ejecución constante, por lo que cada uno de ellos es $O(1)$. Aplicando la propiedad de la suma se obtiene que la complejidad del bloque conformado por los pasos 7, 8 y 9 es $O(1)$, pues $\max(O(1), O(1), O(1)) = O(1)$.

La condición que abarca las líneas 6 a 10 tiene la misma complejidad del bloque interior, asi que su complejidad es $O(1)$. La complejidad del ciclo que va desde la línea 5 a la 12 se obtiene calculando primero la complejidad del bloque interno y luego multiplicandola por el número de veces que se repite el ciclo. En

Algoritmo 1 *Sort* burbuja - v es el vector y tiene n elementos

```
1:  SortBurbuja(v, n)
2:  i ← 1
3:  while i ≤ n − 1 do
4:     j ← 1
5:     while j ≤ n − i do
6:        if v[j − 1] > v[j] then
7:           t ← v[j − 1]
8:           v[j − 1] ← v[j]
9:           v[j] ← t
10:       end if
11:       j ← j + 1
12:    end while
13:    i ← i + 1
14: end while
```

este caso, la regla de la suma indica que la complejidad del bloque es $O(1)$ y el ciclo se repite $n - i$, así que la complejidad total del ciclo es $O(n - i)$.

Aplicando el mismo análisis al ciclo que abarca de la línea 3 a la 14 se tiene que el bloque interno tiene orden $O(n - i)$. El valor de i es diferente para cada iteración del ciclo. Cuando $i = 1$ se repite $n-1$, pero cuando $i = n-1$ se repite 1 vez. Esto se expresa como

$$\sum_{i=1}^{n-1}(n-i) = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} = O(n^2)$$

Por lo tanto se concluye que el algoritmo de *sort* burbuja es de orden $O(n^2)$.

4 Ecuación de la recurrencia

La ecuación de recurrencia surge cuando se va a realizar el análisis de un algoritmo que es recursivo. El principal problema que surge en estos casos es que al analizar el algoritmo, la complejidad de la función se expresa en términos de ella misma: es recursiva. Con el fin de poder obtener la complejidad real, hay que resolver esa recurrencia.

Un ejemplo clásico es el factorial, el cual se define recursivamente así:

$$n! = \begin{cases} 1 & \text{Si } n = 0 \\ n(n-1)! & \text{Si } n > 1 \end{cases}$$

El algoritmo 2 presenta el pseudo código para este problema. Debido a que el algoritmo es recursivo, para calcular su complejidad hay que generar la ecuación de recurrencia correspondiente. Para poder determinar el orden del algoritmo, hay que resolverla.

El algoritmo 2 contiene una decisión. La ecuación de recurrencia del algoritmo es

$$T(n) = \begin{cases} 1 & \text{Si } n = 0 \\ 1 + T(n-1) & \text{Si } n > 0 \end{cases}$$

La expresión $1 + T(n-1)$ surge del hecho de que en realidad son dos instrucciones. La primera consiste en invocar la función y la segunda en realizar la multiplicación.

Algoritmo 2 Factorial - Calcula el factorial de n

```
1: factorial(n)
2: if n = 0 then
3:     return 1
4: else
5:     return n × factorial(n − 1)
6: end if
```

Podemos resolver fácilmente la ecuación de recurrencia 4. Sabemos que $T(n) = 1 + T(n-1)$ por lo que $T(n-k) = 1 + T(n-k-1)$. Realizando sucesivos reemplazos tenemos:

$$\begin{aligned} T(n) &= 1 + T(n-1) \\ &= 2 + T(n-2) \\ &= 3 + T(n-3) \\ &= 4 + T(n-4) \\ &\vdots \\ &= n + T(0) \\ &= n + 1 \end{aligned}$$

Por lo tanto podemos concluir que el orden del algoritmo para calcular el factorial se $O(n)$.

Utilicemos esta técnica para analizar la búsqueda binaria. Se parte de una colección organizada de elementos y un elemento a buscar. Se compara el elemento a buscar con el que está en la mitad de la colección de acerdo con el orden establecido. Si es igual se declara que se encontro. Si es menor, se descarta la mitad compuesta por elementos mayores y se aplica recursivamente a la mitad compuesta por elementos menores. Si es mayor, se descarta la mitad compuesta por elementos menores y se aplica recursivamente a la mitad compuesta por elementos mayores.

El algoritmo 3 presenta el pseudocódigo para esta función. Está escrito de manera recursiva para facilitar la identificación de la recurrencia. Lo normal es que los algoritmos no se escriban recursivamente por el consumo de recursos que tienen, sin embargo, esto facilita la comprensión de la recurrencia.

