Análisis de la complejidad de algoritmos

Alberto Verdejo

Dpto. de Sistemas Informáticos y Computación

Universidad Complutense de Madrid

Octubre 2008

Bibliografía

- R. Peña. *Diseño de programas. Formalismo y abstracción*. Tercera edición. Prentice Hall, 2005.
  Capítulo 1
- N. Martí, M. Palomino y A. Verdejo. *Introducción a la computación*. Anaya, 2006.
  Capítulo 4
- N. Martí Oliet, C. Segura Díaz y J. A. Verdejo López. *Especificación, derivación y análisis de algoritmos: Ejercicios resueltos*. Colección Prentice Practica, Pearson/Prentice Hall, 2006.
  Capítulo 3
- R. Neapolitan y K. Naimipour. *Foundations of Algorithms using C++ pseudocode*. Tercera edición. Jones and Bartlett Publishers, 2004.
  Capítulo 1

Una leyenda ajedrecística

Mucho tiempo atrás, el espabilado visir Sissa ben Dahir inventó el juego del ajedrez para el rey Shirham de la India. El rey ofreció a Sissa la posibilidad de elegir su propia recompensa. Sissa le dijo al rey que podía recompensarle en trigo o bien con una cantidad equivalente a la cosecha de trigo en su reino de dos años, o bien con una cantidad de trigo que se calcularía de la siguiente forma:

- un grano de trigo en la primera casilla de un tablero de ajedrez,
- más dos granos de trigo en la segunda casilla,
- más cuatro granos de trigo en la tercera casilla,
- y así sucesivamente, duplicando el número de granos en cada casilla, hasta llegar a la última casilla.

El rey pensó que la primera posibilidad era demasiado cara mientras que la segunda, medida además en simples granos de trigo, daba la impresión de serle claramente favorable.

Así que sin pensárselo dos veces pidió que trajeran un saco de trigo para hacer la cuenta sobre el tablero de ajedrez y recompensar inmediatamente al visir.

¿Es una buena elección?

El número de granos en la primera fila resultó ser:

$$2^0 + 2^1 + 2^2 + 2^3 + 2^4 + 2^5 + 2^6 + 2^7 = 255$$

La cantidad de granos en las dos primeras filas es:

$$\sum_{i=0}^{15} 2^i = 2^{16} - 1 = 65\,535$$

Al llegar a la tercera fila el rey empezó a pensar que su elección no había sido acertada, pues para llenar las tres filas necesitaba

$$\sum_{i=0}^{23} 2^i = 2^{24} - 1 = 16\,777\,216$$

granos, que pesan alrededor de 600 kilos ...

Endeudado hasta las cejas

En efecto, para rellenar las 64 casillas del tablero hacen falta

$$\sum_{i=0}^{63} 2^i = 2^{64} - 1 = 18\,446\,744\,073\,709\,551\,615 \approx 1{,}84 * 10^{19}$$

granos, cantidad equivalente a las cosechas mundiales actuales de 1000 años!!.

La función $2^n - 1$ (exponencial) representa el número de granos adeudados en función del número n de casillas a rellenar. Toma valores desmesurados aunque el número de casillas sea pequeño.

El coste en tiempo de algunos algoritmos expresado en función del tamaño de los datos de entrada es también exponencial. Por ello es importante estudiar el coste de los algoritmos y ser capaces de comparar los costes de algoritmos que resuelven un mismo problema.

Motivación

- Entendemos por eficiencia el rendimiento de una actividad en relación con el consumo de un cierto recurso. Es diferente de la efectividad.
- Ejemplo para ver la importancia de que el coste del algoritmo sea pequeño.

n	$\log_{10} n$	n	$n \log_{10} n$	n^2	n^3	2^n
10	1 ms	10 ms	10 ms	0,1 s	1 s	1,02 s
10^2	2 ms	0,1 s	0,2 s	10 s	16,67 m	$4{,}02 * 10^{20}$ sig
10^3	3 ms	1 s	3 s	16,67 m	11,57 d	$3{,}4 * 10^{291}$ sig
10^4	4 ms	10 s	40 s	1,16 d	31,71 a	$6{,}3 * 10^{3000}$ sig
10^5	5 ms	1,67 m	8,33 m	115,74 d	317,1 sig	$3{,}16 * 10^{30093}$ sig
10^6	6 ms	16,67 m	1,67 h	31,71 a	317 097,9 sig	$3{,}1 * 10^{301020}$ sig

- Es un error pensar que basta esperar algunos años para que algoritmos tan costosos se puedan ejecutar con un coste en tiempo razonable.

¿Qué medimos y cómo?

