Apuntes sobre el cálculo de la eficiencia de los algoritmos

José L. Balcázar

Dept. Llenguatges i Sistemes Informàtics

> Las magnitudes se representan algebraicamente por letras, los hombres por hombres de letras, y así sucesivamente.
>
> Lewis Carrol: "The dynamics of a parti-cle"

Asignaturas:

Programación II - Diplomatura en Informática
Estructuras de Datos y Algoritmos - Ingenierias en Informática

Contenido

1 Introducción y motivación

Un objetivo natural en el desarrollo de un programa es mantener tan bajo como sea posible el consumo de los diversos recursos, aprovechándolos de la mejor manera que se encuentre. Se desea un buen uso, eficiente, de los recursos disponibles, sin desperdiciarlos.

Conviene notar que se trata de un concepto relativo, no absoluto: un algoritmo es más eficiente que otro si realiza las mismas tareas con menos recursos. Si no se dispone de información adicional, no es posible afirmar que un algoritmo es eficiente o que no; para justificar que un algoritmo es ineficiente debemos proporcionar otro más eficiente que él, y para lo contrario es preciso poder argumentar de modo convincente que no existe una manera más eficiente de desarrollar la misma tarea. Suele ser muy difícil encontrar argumentos de este último tipo, y frecuentemente es preciso conformarse con argumentar que un algoritmo es eficiente porque no se nos ocurre otro más eficiente que él (lo cual reconozcamos que no resulta muy convincente...)

Medir la **eficiencia** de un algoritmo es medir la cantidad de recursos necesarios para su ejecución, a efectos de cotejarla con la de otros algoritmos ya inventados o por inventar. Esta cantidad se puede conocer para ciertos

datos simplemente ejecutando el programa, pero la información que se obtiene es muy escasa: no sabemos qué puede pasar en otra máquina, con otro compilador, y sobre todo con otros datos.

Para comparar los varios enfoques que siempre suelen existir para atacar un determinado problema informático, es preciso obtener información *a priori* sobre su futuro comportamiento una vez programados. Teniendo una predicción suficientemente informativa del consumo de recursos inherente al método empleado, se dispone de criterios para elegir el más apropiado como base del programa a desarrollar. De este modo se puede seleccionar la mejor alternativa sin necesidad de completar el desarrollo de un programa para cada una de ellas.

Para poder obtener predicciones de la eficiencia del algoritmo sobre datos futuros, nos basaremos en el **análisis** del algoritmo, es decir, en la consideración, una por una, de las instrucciones que lo componen.

Suelen elegirse como medida factores que se puedan definir formalmente en términos de programación y que tienen la máxima influencia en el coste real de la ejecución: fundamentalmente el tiempo de cálculo y, en algo menor medida, la cantidad de memoria interna. La cantidad de memoria secundaria puede ser relevante en ciertas ocasiones. En programación paralela también se considera a veces como recurso a medir el número de procesadores.

Idealmente, parecería útil que el coste de los recursos necesarios para ejecutar un algoritmo sobre determinados datos se exprese en unidades monetarias. Sin embargo, usualmente preferiremos que nuestro análisis mantenga una cierta independencia de los factores económicos externos, ajenos al problema del diseño de algoritmos, y entonces es preciso evitar este tipo de unidades de coste.

El análisis de la eficiencia de los algoritmos presenta una dificultad básica, que consiste en que los recursos consumidos dependen de varios factores, de los que citamos los más importantes: la máquina en que se ejecute, la calidad del código generado por el *software* (compilador, *linker*, *librerías*...), y el tamaño de los datos. Pero existe una importante diferencia entre estas dependencias: al cambiar de máquina o compilador, la velocidad de ejecución y la memoria usada varían a lo más por factores constantes: una máquina puede resultar diez veces más rápida que otra, o un compilador generar código tres veces más lento. En cambio, la dependencia del tamaño de los datos presenta frecuentemente otro comportamiento: para muchos algoritmos ocurre que al tratar datos que son unas pocas unidades más grandes el uso de recursos se incrementa en mucho más, o se duplica, o resulta elevado al cuadrado.

Por ello, al medir la eficiencia de un algoritmo, expresaremos de manera explícita la dependencia del tamaño de los datos, y lograremos que el análisis sea independiente de los demás factores haciéndolo independiente de

factores multiplicativos, es decir, del producto por constantes. Este grado de abstracción se conseguirá clasificando las expresiones que denotan el uso de recursos en función de su velocidad de crecimiento.

Existe otra manera de obtener estos resultados, que consiste en medir el número de veces que se ejecutan determinadas instrucciones *patrón* que se consideran particularmente significativas (operaciones aritméticas, comparaciones...). No seguiremos este método, que suele dar resultados equivalentes y cuya descripción puede encontrarse por ejemplo en Wirth (1986) y en Brassard y Bratley (1987).

Al expresar la eficiencia de un algoritmo en función del tamaño de los datos, aún hay una decisión a tomar. En efecto, para datos distintos pero del mismo tamaño el algoritmo analizado puede comportarse de diversas maneras, y es preciso indicar si la expresión que indica su eficiencia corresponde al caso peor, al caso mejor o, en algún sentido, a un caso medio. El análisis del caso peor, el más comunmente usado, presenta como desventaja su excesivo pesimismo, dado que posiblemente el comportamiento real del programa sea algo mejor que el descrito por el análisis. El caso mejor suele ser irreal (por demasiado optimista) para ser de utilidad práctica. El caso medio corresponde al cálculo de una media del uso de recursos que, como es lógico, debe ser ponderada por la frecuencia de aparición de cada caso; es decir, requiere el cálculo de una esperanza matemática a partir de una distribución de probabilidad. Este análisis proporciona datos mucho más interesantes sobre el algoritmo, pero requiere a su vez más datos (la distribución de probabilidad) sobre el entorno en que éste se ejecuta, de los que es común carecer. Incluso se ha demostrado que una decisión salomónica como la de dotar a las entradas de una distribución de probabilidad uniforme puede llevar a resultados engañosos. Debido a esta dificultad, así como a la necesidad de recurrir a herramientas matemáticas más sofisticadas, raramente consideraremos este enfoque aquí.