Algoritmo 3 Búsqueda Binaria - Encuentra el elemento a en el conjunto X que contiene n elementos, donde l es el indice del primer elemento y u es el índice del último

```
1: BúsquedaBinaria(X, l, u, a)
2: if l > u then
3:    return false {El elemento no está en el conjunto}
4: end if
5: m ← (l + u)/2 {Elemento ubicado en la mitad del conjunto de acuerdo con
   la relación de orden}
6: if X[m] = a then
7:    {X[m] se refiere al m-ésimo elemento del conjunto de acuerdo con la
      relación de orden.}
8:    return true {El elemento está en la posición m}
9: else if X[m] < a then
10:   return BúsquedaBinaria(X, m + 1, u, a)
11: else
12:   {X[m] > a}
13:   return BúsquedaBinaria(X, l, m − 1, a)
14: end if
```

De el algoritmo se deduce la siguiente relación de recurrencia

$$T(n) = \begin{cases} 1 & \text{Si } n = 0 \\ 1 + T(n/2) & \text{Si } n > 0 \end{cases}$$

Sabemos que $T(n) = 1 + T(n/2)$ por lo que $T(n-k) = 1 + T((n-k-1)/2)$. Realizando sucesivos reemplazos tenemos:

$$\begin{aligned} T(n) &= 1 + T(\lfloor n/2 \rfloor) \\ &= 2 + T(\lfloor n/4 \rfloor) \\ &= 3 + T(\lfloor n/8 \rfloor) \\ &= 4 + T(\lfloor n/16 \rfloor) \\ &\vdots \\ &= log_2 n + T(0) \\ &= \end{aligned}$$

Por lo tanto podemos concluir que el orden del algoritmo para la busqueda binaria se $O(log_2 n)$.

5 Resolución de recurrencias lineales

La técnica anterior es muy util, sin embargo no es aplicable en todos los casos. En particular, es muy dificil cuando se tiene que la recursividad es múltiple.

Veamos un ejemplo. Sea un algoritmo cuya ecuación de recurrencia es de la forma

$$T(n) = \begin{cases} 1 & \text{Si } n = 0 \wedge n = 1 \\ T(n-1) + T(n-2) & \text{Si } n > 1 \end{cases}$$

Un algortimo de que tiene esta ecuación de recurrencia es la formulación recursiva de la sucesión de Fibonnaci. Sin embargo, como veremos más adelante, para dicha sucesión existe una formula general que es precisamente el orden de los algoritmos que tienen esta formula de recurrencia.

Realizemos el analisis utilizando el procedimiento visto anteriormente:

$$\begin{aligned} T(n) &= 1 + T(n-1) + T(n-2) \\ &= 2 + 2T(n-2) + T(n-3) \\ &= 4 + 3T(n-3) + 2T(n-4) \\ &= 7 + 5T(n-4) + 3T(n-5) \\ &= 12 + 8T(n-5) + 5T(n-6) \\ &= 20 + 13T(n-6) + 8T(n-7) \\ &\vdots \end{aligned}$$

donde se aprecia claramente que los coeficientes son a su vez elementos de la sucesión de fibonnaci. Por esta razón, es imposible utilizar esta técnica para resolver la recurrencia de fibonnaci.Con el fin de poder resolver esta recurrencia, presentaremos el concepto de las recurrencias lineales con coeficientes constantes.

Una *recurrencia lineal con coeficientes constantes* tiene la forma

$$c_0 a_n + c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_k a_{n-k} = f(n) \tag{1}$$

para alguna constante k donde los c_i son todos constantes. Se puede resolver dicha recurrencia para un gran conjunto de funciones f utilizando el *método de los operadores*.

En primer lugar definimos dos operadores sobre secuencias: $\mathcal{S}$ y $\mathcal{C}$. El operador $\mathcal{S}$ desplaza una secuencia una posición a la derecha:

$$\mathcal{S}\langle a_0, a_1, a_2, \cdots\rangle = \langle a_1, a_2, a_3, \cdots\rangle$$

Por su parte el operador $\mathcal{C}$, donde $\mathcal{C}$ es una constante cualquiera, multiplica todos los términos de la secuencia por $\mathcal{C}$:

$$\mathcal{C}\langle a_0, a_1, a_2, \cdots\rangle = \langle \mathcal{C}a_0, \mathcal{C}a_1, \mathcal{C}a_2, \cdots\rangle$$

Dados los operadores A y B definimos la suma y el producto como

$$(A+B)\langle a_0, a_1, a_2, \cdots\rangle = A\langle a_0, a_1, a_2, \cdots\rangle + B\langle a_0, a_1, a_2, \cdots\rangle$$

y

$$(AB)\langle a_0, a_1, a_2, \cdots\rangle = A(B\langle a_0, a_1, a_2, \cdots\rangle)$$

La ecuación (1) puede ser reescrita como

$$P(\mathcal{S})\langle a_i\rangle = \langle f(i)\rangle$$

donde

$$P(\mathcal{S}) = c_0\mathcal{S}^k + c_1\mathcal{S}^{k-1} + c_2\mathcal{S}^{k-2} + \cdots + c_k$$