- La eficiencia es mayor cuanto menor es la complejidad o el coste (consumo de recursos).
- Necesitamos determinar cómo se ha de medir el coste de un algoritmo, de forma que sea posible compararlo con otros que resuelven el mismo problema y decidir cuál de todos es el más eficiente.
- Una posibilidad para medir el coste de un algoritmo es contar cuántas instrucciones de cada tipo se ejecutan, multiplicar este número por el tiempo que emplea la instrucción en ejecutarse, y realizar la suma para los diferentes tipos.

$$\begin{aligned} t_a &= \text{tiempo de asignación} \\ t_c &= \text{tiempo de comparación} \\ t_i &= \text{tiempo de incremento} \\ t_v &= \text{tiempo de acceso a un vector} \end{aligned}$$

Ejemplo

Ordenación por selección del vector $v[1..n]$

para $i = 1$ **hasta** $n - 1$ **hacer**
  $pmin := i$;
  para $j = i + 1$ **hasta** n **hacer**
    si $v[j] < v[pmin]$ **entonces** $pmin := j$ **fsi**
  fpara ;
  `intercambiar`$(v[i], v[pmin])$
fpara

- control del primer bucle: $t_a + (n-1)t_i + nt_c$
- primera asignación: $(n-1)t_a$
- control del bucle interno, para cada i: $t_a + (n-i)t_i + (n-i+1)t_c$
- instrucción **si**, para cada i, tiempo mínimo: $(n-i)(2t_v + t_c)$
  tiempo máximo: $(n-i)(2t_v + t_c) + (n-i)t_a$
- intercambiar: $(n-1)(2t_v + 3t_a)$

Ejemplo

El bucle interno en total en el caso peor, el más desfavorable,

$$\sum_{i=1}^{n-1}(t_a + t_c + (n-i)(t_i + 2t_c + 2t_v + t_a))$$

Para no cansarnos, concluimos que

$$T_{\text{mín}} = An^2 - Bn + C$$
$$T_{\text{máx}} = A'n^2 - B'n + C'$$

Factores

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

1. El tamaño de los datos de entrada. Por ejemplo:
   - Para un vector: su longitud.
   - Para un número natural: su valor o el número de dígitos.
   - Para un grafo: el número de vértices y/o el número de aristas.
2. El contenido de esos datos.
3. El código generado por el compilador y el computador concreto utilizados.

- Si el tiempo que tarda un algoritmo A en procesar una entrada concreta $\overline{x}$ lo denotamos por $t_A(\overline{x})$, definimos la complejidad de A en el caso peor como

$$T_A(n) = \text{máx}\{t_A(\overline{x}) \mid \overline{x} \text{ de tamaño } n\}$$

- Otra posibilidad es realizar un análisis de la eficiencia en el caso promedio. Para ello necesitamos conocer el tiempo de ejecución de cada posible ejemplar y la frecuencia con que se presenta, es decir, su distribución de probabilidades.
  Definimos la complejidad de un algoritmo A en el caso promedio como

$$TM_A(n) = \sum_{\overline{x} \text{ de tamaño } n} p(\overline{x}) t_A(\overline{x})$$

  siendo $p(\overline{x}) \in [0..1]$ la probabilidad de que la entrada sea $\overline{x}$.

Medidas asintóticas

El único factor determinante en el coste de un algoritmo es el tamaño de los datos de entrada.

Por eso, trabajamos con funciones $f : \mathbb{N} \longrightarrow \mathbb{R}_0^+$.

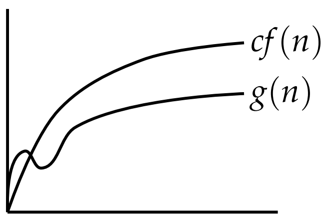
No nos importa tanto las funciones concretas, sino la forma en la que crecen.

Definición El conjunto de las funciones del orden de $f(n)$, denotado $O(f(n))$, se define como

$$O(f(n)) = \{g : \mathbb{N} \longrightarrow \mathbb{R}_0^+ \mid \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}. \forall n \geq n_0. g(n) \leq cf(n)\}$$

Asimismo, diremos que una función g *es del orden de* $f(n)$ cuando $g \in O(f(n))$.

Decimos que el conjunto $O(f(n))$ define un orden de complejidad.



Si el tiempo de un algoritmo está descrito por una función $T(n) \in O(f(n))$ diremos que el tiempo de ejecución del algoritmo es *del orden de* $f(n)$.

Ejemplos

- $\log n \in O(n)$
  Encontramos $n_0 = 1$ y $c = 1$ tal que $\forall n \geq 1. \log n \leq n$

  Lo demostramos por inducción.

