Análisis de Algoritmos

Amalia Duch

Barcelona, marzo de 2007

Índice

1. Costes en tiempo y en espacio

La característica básica que debe tener un algoritmo es que sea **correcto**, es decir, que produzca el resultado deseado en tiempo finito. Adicionalmente puede interesarnos que sea claro, que esté bien estructurado, que sea fácil de usar, que sea fácil de implementar y que sea **eficiente**.

Entendemos por **eficiencia** de un algoritmo la cantidad de recursos de cómputo que requiere; es decir, cuál es su *tiempo de ejecución* y qué *cantidad de memoria* utiliza.

A la *cantidad de tiempo* que requiere la ejecución de un cierto algoritmo se le suele llamar **coste en tiempo** mientras que a la *cantidad de memoria* que requiere se le suele llamar **coste en espacio**.

Es evidente que conviene buscar algoritmos correctos que mantengan tan bajo como sea posible el consumo de recursos que hacen del sistema, es decir, que sean lo más eficientes posible. Cabe hacer notar que el **concepto de eficiencia** de un algoritmo es un **concepto relativo**, esto quiere decir que ante dos algoritmos correctos que resuelven el mismo problema, uno es **más eficiente** que otro si consume **menos recursos**. Por tanto, podemos observar que el concepto de eficiencia y en consecuencia el **concepto de coste** nos permitirá comparar distintos algoritmos entre ellos.

Si consideramos los algoritmos elementales de ordenación por selección y por inserción ¿cómo podríamos elegir cuál de ellos utilizar en una aplicación dada?

Veremos más adelante que para efectos prácticos ambos algoritmos son similares y que son eficientes sólo para ordenar secuencias con un número *pequeño* de elementos. Sin embargo hay varias comparaciones que se pueden hacer entre ellos.

	Tamaño del vector	Num. Comparaciones	Num. Intercambios
Selección	n	$n(n-1)/2$	0 (min.) $n-1$ (max.)
Inserción	n	$n-1$ (min.) prop. $n^2/4$ (prom.) prop. $n^2/2$ (max.)	0 (min.) prop. $n^2/4$ (prom.) prop. $n^2/2$

De la tabla anterior podemos inferir que para ordenar secuencias cuyos elementos sean difíciles de comparar (las comparaciones son *caras*) es más conveniente utilizar el método de ordenación por inserción, y que este método es más eficiente mientras más *ordenada* esté la secuencia de entrada. Si por el contrario queremos ordenar secuencias de elementos que son fáciles de comparar (o que tienen claves asociadas fáciles de comparar) y en cambio los intercambios son *caros* sería más conveniente utilizar el algoritmo de ordenación por selección.

En general, ¿cómo podemos elegir entre un algoritmo y otro si ambos resuelven el mismo problema?

Una posibilidad es hacerlo mediante **pruebas empíricas**. En este caso habría que traducir ambos algoritmos a un lenguaje de programación, realizar medidas del tiempo de ejecución de los programas cuando se aplican a un conjunto de muestras de entrada y a partir de estas medidas intentar inferir el rendimiento de los programas. Pero este método tiene varios **inconvenientes** ya que los resultados dependen:

- del subconjunto de pruebas escogidas
- del ordenador en el que se realicen las pruebas
- del lenguaje de programación utilizado o
- del compilador, entre otros factores.

Además, no siempre es fácil implementar un algoritmo (algunos pueden ser muy largos o complicados), por tanto, en muchas ocasiones puede ser muy útil poder **predecir** cómo va a comportarse un algoritmo sin tenerlo que implementar. Para ello es conveniente **analizar un algoritmo matemáticamente**.

Ejemplo. Análisis de la función `pos_min`, que encuentra la posición del elemento mínimo en un vector de enteros.

El **coste en tiempo** de un algoritmo depende del tipo de operaciones que realiza y del coste especiífico de estas operaciones. Este coste suele calcularse

únicamente en **función del tamaño de las entradas** para que sea independiente del tipo de máquina, lenguaje de programación, compilador, etc. en que se implementa el algoritmo.

Definición 1.1 *Dado un algoritmo A cuyo conjunto de entradas es $\mathcal{A}$, su* **eficiencia** *o* **coste** *(en tiempo, en espacio, en número de operaciones de entrada/salida, etc.) es una función T tal que:*

$$\begin{aligned} T &: \mathcal{A} \to \mathbb{R}^+ \\ &\quad \alpha \mapsto T(\alpha) \end{aligned}$$

2. Coste en los casos mejor, promedio y peor

Utilizando la definición anterior caracterizar a la función T puede ser complicado. Por ello se definen tres funciones que dependen exclusivamente del tamaño de las entradas y describen resumidamente las características de la función T.

