

Eficiencia algorítmica

Roberto Hernando Velasco
<http://www.rhernando.net>

7 de marzo de 2004

Nota

Estos apuntes los he escrito para estudiar la asignatura *Programación II de Ingeniería Técnica de Informática de Sistemas*, de la UNED.

En ningún momento pretende ser un texto original, es por esto que todo el mérito de lo aquí escrito se debe exclusivamente a los autores que cito en la bibliografía. Sí son de mi responsabilidad todos los errores que se encuentren en estos apuntes.

1. Introducción

El tiempo de ejecución de un algoritmo depende de tres factores:

1. El tamaño de los datos de entrada.
2. El contenido de los datos de entrada.
3. El código generado por el compilador y el computador concretos utilizados. Este último factor no se suele tener en cuenta.

Además, el tiempo de resolución no es fijo para un tamaño de entrada dado. Se pueden definir tres tiempos:

1. Un tiempo *mínimo*, para el conjunto de entrada más “favorable”.
2. Un tiempo *promedio*.
3. Un tiempo *máximo*, para el peor caso posible.

Se suele analizar la eficiencia del algoritmo en el *caso peor*.

2. Medidas asintóticas

2.1. Notación “O grande”

La notación “O grande” se utiliza para manejar la complejidad de un algoritmo, es decir, la cota superior del tiempo de ejecución.

Definición 2.1. Sea $f : \mathbb{N} \rightarrow \mathbb{R}^+ \cup \{0\}$. El conjunto de las *funciones del orden de $f(n)$* se define como

$$O(f(n)) = \{g : \mathbb{N} \rightarrow \mathbb{R}^+ \cup \{0\} : \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N} \mid \forall n \geq n_0, g(n) \leq c \cdot f(n)\}$$

Se dice que g es del orden de f cuando $g(n) \in O(f(n))$.

Definición 2.2. Se dice que el conjunto $O(f(n))$ define un *orden de complejidad*.

Como representante del orden $O(f(n))$ se escogerá la función más sencilla posible.

Proposición 2.3 (Propiedades de $O(f(n))$).

1. $P(n, m) = a_m n^m + \dots + a_1 n + a_0 \in O(n^m)$.
2. Si $f(n) \in O(g(n))$ y $g(n) \in O(h(n))$, entonces $f(n) \in O(h(n))$.
3. Se verifica

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \Rightarrow O(f(n)) \subset O(g(n)).$$

4. **Jerarquía de órdenes de complejidad:**

$$O(1) \subset O(\log n) \subset O(n) \subset O(n \cdot \log n) \subset O(n^2) \subset \dots \subset O(n^a) \subset O(2^n) \subset O(n!).$$

2.2. Notación “omega”

La notación “omega” se utiliza para manejar la cota inferior del tiempo de ejecución.

Definición 2.4. Sea $f : \mathbb{N} \rightarrow \mathbb{R}^+ \cup \{0\}$. El conjunto *omega* de $f(n)$ se define como

$$\Omega(f(n)) = \{g : \mathbb{N} \rightarrow \mathbb{R}^+ \cup \{0\} : \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N} \mid \forall n \geq n_0, g(n) \geq c \cdot f(n)\}$$

2.3. Notación “theta”

La notación “theta” se utiliza para indicar el orden exacto de complejidad (en el caso de que exista).

Definición 2.5. Sea $f : \mathbb{N} \rightarrow \mathbb{R}^+ \cup \{0\}$. El conjunto del *orden exacto* de $f(n)$ se define como

$$\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$$

3. Órdenes de complejidad

Definición 3.1. Los algoritmos cuyas tasas de crecimiento están acotados superiormente por n^a –para cualquier a – se llaman de *complejidad polinomial*, y los problemas que resuelven se denominan *problemas tratables*.

Los problemas de complejidad exponencial o mayor se denominan *problemas intratables*.

4. Cálculo de la eficiencia

4.1. Reglas prácticas

1. A las *instrucciones de asignación, de entrada/salida, o expresiones aritméticas simples* se les asigna una complejidad $\Theta(1)$.
2. Para calcular el coste de una *composición secuencial de instrucciones* se aplica la regla de la suma: $\Theta(f_1(n)) + \Theta(f_2(n)) = \Theta(\max(f_1(n), f_2(n)))$.
3. El coste de una *instrucción condicional*

Si B entonces S_1 sino S_2 finSi

no está en un orden exacto en caso general, por lo que hay que calcular cotas.

Si S_1 está en $O(f_1(n))$, S_2 está en $O(f_2(n))$ y la evaluación de la condición B está en $O(f_B(n))$, el coste de la instrucción está en $O(\max(f_B(n), f_1(n), f_2(n)))$.

4. El coste de una *instrucción iterativa*

Mientras B hacer S finMientras

depende del coste de evaluar B y de ejecutar S , que está en $O(f_{B,S}(n))$. Si las iteraciones se ejecutan un número de veces que está en $O(f_{\text{iter}}(n))$, se tiene que el coste de la instrucción iterativa está en $O(f_{B,S}(n) \cdot f_{\text{iter}}(n))$.

5. Resolución de recurrencias

Cuando se analiza la eficiencia de un programa recursivo es frecuente la aparición de funciones de coste también recursivas, es decir, expresiones del tipo $T(n) = E(n)$, donde en la expresión de E puede aparecer la propia función T . Estas expresiones se llaman *ecuaciones recurrentes*, o *recurrencias*.

5.1. Reducción del problema mediante sustracción

Este tipo de recurrencias aparece al calcular el coste de programas recursivos en los que el tamaño del problema decrece en una cantidad constante de una activación recursiva a las siguientes.

El aspecto general de la ecuación recurrente es el siguiente:

$$T(n) = \begin{cases} cn^k & \text{si } 0 \leq n < b \\ aT(n-b) + cn^k & \text{si } n \geq b \end{cases}$$

donde $a, c \in \mathbb{R}^+$, $k \in \mathbb{R}^+ \cup \{0\}$ y $n, b \in \mathbb{N}$.

Se tiene que

$$T(n) = \begin{cases} \Theta(n^k) & \text{si } a < 1 \\ \Theta(n^{k+1}) & \text{si } a = 1 \\ \Theta(a^{n \text{ div } b}) & \text{si } a > 1 \end{cases}$$

5.2. Reducción del problema mediante división

Este tipo de recurrencias es típico de los programas recursivos que responden al esquema “*divide y vencerás*”. En este esquema, una llamada al programa con un problema de tamaño n genera a llamadas recursivas con subproblemas de tamaño n/b .

El aspecto general de la ecuación recurrente es el siguiente:

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

donde $a, c \in \mathbb{R}^+$, $k \in \mathbb{R}^+ \cup \{0\}$ y $n, b \in \mathbb{N}$ y $b > 1$.

Se tiene que

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

Referencias

- [1] RICARDO PEÑA MARÍ. Diseño de Programas (Formalismo y Abstracción), 2ª edición. *Prentice Hall*, 1998
- [2] DIEGO R. LLANOS FERRARIS Seminarios Asignatura Programación II. *UNED – Centro Asociado de Palencia*, 2003