  Base: $n = 1$, $\log 1 = 0 \leq 1$

  Paso inductivo: h.i. $\log n \leq n$

  $\log(n+1) \leq \log 2n \leq \log 2 + \log n = 1 + \log n \leq^{h.i.} n + 1$
- $(n+1)^2 \in O(n^2)$
  Demostramos por inducción que $\forall n \geq 1.(n+1)^2 \leq 4n^2$

  Base: $n = 1$, $(1+1)^2 \leq 4 \cdot 1^2$

  Paso inductivo: h.i. $(n+1)^2 \leq 4n^2$

  Para $n+1$

$$
\begin{array}{rcl}
(n+1+1)^2 & \leq & 4(n+1)^2 \\
(n+1)^2 + 1 + 2(n+1) & \leq & 4n^2 + 4 + 8n \\
(n+1)^2 & \leq & 4n^2 + \underbrace{6n+1}_{\geq 0}
\end{array}
$$

Ejemplos

- $3^n \notin O(2^n)$
  Si perteneciera, tendríamos $c \in \mathbb{R}^+$, $n_0 \in \mathbb{N}$ tal que $3^n \leq c \cdot 2^n$ para todo $n \geq n_0$.

  Esto implica que $(\frac{3}{2})^n \leq c$ para todo $n \geq n_0$.

  Pero esto es falso porque dado c cualquiera, basta tomar $n > \log_{1,5} c$ para que $(\frac{3}{2})^n > c$, es decir, no se puede acotar superiormente.

Propiedades

- $O(a \cdot f(n)) = O(f(n))$ con $a \in \mathbb{R}^+$

  $(\subseteq)$ $g \in O(a \cdot f(n)) \Leftrightarrow \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}$ tal que $\forall n \geq n_0.g(n) \leq c \cdot a \cdot f(n)$
  Tomando $c' = c \cdot a$ se cumple que $\forall n \geq n_0.g(n) \leq c' \cdot f(n)$, luego $g \in O(f(n))$

  $(\supseteq)$ $g \in O(f(n)) \Leftrightarrow \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}$ tal que $\forall n \geq n_0.g(n) \leq c \cdot f(n)$
  Entonces tomando $c' = \frac{c}{a}$ se cumple que $\forall n \geq n_0.g(n) \leq c' \cdot a \cdot f(n)$, luego $g \in O(a \cdot f(n))$

- La base del logaritmo no importa: $O(\log_a n) = O(\log_b n)$, con $a, b > 1$.

$$\log_b n = \frac{\log_a n}{\log_a b}$$

Propiedades

- Si $f \in O(g)$ y $g \in O(h)$, entonces $f \in O(h)$.

  $f \in O(g) \Rightarrow \exists c_1 \in \mathbb{R}^+, n_1 \in \mathbb{N}$ tal que $\forall n \geq n_1.f(n) \leq c_1 \cdot g(n)$
  $g \in O(h) \Rightarrow \exists c_2 \in \mathbb{R}^+, n_2 \in \mathbb{N}$ tal que $\forall n \geq n_2.g(n) \leq c_2 \cdot h(n)$

  Tomando $n_0 = \text{máx}(n_1, n_2)$ y $c = c_1 \cdot c_2$, se cumple

  $$\forall n \geq n_0.f(n) \leq c_1 \cdot g(n) \leq c_1 \cdot c_2 \cdot h(n)$$

  Y por tanto $f \in O(h)$.

Propiedades

- Regla de la suma: $O(f+g) = O(\max(f,g))$.
  $(\subseteq)$ $h \in O(f+g) \Rightarrow \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}. \forall n \geq n_0. h(n) \leq c \cdot (f(n)+g(n))$
  Pero $f \leq \max(f,g)$ y $g \leq \max(f,g)$, luego
  $$h(n) \leq c \cdot (\max(f(n),g(n)) + \max(f(n),g(n))) = 2 \cdot c \cdot \max(f(n),g(n))$$
  Luego tomando $c' = 2 \cdot c$ se cumple que
  $\forall n \geq n_0. h(n) \leq c' \cdot \max(f(n),g(n))$ y por tanto $h \in O(\max(f,g))$.

  $(\supseteq)$ $h \in O(\max(f,g)) \Rightarrow$
  $\exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}. \forall n \geq n_0. h(n) \leq c \cdot \max(f(n),g(n))$
  Pero $\max(f,g) \leq f+g$, luego $h \in O(f+g)$ trivialmente.

- Regla del producto: Si $g_1 \in O(f_1)$ y $g_2 \in O(f_2)$, entonces $g_1 \cdot g_2 \in O(f_1 \cdot f_2)$.

Teorema del límite

- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = m \in \mathbb{R}^+ \quad \Rightarrow \quad f \in O(g)$ y $g \in O(f) \quad \Leftrightarrow \quad O(f) = O(g)$
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \quad \Rightarrow \quad f \in O(g)$ y $g \notin O(f) \quad \Leftrightarrow \quad O(f) \subset O(g)$
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty \quad \Rightarrow \quad f \notin O(g)$ y $g \in O(f) \quad \Leftrightarrow \quad O(f) \supset O(g)$

Demostración de $\Rightarrow$. $\forall \epsilon > 0. \exists n_0 \in \mathbb{N}$ tal que $\forall n \geq n_0. |\frac{f(n)}{g(n)}| < \epsilon$.

Tomando $\epsilon = 1$, tenemos $\forall n \geq n_0. f(n) < g(n) \Rightarrow f \in O(g)$

Demostramos $g \notin O(f)$ por reducción al absurdo.