2 El tamaño de los datos

Ante un algoritmo cuyo consumo de recursos deseamos predecir, en función del tamaño de los datos, es preciso, primero, determinar qué entenderemos por *tamaño de los datos*; y segundo, descomponer el algoritmo en sus unidades fundamentales y deducir de cada una de ellas una predicción del uso de recursos.

Al respecto del primer punto, algunas sugerencias que pueden resultar útiles son las siguientes. Sobre datos naturales o enteros, puede ser bien su propio valor, bien el tamaño de su descripción binaria, que resulta logarítmico

en el valor. Es preciso especificar claramente cuál de estas posibilidades se sigue, dado que se diferencian nada menos que en una exponencial. Sobre pares o ternas de enteros, conviene intentar expresar el tamaño en función de todos ellos; sin embargo, a veces un análisis más sencillo y suficientemente informativo puede hacer depender el consumo de recursos sólo de uno de ellos, suponiendo los demás constantes. Esta decisión es claramente irreal si ambos van a variar, por lo que su utilidad puede ser discutible; sin embargo, puede resultar suficiente en ciertos casos a efectos de comparar distintos algoritmos para resolver el mismo problema.

Cuando los datos son estructuras como vectores o ficheros, si las componentes de éstos no van a alcanzar tamaños considerables entonces se puede considerar como tamaño de los datos el número de éstas. Esta decisión puede ser inadecuada ocasionalmente, como por ejemplo en caso de estructuras cuyos elementos son a su vez otras estructuras, o por ejemplo enteros de magnitudes muy elevadas para los que no basta una palabra de máquina; pero con frecuencia es apropiada. El concepto de tamaño para otros tipos de datos que sea necesario tratar ha de ser definido de alguna manera *ad hoc*, la que más apropiada parezca.

En cuanto al análisis propiamente dicho de la cantidad de recursos requerida por el algoritmo, éste se realiza naturalmente en base a las instrucciones que lo componen. Más adelante lo discutimos en mayor detalle, pero para facilitar la comprensión mencionemos ahora algunas ideas sobre cómo hacerlo. La evaluación de una expresión requiere como tiempo la suma de los tiempos de ejecución de las operaciones (o llamadas a funciones) que aparezcan en ellas. Una asignación a una variable no estructurada, o una operación de escritura de una expresión, suelen requerir el tiempo de evaluar la expresión, más unas operaciones adicionales de gestión interna de coste constante. Una lectura de una variable no estructurada suele requerir asimismo tiempo constante. El coste de una secuencia de instrucciones es naturalmente la suma de los costes de la secuencia, que como veremos suele coincidir con el máximo de los costes. En una alternativa, generalmente consideraremos el coste de evaluar la expresión booleana, sumado al máximo de los costes de las ramas (en el análisis del caso peor). Para un bucle es preciso sumar los costes de cada vuelta, sin olvidar que cada una requiere una evaluación de la expresión indicada en la condición del bucle. Finalmente, una llamada a procedimiento o función requiere el análisis previo del mismo; en caso de que éste sea recursivo aparecerá una ecuación recurrente, por lo cual dedicamos también una sección a presentar una breve introducción a la resolución de este tipo de ecuaciones.

3 Las notaciones asintóticas y sus propiedades

Presentamos a continuación algunas de las notaciones más corrientes para clasificar funciones en base a su velocidad de crecimiento, su **orden de magnitud**. Se basan en su comportamiento en casos límite, por lo que definen lo que denominaremos **coste asintótico** de los algoritmos.

Sólo algunas de ellas son utilizadas con frecuencia; las demás se incluyen para facilitar las definiciones de otras notaciones, o bien para ofrecer un panorama más completo de un tema por lo general insuficientemente tratado en la bibliografía. Seguimos esencialmente Knuth (1973), Aho, Hopcroft y Ullman (1983), y Vitányi y Meertens (1983).

A lo largo de la presente sección, todas las funciones que aparezcan, salvo indicación en contrario, son funciones de $\mathbb{N}^+$, los naturales positivos, en $\mathbb{N}^+$. Todas las definiciones y resultados son válidos también para funciones de $\mathbb{N}^+$ en $\mathbb{R}$ que tomen valores siempre superiores a la unidad.

3.1 La O mayúscula

Las primeras notaciones corresponden a cotas superiores. La notación O mayúscula, $O(f)$, denota el conjunto de las funciones g que *crecen a lo más tan rápido como* f, es decir, las funciones g tales que, módulo constantes multiplicativas, f llega a ser en algún momento una cota superior para g. Su definición formal es:

$$O(f) = \{g \mid \exists c_0 \in \mathbb{R}^+ \; \exists n_0 \in \mathbb{N} \; \forall n \geq n_0 \; g(n) \leq c_0 f(n)\}$$

La correspondencia con la definición intuitiva es: el cuantificador existencial sobre c_0 formaliza la expresión *módulo constantes multiplicativas*, y n_0 indica a partir de qué punto f es realmente una cota superior para g.

A modo de ejemplo, aplicando la definición puede comprobarse que para $f(n) = n^2$ y $g(n) = n$ se tiene que $g \in O(f)$ pero sin embargo $f \notin O(g)$. Para $f(n) = 3n^2$ y $g(n) = 100n^2$ se tiene que $g \in O(f)$ y que $f \in O(g)$.

La notación $O(f)$ tiene las siguientes propiedades, cualesquiera que sean las funciones f, g y h.

1. *Invariancia multiplicativa.* Para toda constante $c \in \mathbb{R}^+$,

   $$g \in O(f) \iff c \cdot g \in O(f)$$

   Se demuestra por simple aplicación de la definición.