Definición 2.1 *Sea $\mathcal{A}_n$ el conjunto de las entradas de tamaño n y $T_n : \mathcal{A}_n \to \mathbb{R}$ la función T restringida a $\mathcal{A}_n$. Los costes en caso mejor, promedio y peor se definen como sigue:*

Coste en caso mejor *: $T_{mejor}(n) = \min\{T_n(\alpha) | \alpha \in \mathcal{A}_n\}$*

Coste en caso promedio *: $T_{prom}(n) = \sum_{\alpha \in \mathcal{A}_n} Pr(\alpha).T_n(\alpha)$, donde $Pr(\alpha)$ es la probabilidad de ocurrencia de la entrada α.*

Coste en caso peor *: $T_{peor}(n) = \max\{T_n(\alpha) | \alpha \in \mathcal{A}_n\}$*

Por lo general en este curso estudiaremos el coste en caso peor de los algoritmos por dos razones:

1. Proporciona garantías sobre el coste del algoritmo ya que en ningún caso excederá el coste en caso peor.
2. Es más fácil de calcular que el coste en el caso promedio.

Una característica esencial del coste de un algoritmo en cualquiera de los casos descritos anteriormente es su **tasa de crecimiento** o dicho de otra manera su **orden de magnitud**. La tasa de crecimiento de una función marca una diferencia importante con las funciones que tengan un tasa de crecimiento distinta.

Ejemplos. Tablas y curvas de diversas funciones que aparecen frecuentemente en el análisis de algoritmos.

A partir de los ejemplos anteriores puede observarse la conveniencia de omitir los factores constantes, ya que son poco relevantes para comparar tasas de crecimiento (una función cuadrática superará a una lineal más tarde o más temprano). Por este motivo se introduce la notación asintótica.

3. Notación asintótica

En este curso veremos tres clases diferentes de funciones que son: $\mathcal{O}$, Ω y Θ.

Definición 3.1 *Dada una función $f : \mathbb{N} \to \mathbb{R}^+$ la clase $\mathcal{O}(f)$ (que se lee o–grande de f) se define por:*

$$\mathcal{O}(f) = \{g : \mathbb{N} \to \mathbb{R}^+ | \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N} : \forall n \geq n_0, g(n) \leq cf(n)\}$$

Intuitivamente, la definición anterior refleja el hecho de que el crecimiento asintótico de las funciones g es como mucho proporcional al de la función f. Informalmente puede decirse que la tasa de crecimiento de la función f es una cota superior para las tasas de crecimiento de las funciones g.

Ejemplos y ejercicios.

Abuso de notación. Aunque $\mathcal{O}(f)$ es una clase de funciones por tradición se escribe $g = \mathcal{O}(f)$ en vez de $g \in \mathcal{O}(f)$. Cabe observar que escribir $\mathcal{O}(f) = g$ no tiene sentido.

Propiedades básicas de la notación $\mathcal{O}$

1. Si $\lim_{n\to\infty} \frac{g(n)}{f(n)} < \infty$ entonces $g = \mathcal{O}(f)$.
2. $\forall f : \mathbb{N} \to \mathbb{R}^+, f = \mathcal{O}(f)$ (reflexividad).
3. Si $f = \mathcal{O}(g)$ y $g = \mathcal{O}(h)$ entonces $f = \mathcal{O}(h)$ (transitividad).
4. $\forall c > 0, \mathcal{O}(f) = \mathcal{O}(cf)$.

La úlltima propiedad justifica la preferencia por omitir factores constantes. En el caso de la función log se omite la base porque $\log_c(x) = \frac{\log_b(x)}{\log_b(c)}$.

Definición 3.2 *Dada una función $f : \mathbb{N} \to \mathbb{R}^+$ la clase $\Omega(f)$ (que se lee omega de f) se define por:*

$$\Omega(f) = \{g : \mathbb{N} \to \mathbb{R}^+ | \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N} : \forall n \geq n_0, g(n) \geq cf(n)\}$$

Intuitivamente, la definición anterior refleja el hecho de que el crecimiento asintótico de las funciones g es más rápido que el de la función f. Informalmente la tasa de crecimiento de la función f puede verse en este caso como una cota inferior para las tasas de crecimiento de las funciones g.

Ejemplos y ejercicios.

Propiedades básicas de la notación Ω

1. Si $\lim_{n\to\infty} \frac{g(n)}{f(n)} > 0$ entonces $g = \Omega(f)$.
2. $\forall f : \mathbb{N} \to \mathbb{R}^+, f = \Omega(f)$ (reflexividad).
3. Si $f = \Omega(g)$ y $g = \Omega(h)$ entonces $f = \Omega(h)$ (transitividad).
4. Si $f = \mathcal{O}(g)$ entonces $g = \Omega(f)$ y viceversa.

Definición 3.3 *Dada una función $f : \mathbb{N} \to \mathbb{R}^+$ la clase $\Theta(f) = \mathcal{O}(f) \cap \Omega(f)$ (esta clase se lee zita de f).*

Intuitivamente, la definición anterior refleja el hecho de que el crecimiento asintótico de las funciones g es similar al de la función f.