$g \in O(f) \Leftrightarrow \exists n_0 \in \mathbb{N}, \exists c \in \mathbb{R}^+$ tal que $\forall n \geq n_0 \,.\, g(n) \leq cf(n)$
$\Leftrightarrow \exists n_0 \in \mathbb{N}, \exists c \in \mathbb{R}^+$ tal que $\forall n \geq n_0 \,.\, \frac{1}{c} \leq \frac{f(n)}{g(n)}$

$$\frac{1}{c} = \lim_{n\to\infty} \frac{1}{c} \leq \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

$\frac{1}{c} \leq 0$ contradicción con $c \in \mathbb{R}^+$.

Ejemplos

- $\log n \in O(n)$

$$\lim_{n\to\infty} \frac{n}{\log n} = [\ \tfrac{\infty}{\infty}\ ] =^{L'Hopital} \lim_{n\to\infty} \frac{n\ln 2}{\ln n} = \lim_{n\to\infty} \frac{\ln 2}{1/n} = \lim_{n\to\infty} n\ln 2 = \infty$$

- $P(x) = a_k x^k + a_{k-1}x^{k-1} + \ldots + a_1 x + a_0$, con $a_k \in \mathbb{R}^+$, $P(x) \in O(x^k)$

$$\lim_{x\to\infty} \frac{P(x)}{x^k} = a_k > 0$$

- $O(n^k) \subset O(2^n)$

$$\begin{aligned} \lim_{n\to\infty} \tfrac{2^n}{n^k} &= [\ \tfrac{\infty}{\infty}\ ] =^{L'Hopital} \\ &= \lim_{n\to\infty} \tfrac{2^n \ln 2}{kn^{k-1}} = \lim_{n\to\infty} \tfrac{2^n (\ln 2)^2}{k(k-1)n^{k-2}} = (k \text{ veces}) = \\ &= \lim_{n\to\infty} \tfrac{2^n (\ln 2)^k}{k! n^0} = \tfrac{(\ln 2)^k}{k!} \lim_{n\to\infty} 2^n = \infty \end{aligned}$$

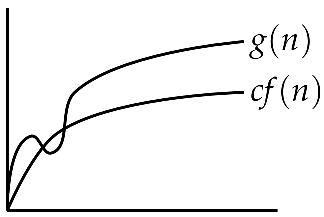
- $O(2^n) \subset O(n!)$

$$\lim_{n\to\infty} \frac{n!}{2^n} = \lim_{n\to\infty} \frac{n}{2}\frac{n-1}{2} \cdots \frac{4}{2}\frac{3}{2}\frac{2}{2}\frac{1}{2} \geq \lim_{n\to\infty} \frac{4}{2}\frac{4}{2} \cdots \frac{4}{2}\frac{3}{2}\frac{2}{2}\frac{1}{2} = \frac{3}{4} \lim_{n\to\infty} 2^{n-3} = \infty$$

Cotas inferiores

Definición Sea $f : \mathbb{N} \longrightarrow \mathbb{R}_0^+$. El conjunto $\Omega(f(n))$, leído omega de $f(n)$, se define como

$$\Omega(f(n)) = \{g : \mathbb{N} \longrightarrow \mathbb{R}_0^+ \mid \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}. \forall n \geq n_0 \,.\, g(n) \geq cf(n)\}$$



Cuando decimos que el coste de un algoritmo está en $\Omega(f(n))$ lo que estamos diciendo es que la complejidad del algoritmo *no es mejor* que la representada por la función f.

Principio de dualidad $g \in \Omega(f) \Leftrightarrow f \in O(g)$

Teorema del límite

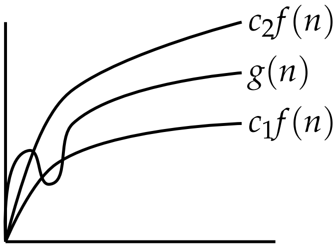
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = m \in \mathbb{R}^+ \quad \Rightarrow \quad g \in \Omega(f)$ y $f \in \Omega(g)$
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \quad \Rightarrow \quad g \in \Omega(f)$ y $f \notin \Omega(g)$
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty \quad \Rightarrow \quad g \notin \Omega(f)$ y $f \in \Omega(g)$

Orden exacto

Definición El conjunto de funciones $\Theta(f(n))$, leído del orden exacto de $f(n)$, se define como

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

$$\Theta(f(n)) = \{g : \mathbb{N} \longrightarrow \mathbb{R}_0^+ \mid \exists c_1, c_2 \in \mathbb{R}^+, n_0 \in \mathbb{N}. \forall n \geq n_0 \,.\, c_1 f(n) \leq g(n) \leq c_2 f(n)\}$$