2. *Reflexividad.* $f \in O(f)$. Basta comprobar la definición.

3. *Transitividad.* Si $h \in O(g)$ y $g \in O(f)$ entonces $h \in O(f)$. Se demuestra mediante sencillas manipulaciones aritméticas.

4. *Criterio de caracterización.* $g \in O(f) \iff O(g) \subseteq O(f)$. Se deduce fácilmente de las dos anteriores.

5. *Regla de la suma para O.* $O(f+g) = O(\max(f,g))$, donde denotamos $f+g$ la función que sobre el argumento n vale $f(n)+g(n)$, y $\max(f,g)$ la función que sobre el argumento n vale el máximo de $\{f(n), g(n)\}$. Se puede demostrar a partir de las siguientes desigualdades:

   $$\max(f,g)(n) \leq f(n)+g(n) \leq \max(f,g)(n)+\max(f,g)(n) \leq 2\max(f,g)(n)$$

   Frecuentemente esta regla se aplica así: si $g_1 \in O(f_1)$ y $g_2 \in O(f_2)$, entonces $g_1 + g_2 \in O(\max(f_1, f_2))$.

6. *Regla del producto para O.* Si $g_1 \in O(f_1)$ y $g_2 \in O(f_2)$, entonces $g_1 \cdot g_2 \in O(f_1 \cdot f_2)$, donde denotamos $f \cdot g$ la función que sobre el argumento n vale $f(n) \cdot g(n)$. La demostración es inmediata.

7. *Invariancia aditiva.* Para toda constante $c \in \mathbb{R}^+$,

   $$g \in O(f) \iff c + g \in O(f)$$

   Es consecuencia inmediata de la regla de la suma.

3.2 La *o* minúscula

La siguiente notación es también una cota superior, pero con un significado diferente. Mientras que $g \in O(f)$ indica que, módulo constantes multiplicativas, f llega a ser en algún momento una cota superior para g, y por tanto que el crecimiento de g no supera el de f, en cambio $g \in o(f)$ indica que el crecimiento de g es estrictamente más lento que el de f. Su definición es:

$$o(f) = \{g \mid \forall c_0 \in \mathbb{R}^+ \; \exists n_0 \in \mathbb{N} \; \forall n \geq n_0 \; g(n) \leq c_0 f(n)\}$$

Comparando con la definición de $O(f)$ vemos que la diferencia formal radica en la cuantificación de la constante real positiva. Si un cuantificador existencial significaba en $O(f)$ que *teníamos permiso* para multiplicar f por cualquier constante con tal de alcanzar a g, ahora un cuantificador universal indica que, cualquiera que sea el factor constante de *aplastamiento* de f, ésta aún es capaz a la larga de dominar el crecimiento de g. Por ello, el rango interesante del $\forall c_0$ en la definición de $o(f)$ está formado por constantes inferiores a la unidad.

Como ejemplo, tomemos de nuevo $f(n) = n^2$ y $g(n) = n$. Aplicando la definición puede comprobarse inmediatamente que $g \in o(f)$ y que $f \notin o(g)$. Para el otro ejemplo anteriormente considerado, $f(n) = 3n^2$ y $g(n) = 100n^2$, se tiene que $g \notin o(f)$ y $f \notin o(g)$.

La notación $o(f)$ tiene las siguientes propiedades, cualesquiera que sean las funciones f, g y h. Denotamos por $\subset$ la inclusión propia, es decir inclusión sin igualdad.

1. *Invariancia multiplicativa.* Para toda constante $c \in \mathbb{R}^+$,

$$g \in o(f) \iff c \cdot g \in o(f)$$

   Se demuestra aplicando la definición.

2. *Invariancia aditiva.* Para toda constante $c \in \mathbb{R}^+$,

$$g \in o(f) \iff c + g \in o(f)$$

3. *Antirreflexividad.* $f \notin o(f)$. Basta comprobar la definición.

4. *Caracterización en términos de O.* $g \in o(f)$ si y sólo si $g \in O(f)$ pero $f \notin O(g)$; en particular, $o(f) \subset O(f)$, y la antirreflexividad demuestra que la inclusión es propia. La demostración es sencilla.

5. *Otra relación con O.* $g \in o(f) \iff O(g) \subseteq o(f)$. Se demuestra por simple manipulación aritmética, y usando las propiedades anteriores.

6. *Transitividad.* Si $h \in o(g)$ y $g \in o(f)$ entonces $h \in o(f)$. Se deduce de las propiedades anteriores.

7. *Caracterización por límites.* Se cumple que:

$$g \in o(f) \iff \lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$$

   Demostración: escribiendo la expresión $g(n) \leq c_0 f(n)$ como $\frac{g(n)}{f(n)} \leq c_0$ queda precisamente la definición de que el límite indicado exista y sea cero.

3.3 Las notaciones Ω

Para denotar cotas inferiores de manera análoga a las cotas superiores, disponemos de las notaciones Ω. En la bibliografía existen dos definiciones distintas, que rara vez aparecen juntas en el mismo texto ya que en general los autores siempre se decantan hacia una u otra. La primera, que denotaremos $g \in \Omega_K(f)$, indica que f es una cota inferior a g desde un punto en adelante. El subíndice K es la inicial del autor que la propuso (D. Knuth). Proporciona bastante información si el crecimiento de g es homogéneo, pues indica un mínimo del cual g nunca baja. Sin embargo, si g presenta fuertes oscilaciones (y esto ocurre a veces en las funciones obtenidas por análisis de algoritmos), se puede obtener más información si buscamos cotas inferiores a los máximos (los *picos*) de g. Denotamos $g \in \Omega_\infty(f)$ el hecho de que en una cantidad infinita de ocasiones g crezca lo suficiente como para alcanzar a f. Las definiciones formales son:

$\Omega_K(f) = \{g \mid \exists c_0 \in \mathbb{R}^+ \; \exists n_0 \in \mathbb{N} \; \forall n \geq n_0 \; g(n) \geq c_0 f(n)\}$
$\Omega_\infty(f) = \{g \mid \exists c_0 \in \mathbb{R}^+ \; \forall n_0 \in \mathbb{N} \; \exists n \geq n_0 \; g(n) \geq c_0 f(n)\}$

La disposición de los cuantificadores describe que la primera definición requiere a g crecer por encima de f para siempre de un punto en adelante, mientras que en la segunda sólo se requiere que g supere a f en puntos tan avanzados como se desee. La notación Ω con el significado de $\Omega_\infty(f)$ es muy anterior a Knuth, y ya aparece en los trabajos de G. H. Hardy y J. E. Littlewood en 1914.