Esto es debido a la forma como se definió el producto de operadores, pues no es realmente una multiplicación sino una composición. Así, la expresión $\mathcal{S}^2$ es realmente aplicar dos veces el operador $\mathcal{S}$. Un ejemplo:

$$\begin{aligned}(\mathcal{S}^2 - 4)\langle a_i\rangle &= \mathcal{S}^2\langle a_i\rangle - 4\langle a_i\rangle \\ &= \mathcal{S}(\mathcal{S}\langle a_i\rangle) - \langle 4a_i\rangle \\ &= \mathcal{S}\langle a_{i+1}\rangle - \langle 4a_i\rangle \\ &= \langle a_{i+2}\rangle - \langle 4a_i\rangle \\ &= \langle a_{i+2} - 4a_i\rangle\end{aligned}$$

Dada una secuencia $\langle a_i\rangle$, se dice que el operador $P(\mathcal{S})$ anula a $\langle a_i\rangle$ si $(P(\mathcal{S})\langle a_i\rangle = \langle 0\rangle$. En general el operador $(\mathcal{S} - c)^{k+1}$ anula la secuencia $\langle c^i P(i)\rangle$, donde $P(i) = a_0 i^k + a_1 i^{k-1} + a_2 i^{k-2} + \cdots + a_k$ es un polinomio en i de grado k. Adicionalmente, si A anula a $\langle a_i\rangle$ y B anula a $\langle b_i\rangle$, entonces AB anula a $\langle a_i + b_i\rangle$. Para mayores detalles, consultar la solución de ecuaciones de diferencias homogéneas.

El método para resolver una recurrencia lineal es el siguiente:

- Formule la ecuación característica de la relación de recurrencia.
- Encuentre las raices de la ecuación característica.
- Encuentre las secuencias que son anuladas por cada una de las raices.
- La solución general de la relación de recurrencia es la suma de dichas secuencias.
- Utilice las condiciones iniciales para resolver el problema particular.

Con el fin de clarificar los pasos, vamos a utilizar como ejemplo una relación de recurrencia que se define como los números de fibonnaci.

Ecuación característica Para formular la ecuación caraceterística se escribe la relación de recurrencia y luego se reemplaza $T(n)$ por $\mathcal{S}^n$. En el caso que vamos a analizar tenemos que la relación de recurrencia es

$$T(n) = \begin{cases} 1 & \text{Si } n = 0 \wedge n = 1 \\ T(n-1) + T(n-2) & \text{Si } n > 1 \end{cases}$$

y por lo tanto la ecuación característica es

$$\mathcal{S}^n = \mathcal{S}^{n-1} + \mathcal{S}^{n-2}$$

que se puede simplificar como

$$\mathcal{S}^2 - \mathcal{S}^1 - 1 = 0 \tag{2}$$

Encontrar las raices La ecuación característica es un polinomio, así que se utilizan las técnicas tradicionales de factorización para factorizar el polinomio y hallar las raices. En el caso que nos ocupa, tenemos una ecuación cuadrática, así que la solución es

$$\begin{aligned} \mathcal{S} &= \frac{1 \pm \sqrt{1+4}}{2} \\ &= \frac{1 \pm \sqrt{5}}{2} \end{aligned}$$

Encuentre las secuencias anuladas Cada una de las raices encontradas es de la forma $(\mathcal{S} - c)^{k+1}$, así que conocemos la forma de las secuencias que son anuladas. En el caso que nos ocupa, las raices son de la forma $\mathcal{S} - c$ y son de multiplicidad 1. Sean $\phi_1 = \frac{1+\sqrt{5}}{2}$ y $\phi_2 = \frac{1-\sqrt{5}}{2}$, por lo tanto

$$\begin{aligned} (\mathcal{S} - \phi_1)\langle u\phi_1^i \rangle &= \langle 0 \rangle \\ (\mathcal{S} - \phi_2)\langle v\phi_2^i \rangle &= \langle 0 \rangle \end{aligned}$$

Sume las secuencias encontradas Como se vio anteriormente, si A anula a $\langle a_i \rangle$ y B anula a $\langle b_i \rangle$, entonces AB anula a $\langle a_i + b_i \rangle$. En el caso analizado se tiene que

$$T(n) = u\phi_1^n + v\phi_2^n$$

Utilice las condiciones iniciales Como en la formulación de la relación de recurrencia se incluyen las condiciones iniciales, se utilizan dichas condiciones para determinar cual es el valor de las constantes. En este caso, sabemos que $T(0) = 1$ y $T(1) = 1$ por lo que se tiene

$$\begin{aligned} 1 &= T(0) \\ &= u\phi_1^0 + v\phi_2^0 \\ &= u + v \end{aligned} \tag{3}$$

y

$$\begin{aligned} 1 &= T(1) \\ &= u\phi_1^1 + v\phi_2^1 \\ &= u\phi_1 + v\phi_2 \end{aligned} \tag{4}$$

de donde $u = v = 1/\sqrt{5}$ y por lo tanto tenemos que

$$T(n) = \frac{\frac{1+\sqrt{5}}{2}^n - \frac{1-\sqrt{5}}{2}^n}{\sqrt{5}}$$

Referencias

[1] Edward M. Reingold. Basic techniques for design and analysis of algorithms. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*, chapter 4. CRC Press - ACM, 1997.

[2] Alberto Restrepo. Teoría de la complejidad de los algoritmos. Notas de clase.