Teorema del límite

- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = m \in \mathbb{R}^+ \quad \Rightarrow \quad g \in \Theta(f)$ y $f \in \Theta(g)$
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \quad \Rightarrow \quad f \in O(g)$ pero $f \notin \Theta(g)$
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty \quad \Rightarrow \quad g \in O(f)$ pero $g \notin \Theta(f)$

Ejemplo:
$P(x) = a_k x^k + a_{k-1}x^{k-1} + \ldots + a_1 x + a_0$, con $a_k > 0$,
$Q(x) = b_l x^l + b_{l-1}x^{l-1} + \ldots + b_1 x + b_0$, con $b_l > 0$, entonces

$$\lim_{x\to\infty} \frac{P(x)}{Q(x)} = 0 \text{ si } k < l \quad \Rightarrow \quad P(x) \in O(Q(x))$$
$$\lim_{x\to\infty} \frac{P(x)}{Q(x)} = \frac{a_k}{b_l} \text{ si } k = l \quad \Rightarrow \quad P(x) \in \Theta(Q(x))$$
$$\lim_{x\to\infty} \frac{P(x)}{Q(x)} = \infty \text{ si } k > l \quad \Rightarrow \quad P(x) \in \Omega(Q(x))$$

Jerarquía de órdenes de complejidad

$$\underbrace{\underbrace{O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^2)}_{\text{razonables en la práctica}} \subset \ldots \subset O(n^k)}_{\text{tratables}} \subset \underbrace{\ldots \subset O(2^n) \subset O(n!)}_{\text{intratables}}$$

La notación $O(f)$ nos da una *cota superior* del tiempo de ejecución $T(n)$ de un algoritmo.

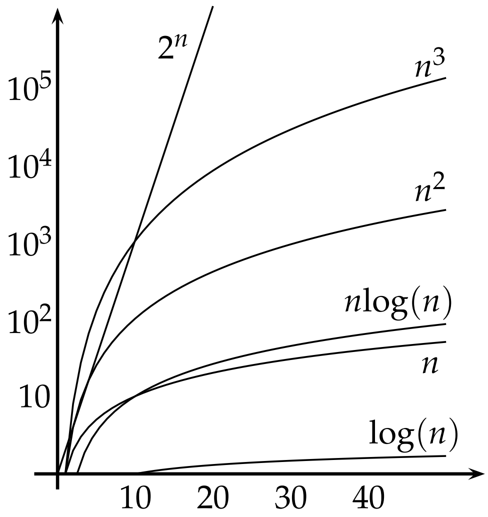
Normalmente estaremos interesados en la menor función $f(n)$ tal que

$$T(n) \in O(f(n)).$$

Ejemplos: cotas superiores hay muchas, pero algunas son poco informativas

$$3n \in O(n)$$
$$3n \in O(n^2)$$
$$3n \in O(2^n)$$

Ordenes de complejidad



Supongamos que tenemos 6 algoritmos diferentes para resolver el mismo problema tales que su menor cota superior está en $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, $O(n^3)$ y $O(2^n)$.

Supongamos que para un tamaño $n = 100$ todos tardan 1 hora en ejecutarse.

¿Qué ocurre si duplicamos el tamaño de los datos?

$T(n)$	$n = 100$	$n = 200$
$k_1 \cdot \log n$	$1h.$	$1,15h.$
$k_2 \cdot n$	$1h.$	$2h.$
$k_3 \cdot n \log n$	$1h.$	$2,3h.$
$k_4 \cdot n^2$	$1h.$	$4h.$
$k_5 \cdot n^3$	$1h.$	$8h.$
$k_6 \cdot 2^n$	$1h.$	$1,27 \cdot 10^{30}h.$

¿Qué ocurre si duplicamos la velocidad del computador? O lo que es lo mismo, ¿qué ocurre si duplicamos el tiempo disponible?

$T(n)$	$t = 1h.$	$t = 2h.$
$k_1 \cdot \log n$	$n = 100$	$n = 10000$
$k_2 \cdot n$	$n = 100$	$n = 200$
$k_3 \cdot n \log n$	$n = 100$	$n = 178$
$k_4 \cdot n^2$	$n = 100$	$n = 141$
$k_5 \cdot n^3$	$n = 100$	$n = 126$
$k_6 \cdot 2^n$	$n = 100$	$n = 101$

Análisis de algoritmos

Podemos simplificar el cálculo del coste (en el caso peor) de un algoritmo gracias a la teoría presentada.

1. Las instrucciones de asignación, entrada/salida, o expresiones aritméticas tienen un coste en $\Theta(1)$.
2. Para calcular el coste de una composición secuencial se utiliza la regla de la suma. Si el coste de S_1 está en $\Theta(f_1(n))$ y el coste de S_2 está en $\Theta(f_2(n))$, entonces el coste de S_1 ; S_2 está en $\Theta(\max(f_1(n), f_2(n)))$.
3. El coste de una instrucción condicional **si** B **entonces** S_1 **si no** S_2 **fsi** está en $\Theta(\max(f_B(n), f_1(n), f_2(n)))$.
4. Para calcular el coste de una instrucción iterativa

   mientras B **hacer** S **fmientras**

   utilizamos la regla del producto. Si el número de iteraciones está en $\Theta(f_{iter}(n))$, el coste total del bucle está en $\Theta(f_{B,S}(n) \cdot f_{iter}(n))$.
   Si el coste de cada iteración es distinto, realizamos una suma desde 1 hasta $f_{iter}(n)$ de los costes individuales.