Ejemplos y ejercicios.

Propiedades básicas de la notación Θ

1. $\forall f : \mathbb{N} \to \mathbb{R}^+, f = \Theta(f)$ (reflexividad).
2. Si $f = \Theta(g)$ y $g = \Theta(h)$ entonces $f = \Theta(h)$ (transitividad).
3. $f = \Theta(g) \iff g = \Theta(f)$ (simetría).
4. Si $lim_{n\to\infty} \frac{g(n)}{f(n)} = c, 0 < c < \infty$, entonces $g = \Theta(f)$.

Otras propiedades de las notaciones asintóticas

1. Para cualesquiera constantes positivas $\alpha < \beta$, si f es una función creciente entonces $\mathcal{O}(f^\alpha) \subset \mathcal{O}(f^\beta)$.
2. Para cualesquiera constantes a y b mayores que cero, si f es una función creciente entonces $\mathcal{O}(\log f^a) \subset \mathcal{O}(f^b)$.
3. Para cualquier constante $c > 0$, si f es una función creciente, $\mathcal{O}(f) \subset \mathcal{O}(c^f)$.

Reglas útiles

Regla de las sumas: $\Theta(f) + \Theta(g) = \Theta(f + g) = \Theta(\text{máx}\{f, g\})$.

Regla del producto:: $\Theta(f)\Theta(g) = \Theta(fg)$.

4. Coste de los algoritmos iterativos

Las siguientes reglas facilitan el cálculo o el análisis del coste de los algoritmos iterativos en el caso peor.

1. El coste de una operación elemental es $\Theta(1)$.
2. Si el coste de un fragmento F_1 es f_1 y el de un fragmento F_2 es f_2 entonces el coste en caso peor del fragmento:

   $$F_1; F_2$$

   es:

   $$f_1 + f_2$$

3. Si el coste de F_1 es f_1 y el de F_2 es f_2 y el de evaluar B es g entonces el coste en caso peor del fragmento:

   $$\texttt{if(}B\texttt{)}\ F_1\texttt{; else}\ F_2$$

   es:

   $$g + \text{máx}\{f_1, f_2\}$$

4. Si el coste de F durante la i-ésima iteración es f_i y el coste de evaluar B es g_i y el número de iteraciones es h, entonces el coste en le caso peor del fragmento:

   $$\texttt{while (}B\texttt{)}\ F\texttt{;}$$

   es:

   $$T(n) = \sum_{i=0}^{h(n)} f_i(n) + g_i(n)$$

   Si $f = \text{máx}\{f_i + g_i\}$ entonces $T = \mathcal{O}(hf)$.

5. Coste de los algoritmos recursivos, teoremas maestros

El coste $T(n)$ (en caso peor, medio o mejor) de un algoritmo recursivo, satisface una ecuación recurrente. Esto quiere decir que $T(n)$ dependerá del valor de T para valores más pequeños de n.

Con frecuencia la recurrencia adopta una de las formas siguientes:

1. $T(n) = aT(n-c) + g(n)$, con a y c constantes tales que $a > 0$ y $c > 1$.

2. $T(n) = aT(n/b) + g(n)$, con a y b constantes tales que $a > 0$ y $b > 1$.

La primera recurrencia (llamada recurrencia sustractora) corresponde a algoritmos que tienen una parte no recursiva con coste $g(n)$ y hacen a llamadas recursivas con problemas de tamaño $n - c$.

La segunda (llamada recurrencia divisora) corresponde a algoritmos que tienen una parte no recursiva con coste $g(n)$ y hacen a llamadas recursivas con subproblemas de tamaño aproximado n/b.

Los siguientes teoremas (llamados teoremas maestros) proporcionan un mecanismo para calcular el coste de algoritmos recursivos que se expresen mediante recurrencias sustractoras o divisoras respectivamente.

Teorema 5.1 *Sea $T(n)$ el coste (en el caso peor, medio o mejor) de un algoritmo recursivo que satisface la recurrencia:*

$$T(n) = \begin{cases} f(n) & si\ 0 \leq n < n_0 \\ aT(n-c) + g(n) & si\ n \geq n_0 \end{cases}$$

donde n_0 es un número natural, $c \geq 1$ es una constante, $f(n)$ es una función arbitraria y $g(n) = \Theta(n^k)$ para una constante $k \geq 0$, entonces:

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

Teorema 5.2 *Sea $T(n)$ el coste (en el caso peor, medio o mejor) de un algoritmo recursivo que satisface la recurrencia:*

$$T(n) = \begin{cases} f(n) & si\ 0 \leq n < n_0 \\ aT(n/b) + g(n) & si\ n \geq n_0 \end{cases}$$

donde n_0 es un número natural, $b > 1$ es una constante, $f(n)$ es una función arbitraria y $g(n) = \Theta(n^k)$ para una constante $k \geq 0$. Sea $\alpha = \log_b(a)$. Entonces:

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