Las notaciones $\Omega(f)$ tienen las siguientes propiedades, cualesquiera que sean las funciones f, g y h.

1. *Invariancia multiplicativa.* Para toda constante $c \in \mathbb{R}^+$, $g \in \Omega_K(f) \iff c \cdot g \in \Omega_K(f)$
   $g \in \Omega_\infty(f) \iff c \cdot g \in \Omega_\infty(f)$

2. *Invariancia aditiva.* Para toda constante $c \in \mathbb{R}^+$, $g \in \Omega_K(f) \iff c + g \in \Omega_K(f)$
   $g \in \Omega_\infty(f) \iff c + g \in \Omega_\infty(f)$

3. *Relación entre Ω_K y Ω_∞.* $\Omega_K(f) \subseteq \Omega_\infty(f)$. Se puede demostrar que la inclusión es estricta.

4. *Reflexividad.* $f \in \Omega_K(f)$, y por tanto $f \in \Omega_\infty(f)$.

5. *Transitividad.* Si $h \in \Omega_K(g)$ y $g \in \Omega_K(f)$ entonces $h \in \Omega_K(f)$. Sin embargo, se puede demostrar que $\Omega_\infty(f)$ no es transitiva.

6. *Relación con O.* $g \in \Omega_K(f) \iff f \in O(g)$.

7. *Relación con o.* $g \in \Omega_\infty(f) \iff g \notin o(f)$.

3.4 Las notaciones Θ

Denotan las funciones con la misma tasa de crecimiento que f.

$$\Theta(f) = O(f) \cap \Omega_K(f)$$

Se puede definir otra versión diferente, $\Theta_\infty(f) = O(f) \cap \Omega_\infty(f)$, que no trataremos aquí. La clase $\Theta(f)$ es un subconjunto de $O(f)$, y se denomina a veces el **orden de magnitud** de f.

Algunas propiedades de la notación $\Theta(f)$ son:

1. *Invariancia multiplicativa.* Para toda constante $c \in \mathbb{R}^+$,

$$g \in \Theta(f) \iff c \cdot g \in \Theta(f)$$

2. *Invariancia aditiva.* Para toda constante $c \in \mathbb{R}^+$,

$$g \in \Theta(f) \iff c + g \in \Theta(f)$$

3. *Relación con O.* Se tienen las siguientes equivalencias:

$$g \in \Theta(f) \iff (g \in O(f) \text{ y } f \in O(g)) \iff O(f) = O(g)$$

   Pueden demostrarse fácilmente observando que en previas propiedades se ha indicado que $g \in O(f)$ si y sólo si $O(g) \subseteq O(g)$, y que $g \in \Omega_K(f)$ si y sólo si $f \in O(g)$.

4. *Condición de límite.* Si el límite $\lim_{n\to\infty} g(n)/f(n)$ existe, es finito y no es cero, entonces $g \in \Theta(f)$.

5. *Reflexividad.* $f \in \Theta(f)$.

6. *Simetría.* $g \in \Theta(f) \iff f \in \Theta(g) \iff \Theta(f) = \Theta(g)$.

7. *Transitividad.* Si $h \in \Theta(g)$ y $g \in \Theta(f)$ entonces $h \in \Theta(f)$.

8. *Regla de la suma para Θ.* $\Theta(f + g) = \Theta(\max(f, g))$. Frecuentemente esta regla se aplica así: si $g_1 \in \Theta(f_1)$ y $g_2 \in \Theta(f_2)$ entonces $g_1 + g_2 \in \Theta(\max(f_1, f_2))$. Se deduce de la regla de la suma para O y de las propiedades ya mencionadas.

9. *Regla del producto para Θ.* Si $g_1 \in \Theta(f_1)$ y $g_2 \in \Theta(f_2)$ entonces $g_1 \cdot g_2 \in \Theta(f_1 \cdot f_2)$.

4 Formas de crecimiento frecuentes

La escala con arreglo a la cual clasificaremos el uso de recursos de los algoritmos consiste en una serie de funciones elegidas entre las que es posible definir por operaciones aritméticas, logaritmos y exponenciales. A lo largo de esta sección, y salvo indicación explícita en contrario, la base de todos los logaritmos es 2 (si bien en la mayoría de los casos la independencia de factores constantes da lugar a que la base del logaritmo sea indiferente).

Las más importantes formas de crecimiento asintótico son las siguientes:

- Logarítmico: $\Theta(\log n)$
- Lineal: $\Theta(n)$
- Quasilineal: $\Theta(n \cdot \log n)$
- Cuadrático: $\Theta(n^2)$
- Cúbico: $\Theta(n^3)$
- Polinómico: $\Theta(n^k)$ con k fijo.
- Exponencial: $\Theta(k^n)$ con k fijo.

Otras formas de crecimiento que pueden aparecer con cierta frecuencia incluyen $\Theta(\sqrt{n})$, $\Theta(n!)$ o $\Theta(n^n)$. En ocasiones se entiende por crecimiento quasilineal el que corresponde a $\Theta(n \cdot (\log n)^k)$ con k fijo. A falta de posibilidades de comparación, suele admitirse como algoritmo eficiente el que alcanza un coste quasilineal. Cualquier coste superior a polinómico, e incluso un coste polinómico con un grado elevado, es un coste que se considera intratable; en efecto, incluso sobre casos de tamaño moderado, los algoritmos que exhiben ese coste suelen exigir ya unos recursos prohibitivos.