Ejemplos

Producto de matrices cuadradas.

fun $\texttt{producto}(A[1..n,1..n], B[1..n,1..n]$ **de** *ent*)
  dev $C[1..n,1..n]$ **de** *ent*
var i, j, k : *nat*, s : *ent*
  para $i = 1$ **hasta** n **hacer**
    para $j = 1$ **hasta** n **hacer**
      $s := 0$;
      para $k = 1$ **hasta** n **hacer**
        $s := s + A[i,k] * B[k,j]$
      fpara ;
      $C[i,j] := s$
    fpara
  fpara
ffun

$$T(n) \in \Theta(n^3)$$

Ejemplos

Ordenación por selección.

proc $\texttt{ord-selección}(V[1..n]$ **de** *ent*)
  para $i = 1$ **hasta** $n - 1$ **hacer**
    $pmin := i$;
    para $j = i + 1$ **hasta** n **hacer**
      si $V[j] < V[pmin]$ **entonces** $pmin := j$ **fsi**
    fpara ;
    $\texttt{intercambiar}(V[i], V[pmin])$
  fpara
fproc

$$T(n) \in \Theta(\sum_{i=1}^{n-1}(n-i)) = \Theta(n^2)$$

Ejemplos

Determinar si una matriz cuadrada es simétrica.

fun `simétrica?`$(V[1..n,1..n]$ **de** *ent*$)$ **dev** b : *bool*
var i,j : *nat*
  b := `cierto` ; $i := 1$;
  mientras $i \leq n \ \wedge \ b$ **hacer**
    $j := i+1$;
    mientras $j \leq n \ \wedge \ b$ **hacer**
      $b := (V[i,j] = V[j,i])$;
      $j := j+1$
    fmientras ;
    $i := i+1$
  fmientras
ffun

$$T_{\text{mín}}(n) \in \Theta(1) \qquad T_{\text{máx}}(n) \in \Theta(n^2)$$

Instrucción crítica

Se puede simplificar más el cálculo si hacemos uso del concepto de instrucción crítica: instrucción que más veces se ejecuta.

Calcular el número de veces que se ejecuta la instrucción crítica.

proc `ord-selección`$(V[1..n]$ **de** *ent*$)$
  para $i = 1$ **hasta** $n-1$ **hacer**
    $pmin := i$;
    para $j = i+1$ **hasta** n **hacer**
      si $V[j] < V[pmin]$ **entonces** $pmin := j$ **fsi**
    fpara ;
    `intercambiar`$(V[i], V[pmin])$
  fpara
fproc

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

Ejemplo

proc `ord-inserción`$(V[1..n]$ **de** *ent*$)$
  para $i = 2$ **hasta** n **hacer**
    $elem := V[i]$;
    $j := i - 1$;
    mientras $j > 0 \wedge_c elem < V[j]$ **hacer**
      $V[j+1] := V[j]$;
      $j := j - 1$
    fmientras ;
    $V[j+1] := elem$
  fpara
fproc

caso peor: $T_{\text{máx}}(n) = \sum_{i=2}^{n} i = \frac{(n+2)(n-1)}{2} \in \Theta(n^2)$

caso mejor: $T_{\text{mín}}(n) = \sum_{i=2}^{n} 1 = n - 1 \in \Theta(n)$

caso promedio: $\Theta(n^2)$

Algoritmos recursivos

Cuando se analiza la complejidad de un algoritmo recursivo es frecuente que aparezcan funciones de coste también recursivas, llamadas recurrencias.

fun `factorial`$(n : nat)$ **dev** $f : nat$
  casos
    $n = 0 \rightarrow f := 1$
  $\square$ $n > 0 \rightarrow$
      $f := n * \texttt{factorial}(n-1)$
  fcasos
ffun

$$T(n) = \begin{cases} k_1 & \text{si } n = 0 \\ T(n-1) + k_2 & \text{si } n > 0 \end{cases}$$

Queremos obtener el *orden de complejidad* de $T(n)$. Dos formas:

- Mediante despliegue: obtener fórmula explícita de $T(n)$.
- Utilizando los teoremas que veremos: disminución del tamaño del problema por sustracción o por división.

Despliegue de recurrencias

El objetivo es conseguir una fórmula explícita (en función de n) de $T(n)$.

Seguimos tres pasos:

Despliegue Sustituimos T por la parte derecha de la ecuación tantas veces como sea necesario hasta encontrar una fórmula que dependa del número de despliegues (o llamadas recursivas) i.

Postulado Obtenemos el valor de i que hace que T desaparezca (caso básico), y en la fórmula sustituimos i por ese valor, obteniendo la fórmula explícita T^* (que solo depende de n).

Demostración Demostramos (por inducción) que $T = T^*$.

Ejemplo: factorial

$$T(n) = \begin{cases} k_1 & \text{si } n = 0 \\ T(n-1) + k_2 & \text{si } n > 0 \end{cases}$$

$$\begin{array}{rclcl} T(n) & \overset{1}{=} & T(n-1) + k_2 & & \\ & \overset{2}{=} & T(n-2) + k_2 + k_2 & = & T(n-2) + 2 \cdot k_2 \\ & \overset{3}{=} & T(n-3) + k_2 + 2 \cdot k_2 & = & T(n-3) + 3 \cdot k_2 \\ & \vdots & & & \\ & \overset{i}{=} & T(n-i) + i \cdot k_2 & & \\ & \vdots & & & \\ & \overset{n}{=} & T(0) + n \cdot k_2 & = & k_2 n + k_1 = T^*(n) \in \Theta(n) \end{array}$$

Demostramos que $T(n) = T^*(n)$ para todo $n \geq 0$.
Base: $T(0) = k_1 = T^*(0)$
H.I.: $T(n) = T^*(n)$
Para $n+1$:
$T(n+1) = T(n) + k_2 \overset{h.i.}{=} k_2 n + k_1 + k_2 = (n+1)k_2 + k_1 = T^*(n+1)$

Ejemplos

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ 3T(n-1)+2 & \text{si } n \geq 2 \end{cases}$$

$$\begin{array}{rcl}
T(n) & \stackrel{1}{=} & 3T(n-1)+2 \\
& \stackrel{2}{=} & 3(3T(n-2)+2)+2 = 3^2T(n-2)+3\cdot 2+2 \\
& \stackrel{3}{=} & 3^2(3T(n-3)+2)+3\cdot 2+2 = 3^3T(n-3)+3^2\cdot 2+3\cdot 2+2 \\
& \vdots & \\
& \stackrel{i}{=} & 3^iT(n-i)+\sum_{j=0}^{i-1} 3^j\cdot 2 \\
& \vdots & \\
& \stackrel{n-1}{=} & 3^{n-1}T(1)+\sum_{j=0}^{n-2} 3^j\cdot 2 = 3^{n-1}+2\cdot\dfrac{3\cdot 3^{n-2}-3^0}{3-1} \\
& = & 2\cdot 3^{n-1}-1 \in \Theta(3^n)
\end{array}$$

Ejemplos

$$T(n) = \begin{cases} 4 & \text{si } n = 1 \\ 2T(n/2)+3n+2 & \text{si } n \geq 2, \text{ potencia de } 2 \end{cases}$$

Haciendo sucesivos desplegados obtenemos

$$\begin{array}{rcl}
T(n) & \stackrel{1}{=} & 2T(\frac{n}{2})+3n+2 \\
& \stackrel{2}{=} & 2(2T(\frac{n/2}{2})+3\frac{n}{2}+2)+3n+2 = 2^2T(\frac{n}{2^2})+2\cdot 3n+2^2+2 \\
& \stackrel{3}{=} & 2^2(2T(\frac{n/2^2}{2})+3\frac{n}{2^2}+2)+2\cdot 3n+2^2+2 \\
& = & 2^3T(\frac{n}{2^3})+3\cdot 3n+2^3+2^2+2 \\
& \stackrel{4}{=} & 2^3(2T(\frac{n/2^3}{2})+3\frac{n}{2^3}+2)+3\cdot 3n+2^3+2^2+2 \\
& = & 2^4T(\frac{n}{2^4})+4\cdot 3n+2^4+2^3+2^2+2^1 \\
& \vdots &
\end{array}$$

Ejemplos

$$\begin{array}{cl} \stackrel{i}{=} & 2^i T(\frac{n}{2^i}) + i \cdot 3n + \sum_{j=1}^{i} 2^j \\ = & 2^i T(\frac{n}{2^i}) + i \cdot 3n + 2^{i+1} - 2. \end{array}$$

Al caso básico se llega cuando $\frac{n}{2^i} = 1$, es decir, cuando $i = \log n$.