(Figura: Crecimiento de algunas funciones)

La figura, debida a C. Rosselló, describe gráficamente las relaciones entre los crecimientos de algunas de estas funciones. Demostramos a continuación que las relaciones que esta figura sugiere efectivamente se cumplen (obsérvese que en su mayoría se enuncian para funciones de valor real).

1. Para cualesquiera constantes reales positivas c_1, c_2, $c_1 \cdot \log n \in o(n^{c_2})$.

   Demostración. Evaluamos por la regla de L'Hôpital el siguiente límite:

$$\lim_{n\to\infty} \frac{c_1 \cdot \log n}{n^{c_2}} = \lim_{n\to\infty} \frac{c_1/n}{c_2 \cdot n^{c_2-1}} = \lim_{n\to\infty} \frac{c_1}{c_2 \cdot n^{c_2}} = 0$$

y aplicamos la caracterización por límites de $o(f)$. □

2. Para cualesquiera constantes reales positivas c_1, c_2, si $c_1 < c_2$ entonces $n^{c_1} \in o(n^{c_2})$.

   Demostración. Si $c_1 < c_2$ entonces el cociente $\frac{n^{c_1}}{n^{c_2}}$ tiende a cero. □

3. Dado un polinomio de grado k, $p(n) = a_0 n^k + a_1 n^{k-1} + \cdots + a_{k-1} n + a_k$, con $a_0 \neq 0$, $p(n) \in \Theta(n^k)$.

   Demostración. Evaluamos el límite:

$$\lim_{n\to\infty} \frac{a_0 n^k + a_1 n^{k-1} + \cdots + a_{k-1} n + a_k}{n^k} = a_0 \neq 0$$

   y aplicamos la condición de límite de $\Theta(f)$. □

4. Para cualesquiera constantes reales positivas c_1, c_2, con $c_2 > 1$, $n^{c_1} \in o({c_2}^n)$.

   Demostración. Demostramos que

$$\lim_{n\to\infty} \frac{n^{c_1}}{{c_2}^n} = 0$$

   Tomando logaritmos y antilogaritmos adecuadamente, basta demostrar que

$$\lim_{n\to\infty} (c_2 \cdot n - c_1 \cdot \log n) = \infty$$

   Sea c_3 una constante real positiva cualquiera. Evaluamos primero el siguiente límite de manera análoga al punto 1.

$$\lim_{n\to\infty} \frac{c_1 \cdot \log n + c_3}{n} = 0$$

   Por la definición de límite, este hecho equivale a decir que para cualesquiera constantes reales positivas c_1, c_2, c_3, y de un cierto n_0 en adelante, $\frac{c_1 \cdot \log n + c_3}{n} < c_2$. Operando, obtenemos que para cualesquiera constantes reales positivas c_1, c_2, c_3, y de un cierto n_0 en adelante, $c_3 < c_2 \cdot n - c_1 \cdot \log n$, lo cual equivale a decir que el límite

$$\lim_{n\to\infty} c_2 \cdot n - c_1 \cdot \log n = \infty$$

   como deseábamos demostrar. □

5. Si $1 < c_1 < c_2$, entonces ${c_1}^n \in o({c_2}^n)$.

   Demostración. Evaluamos

   $$\lim_{n\to\infty} \frac{{c_1}^n}{{c_2}^n} = \lim_{n\to\infty} \left(\frac{c_1}{c_2}\right)^n = 0$$

   ya que $\frac{c_1}{c_2} < 1$. □

6. $c^n \in o(n!)$.

   Demostración. Consideremos la sucesión cuyo término general es $\frac{c^n}{n!}$. Para $n > c$, el valor de $\frac{c^n}{n!}$ puede acotarse de la siguiente forma: un primer factor c/n; el producto de factores $c/(n-1)\cdot c/(n-2)\cdots c/(c+1)$, que es menor que 1 por serlo todos los factores; y el producto de sus c últimos términos $c/c\cdots c/2 \cdot c/1$, que es menor que c^c. Por tanto tenemos

   $$0 \leq \frac{c^n}{n!} \leq \frac{c}{n}c^c = \frac{c^{c+1}}{n}$$

   y por tanto $\lim_{n\to\infty} \frac{c^n}{n!} = 0$. □

7. $n! \in o(n^n)$.

   Demostración. Por un razonamiento análogo al anterior, podemos acotar el cociente $\frac{n!}{n^n}$ por $\frac{1}{n}$. El argumento es el mismo. □

5 Algunas precauciones a tomar

La independencia de factores constantes propia de las notaciones de orden de magnitud ofrece la indudable ventaja de proporcionar datos sobre el algoritmo independientemente de las circunstancias de su ejecución, pero no está exenta de inconvenientes. En efecto, en la interpretación de medidas así expresadas se ha de ser cauteloso.

Supongamos que comparamos dos algoritmos, uno de ellos de bajo coste asintótico, pero con un elevado factor constante, *escondido* en la notación de orden de magnitud, y otro que asintóticamente es peor pero cuya constante escondida es muy inferior. Es posible que el mayor orden de crecimiento del segundo, comparado con la alta constante escondida del primero, sólo se manifieste más ineficiente a partir de un tamaño de datos exageradamente grande, haciendo inútil el algoritmo teóricamente mejor. Por ejemplo, existe un algoritmo de multiplicación de enteros más eficiente en orden de magnitud que el algoritmo de multiplicación clásico, pero debido a su elevada constante escondida resulta más lento que éste para multiplicar enteros de menos de

500 bits; escasean por tanto las oportunidades de aplicación práctica del algoritmo más eficiente.

Sin embargo, estos casos son más la excepción que la regla, siendo usual que, para una misma instalación informática, las constantes escondidas correspondientes a dos algoritmos distintos que resuelven el mismo problema no sean muy diferentes; en tales casos, los algoritmos con mejores crecimientos asintóticos son ya suficientemente más eficientes para datos de tamaños prácticos como para hacer aconsejable su uso.

No ha de olvidarse que, junto al principio informático que sugiere que *es mejor no hacer a mano lo que se puede hacer eficientemente por ordenador*, existe también el inverso, *es mejor no hacer por ordenador lo que se puede hacer eficientemente a mano*; es decir, que si merece la pena programar la tarea a realizar es porque el tamaño de los datos hace desaconsejable realizar la tarea manualmente, y por tanto se puede esperar que este tamaño también sea suficientemente grande para hacer notar el crecimiento asintótico por encima de las constantes escondidas en el consumo de recursos.

Un comentario adicional de carácter práctico que debemos recordar es que, para tomar una decisión sobre el uso de uno u otro algoritmo en una aplicación determinada, el tiempo de desarrollo del programa debe ser también tenido en cuenta, y que puede ser inadecuado dedicar una cantidad excesiva de tiempo de programación a desarrollar un eficiente pero complejo programa que sólo vaya a ser usado después unas pocas veces. El ahorro conseguido por la eficiencia del algoritmo debe compensar la sobrecarga de trabajo que pueda suponer su programación.

6 Cálculo del coste

Para evaluar en orden de magnitud el tiempo de ejecución de un programa suele bastar una descripción de éste en relativamente alto nivel: basta con conocer los rasgos principales del algoritmo, y no suelen afectar (como en cambio sí lo hace a la corrección) los pequeños pero importantísimos detalles de programación. Por ejemplo, en el diseño de un programa recursivo, tan pronto como se completa el análisis de casos y está claro el número de llamadas recursivas, el tamaño de los parámetros de éstas y el coste de las operaciones adicionales a realizar, ya es posible en muchos casos completar el análisis.

Como ya hemos dicho antes, para predecir el coste de un algoritmo es preciso combinar apropiadamente el de todas sus instrucciones. Lo natural es sumarlos, pero obsérvese que por la regla de la suma el coste de una suma de una cantidad fija de funciones está en el mismo orden de magnitud que

la mayor de ellas. Así, por ejemplo, la evaluación de una expresión requiere, como ya hemos dicho, sumar los tiempos de ejecución de las operaciones que aparezcan en ella, y ello es equivalente a considerar sólo la más *cara*. Hay que prestar atención a que el tamaño de los argumentos en llamadas a funciones puede no ser siempre el mismo. Por lo general se pueden considerar constantes las evaluaciones de operaciones booleanas. También las aritméticas si se puede argumentar que las podrá efectuar el *hardware*, y que por tanto los operandos son suficientemente pequeños respecto de los datos del programa como para considerarlos de tamaño constante (representables en pocas palabras de memoria). En caso contrario, la aritmética se está efectuando también por *software*, y su coste dependerá del algoritmo que se emplee para su cálculo. Otras funciones que se hayan programado requerirán también el correspondiente análisis de coste. Todos estos criterios pueden aplicarse tanto a las asignaciones simples como a las múltiples.

En general este coste de evaluar expresiones aparece en todas las instrucciones, sean de asignación, alternativa o bucle, o incluso de lectura o escritura. Asimismo, toda instrucción contribuye un coste fijo pero positivo, incluso la instrucción de *no hacer nada*, debido a que el programa en ejecución ha de efectuar actividades fijas concretas inevitables (incremento del contador de programa, *fetch* de la instrucción...). Del mismo modo, cuando se tiene una instrucción compuesta secuencialmente de otras, su coste es el del máximo de éstas más el coste constante adicional debido a estas actividades fijas.

En una alternativa, el cálculo del coste ha de considerar todas las ramas programadas, seleccionando el máximo de todas ellas, ya que hemos convenido en efectuar el análisis del caso peor. De no ser así, es necesaria información sobre la probabilidad de elegir cada una de las ramas. Existe un caso particular de análisis de alternativas, que se da cuando alguna de las ramas contiene llamadas recursivas a la función que se está analizando. Aparece entonces una ecuación de recurrencia, que será preciso resolver: enseguida explicaremos métodos para hacerlo.

Para evaluar el coste de un bucle sumaremos los costes de todas las vueltas, contando en cada una al menos el tiempo constante requerido para evaluar la expresión booleana que aparece como condición del bucle. Si supiéramos que el número de vueltas es fijo y constante, podríamos aplicar de nuevo la regla de la suma; pero en la inmensa mayoría de los casos ésto no es así, sino que el número de vueltas depende del tamaño de los datos, y ya no es posible aplicar la regla de la suma. A veces resulta difícil sumar con exactitud el coste de todas las vueltas, y en este caso puede ser útil conformarse no con una expresión exacta $\Theta(f)$ del tiempo de ejecución sino con sóo una cota superior $O(f)$. Esta se puede obtener, por ejemplo, si conseguimos evaluar el coste de la vuelta más cara y también una fórmula del número de

vueltas, pues entonces basta multiplicarlas. Otra posibilidad es que la suma a evaluar sea parte de una serie infinita cuya suma pueda calcularse mediante las técnicas del Análisis Matemático.

7 Ecuaciones recurrentes

Como ya hemos indicado, el análisis del tiempo de ejecución de un programa recursivo vendrá en función del tiempo requerido por la(s) llamada(s) recursiva(s) que aparezcan en él. De la misma manera que la verificación de programas recursivos requiere razonar por inducción, para tratar apropiadamente estas llamadas, el cálculo de su eficiencia requiere un concepto análogo: el de ecuación de recurrencia. Demostraciones por inducción y ecuaciones de recurrencia son por tanto los conceptos básicos para tratar programas recursivos.

Supongamos que se está analizando el coste de un algoritmo recursivo, intentando calcular una función que indique el uso de recursos, por ejemplo el tiempo, en términos del tamaño de los datos; denominémosla $T(n)$ para datos de tamaño n. Nos encontramos con que este coste se define a su vez en función del coste de las llamadas recursivas, es decir, de $T(m)$ para otros tamaños m (usualmente menores que n): esta manera de expresar el coste T en función de sí misma es lo que denominamos una ecuación recurrente, y su resolución, es decir, la obtención para T de una fórmula cerrada (independiente de T) puede ser dificultosa.

A continuación veremos la solución de una amplia clase de recurrencias típicas que aparecen frecuentemente en el análisis de algoritmos recursivos más o menos sencillos; esta discusión bastará para la mayoría de los análisis que se requieran efectuar en las asignaturas de Algorítmica, salvo quizá las más especializadas.

Teorema. Sean $T_1(n)$ y $T_2(n)$ funciones que cumplen las siguientes ecuaciones recurrentes:

$$T_1(n) = \begin{cases} f(n) & \text{si } 0 \leq n < c \\ a \cdot T_1(n-c) + b \cdot n^k & \text{si } c \leq n \end{cases}$$

$$T_2(n) = \begin{cases} g(n) & \text{si } 0 \leq n < c \\ a \cdot T_2(n/c) + b \cdot n^k & \text{si } c \leq n \end{cases}$$

donde f y g son funciones arbitrarias. Entonces los órdenes de magnitud de

los respectivos crecimientos son:

$$T_1(n) \in \begin{cases} \Theta(n^k) & \text{si } a < 1 \\ \Theta(n^{k+1}) & \text{si } a = 1 \\ \Theta(a^{n/c}) & \text{si } a > 1 \end{cases}$$

$$T_2(n) \in \begin{cases} \Theta(n^k) & \text{si } a < c^k \\ \Theta(n^k \cdot \log n) & \text{si } a = c^k \\ \Theta(n^{\log_c a}) & \text{si } a > c^k \end{cases}$$

Demostración. Expandiendo la recurrencia de $T_1(n)$ tantas veces como sea posible se obtiene

$$T_1(n) = a^m \cdot T_1(n - m \cdot c) + \sum_{i=0}^{m-1} b \cdot a^i \cdot (n - i \cdot c)^k$$

donde m es tal que la recurrencia no se puede aplicar más, es decir, tal que $0 \leq n - m \cdot c < c$, o lo que es equivalente, $m \leq n/c < m + 1$. Como c es constante, tenemos $\Theta(m) = \Theta(n)$. Sacando factores comunes, multiplicando por c^k, y seleccionando la apropiada constante adicional b' (que dependerá del valor máximo de T_1 para argumentos inferiores a c), se puede obtener la siguiente acotación:

$$b \cdot c^k \cdot \sum_{i=0}^{m-1} a^i \cdot (m - i)^k \leq T_1(n) \leq b' \cdot c^k \cdot \sum_{i=0}^{m} a^i \cdot (m + 1 - i)^k$$

Para $a < 1$, conservando sólo el primer sumando de la primera fórmula, sustituyendo $(m+1-i)$ por $m+1$ que es superior y observando que $\sum_{i=0}^{\infty} a^i = (1 - a)^{-1}$, se deduce la siguiente acotación, más simple

$$b \cdot c^k \cdot m^k \leq T_1(n) \leq b' \cdot c^k \cdot (m + 1)^k \frac{1}{(1 - a)}$$

y por tanto $T_1(n) \in \Theta(m^k) = \Theta(n^k)$ ya que a, b, b', c y k son constantes. Para $a = 1$, invirtiendo el orden en que se realiza el primer sumatorio y sustituyendo de nuevo $(m + 1 - i)$ por $m + 1$, se obtiene

$$b \cdot c^k \cdot \sum_{i=1}^{m} i^k \leq T_1(n) \leq b' \cdot c^k \cdot (m + 1)^{k+1}$$

y usando el hecho de que $\sum_{i=1}^{m} i^k \in \Theta(m^{k+1})$, deducimos que $T_1(n) \in \Theta(n^{k+1})$. Finalmente, para $a > 1$, conservando sólo el último sumando de la cota inferior y operando en la superior tenemos:

$$b \cdot c^k \cdot a^{m-1} \leq T_1(n) \leq b' \cdot c^k \cdot a^{m+1} \sum_{i=0}^{m} a^{i-m-1} \cdot (m + 1 - i)^k$$

El sumatorio de la cota superior cumple ahora:

$$\sum_{i=0}^{m} a^{i-m-1} \cdot (m+1-i)^k = \sum_{i=1}^{m+1} a^{-i} \cdot i^k \leq \sum_{i=1}^{\infty} a^{-i} \cdot i^k = k'$$

para alguna constante k', ya que la última serie es convergente. Por tanto,

$$b \cdot c^k \cdot a^{m-1} \leq T_1(n) \leq b' \cdot c^k \cdot a^{m+1} k'$$

de donde se deduce de inmediato que $T_1(n) \in \Theta(a^{n/c})$.

Pasando a estudiar el crecimiento de $T_2(n)$, empezamos nuevamente por expandir la recurrencia tanto como sea posible:

$$T_2(n) = a^m \cdot T_2(\frac{n}{c^m}) + \sum_{i=0}^{m-1} b \cdot a^i \cdot (\frac{n}{c^i})^k$$

donde m es de nuevo el límite de aplicación de la recurrencia: $1 \leq n/c^m < c$, o lo que es equivalente, $c^m \leq n < c^{m+1}$; es decir, $m = \lfloor \log_c n \rfloor$. Seleccionando las apropiadas constantes adicionales b' y b'', y operando, obtenemos la acotación siguiente:

$$b' \cdot n^k \cdot \sum_{i=0}^{m-1} \left(\frac{a}{c^k}\right)^i \leq T_2(n) \leq b'' \cdot n^k \cdot \sum_{i=0}^{m} \left(\frac{a}{c^k}\right)^i$$

Para $a < c^k$, es decir, $(a/c^k) < 1$, los sumatorios están acotados inferior y superiormente por constantes, ya que la serie $\sum_{i=0}^{\infty}(a/c^k)^i$ es convergente, y por tanto tenemos que $T_2(n) \in \Theta(n^k)$.

Para $a = c^k$, se tiene que $\sum_{i=0}^{m}(a/c^k)^i = \sum_{i=0}^{m} 1 = m = \lfloor \log_c n \rfloor$, y obtenemos $T_2(n) \in \Theta(n^k \cdot \log n)$.

Finalmente, para $a > c^k$, es decir, $(a/c^k) > 1$, dado que el segundo sumatorio describe una progresión geométrica cuya suma es conocida, tenemos

$$\sum_{i=0}^{m} \left(\frac{a}{c^k}\right)^i = \frac{(a/c^k)^{m+1} - 1}{(a/c^k) - 1} \in \Theta\left(\left(\frac{a}{c^k}\right)^{\log_c n}\right)$$

(obsérvese que es preciso usar la condición de que $(a/c^k) \neq 1$). El primer sumatorio, de límite $m-1$, está en el mismo orden de crecimiento por un razonamiento análogo; obtenemos que $T_2(n) \in \Theta(n^k \cdot (a/c^k)^{\log_c n})$, expresión que mediante manipulaciones algebraicas se puede simplificar a $T_2(n) \in \Theta(n^{\log_c a})$. □

Obsérvese que en caso de no tener igualdad sino sólo una desigualdad, aún podemos deducir propiedades de las funciones estudiadas. Por ejemplo, supongamos que sabemos que $T_3(n)$ cumple la siguiente desigualdad:

$$T_3(n) \leq a \cdot T_3(n/c) + b \cdot n^k$$

para $c \leq n$; entonces, de los pasos dados en la demostración precedente, se pueden conservar los suficientes para afirmar que

$$T_3(n) \in \begin{cases} O(n^k) & \text{si } a < c^k \\ O(n^k \cdot \log n) & \text{si } a = c^k \\ O(n^{\log_c a}) & \text{si } a > c^k \end{cases}$$

Así pues, para obtener una cota superior $O(\ )$ basta con una cota superior recurrente; de manera análoga, de una cota inferior recurrente se puede obtener una cota inferior absoluta para la función en términos de $\Omega_K(\ )$.

La aplicación usual de este resultado será como sigue: se deseará calcular el crecimiento del tiempo de ejecución de un programa recursivo; llamemos $T(n)$ a este tiempo de ejecución. En este cálculo intervendrán algunas operaciones más o menos complejas pero independientes de las llamadas recursivas; supongamos que el tiempo requerido por estas operaciones es $b \cdot n^k$. Adicionalmente, tendremos a llamadas recursivas sobre datos más pequeños; si el tamaño de los datos decrece en progresión aritmética, tendremos que el tiempo total es $T(n) = a \cdot T(n-c) + b \cdot n^k$, y podremos aplicar uno de los tres primeros casos del resultado anterior. Si el tamaño de los datos decrece en progresión geométrica, tendremos que el tiempo total es $T(n) = a \cdot T(n/c) + b \cdot n^k$, y podremos aplicar uno de los tres últimos casos.

Obsérvese que el primero de todos los casos expuestos se presenta tan sólo a efectos de completar el teorema. Será inútil en este tipo de aplicaciones, porque no tiene sentido un programa recursivo en el que el número de llamadas recursivas sea inferior a la unidad.

Los valores más frecuentes para k serán $k = 0$ y $k = 1$. El primero de los casos corresponde a un programa recursivo en el que aparte de las llamadas recursivas prácticamente no hay operaciones que consuman un tiempo apreciable, y por tanto el tiempo adicional está acotado por una constante ($b = b \cdot n^0$). En el segundo caso, las operaciones adicionales requieren un tiempo lineal ($b \cdot n^1$).

Nótese finalmente la sustancial diferencia entre dos programas que provoquen el mismo número de llamadas recursivas, con similares costes adicionales, pero en los que los datos decrezcan aritméticamente en uno y geométricamente en otro: comparando los crecimientos asintóticos de los costes predichos por el análisis anterior, vemos que el segundo proporciona

unos órdenes de magnitud sustancialmente inferiores al primero, en algunos casos presentando una diferencia tan notable como un paso de exponencial a polinómico. Por tanto, un decrecimiento geométrico será preferible en la práctica totalidad de los casos, pues el único riesgo de ser inadecuado radica en una constante escondida alta, peligro bastante infrecuente en la práctica.

Referencias

- A. V. Aho, J. E. Hopcroft, J. D. Ullman: "Data structures and algorithms", Addison-Wesley 1983. Traducción castellana: Addison-Wesley Iberoamericana. En particular: secciones 1.4 y 1.5 y capítulo 9?.

- G. Brassard, P. Bratley: "Algorithmique: conception et analyse", Masson 1987. Traducción castellana: Masson. En particular: secciones 1.3 a 1.6.

- D. E. Knuth: "The art of computer programming 1: Fundamental algorithms", Addison-Wesley 1973 (2a edición). Traducción castellana: Masson. En particular: sección 1.2.11.1.

- K. Mehlhorn: "Data structures and algorithms 1: Sorting and searching", Springer-Verlag EATCS Monographs 1984. En particular: sección I.6.

- P. M. B. Vitányi, L. G. L. T. Meertens: "Big omega versus the wild functions", Stitching Mathematisch Centrum Amsterdam IW 239/83, 1983.

- N. Wirth: "Algorithms and data structures", Prentice-Hall 1986. Traducción castellana: Prentice-Hall Hispanoamericana.