Por tanto para n potencia de 2,

$$\begin{array}{rcl} T(n) & \stackrel{\log n}{=} & 2^{\log n} T(1) + 3n \log n + 2^{1+\log n} - 2 \\ & = & 4n + 3n \log n + 2n - 2 \\ & = & 3n \log n + 6n - 2 \\ & \in & \Theta(n \log n). \end{array}$$

Teorema de la resta

Cuando

- la descomposición recursiva se obtiene restando una cantidad constante,
- el caso directo tiene coste constante,
- la preparación de las llamadas y la combinación de los resultados tiene coste polinómico,

tenemos

$$T(n) = \begin{cases} k_0 & \text{si } 0 \leq n < b \\ a \cdot T(n-b) + k_1 \cdot n^k & \text{si } n \geq b \end{cases}$$

Entonces

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

Teorema de la división

Si la descomposición se obtiene dividiendo por una cantidad constante $b \geq 2$,

$$T(n) = \begin{cases} k_0 & \text{si } 0 \leq n < b \\ a \cdot T(\frac{n}{b}) + k_1 \cdot n^k & \text{si } n \geq b \end{cases}$$

Entonces

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

El coste es menor cuanto más pequeñas son a y k y más grande es b.

Ejemplo: búsqueda binaria

fun búsqueda-binaria$(V[1..n]$ **de** *ent*$, e :$ *ent*$, c, f :$ *nat*$)$ **dev** $\langle b :$ *bool*$, p :$ *nat*$\rangle$
  si $c > f$ **entonces** $\langle b, p \rangle := \langle$ falso$, c \rangle$
  si no
    $m := (c + f) \text{ div } 2$;
    casos
      $e < V[m] \rightarrow \langle b, p \rangle :=$ búsqueda-binaria$(V, e, c, m - 1)$
      $\square$ $e = V[m] \rightarrow \langle b, p \rangle := \langle$ cierto$, m \rangle$
      $\square$ $e > V[m] \rightarrow \langle b, p \rangle :=$ búsqueda-binaria$(V, e, m + 1, f)$
    fcasos
  fsi
ffun

Coste: tamaño $n = f - c + 1$. Suponiendo n potencia de 2:

$$T(n) = \begin{cases} k_1 & \text{si } n = 0 \\ T(n/2) + k_2 & \text{si } n > 0 \end{cases}$$

$$T(n) \in \Theta(\log n)$$

Ejemplo

Supongamos que el siguiente algoritmo se ejecuta sobre un número natural n que es potencia de 5.

proc `mickey`$(n : nat)$
  si $n = 1$ **entonces** `nada`
  si no $mouse := n/5$;
    para $i = 1$ **hasta** 4 **hacer**
      `mickey`$(mouse)$
    fpara
  fsi
fproc

Queremos conocer el número exacto de divisiones realizadas (en función de n).

$$D(n) = \begin{cases} 0 & n = 1 \\ 4D(n/5) + 1 & n > 1. \end{cases}$$

Ejemplo

Para resolver esta recurrencia, realizamos sucesivos desplegados obteniendo

$$\begin{aligned}
D(n) &\overset{1}{=} 4D(\frac{n}{5}) + 1 \\
&\overset{2}{=} 4(4D(\frac{n/5}{5}) + 1) + 1 = 4^2 D(\frac{n}{5^2}) + 4 + 1 \\
&\overset{3}{=} 4^2(4D(\frac{n/5^2}{5}) + 1) + 4 + 1 = 4^3 D(\frac{n}{5^3}) + 4^2 + 4 + 1 \\
&\overset{4}{=} 4^3(4D(\frac{n/5^3}{5}) + 1) + 4^2 + 4 + 1 = 4^4 D(\frac{n}{5^4}) + 4^3 + 4^2 + 4^1 + 4^0 \\
&\overset{i}{=} 4^i D(\frac{n}{5^i}) + \sum_{j=0}^{i-1} 4^j = 4^i D(\frac{n}{5^i}) + \frac{4^i - 1}{4 - 1}.
\end{aligned}$$

Ejemplo

Al caso básico se llega cuando $\frac{n}{5^i} = 1$, es decir, cuando $i = \log_5 n$.

Sustituyendo en la expresión anterior, obtenemos para n potencia de 5,

$$\begin{aligned} D(n) &\overset{\log_5 n}{=} 4^{\log_5 n} D(1) + \frac{4^{\log_5 n} - 1}{3} \\ &\overset{(*)}{=} \frac{1}{3}(n^{\log_5 4} - 1) \\ &\in \Theta(n^{\log_5 4}). \end{aligned}$$

(*)Nota:

$$\log_4 n = \frac{\log_5 n}{\log_5 4} \Rightarrow \log_4 n * \log_5 4 = \log_5 n \Rightarrow$$

$$\log_4 n^{\log_5 4} = \log_4 4^{\log_5 n} \Rightarrow n^{\log_5 4} = 4^{\log_5 n}$$

Extender resultados para n una potencia a cualquier n

Una función de coste $f(n)$ es eventualmente no decreciente si a partir de cierto punto la función nunca decrece al aumentar n. Es decir, existe un n_0 tal que si $n_1 > n_2 > n_0$, entonces $f(n_1) \geq f(n_2)$.

Una función de coste $f(n)$ es armónica si es eventualmente no decreciente y

$$f(2n) \in \Theta(f(n)).$$

Ejemplos:

$\log n$ es armónica, ya que $\log(2n) = \log 2 + \log n \in \Theta(\log n)$.

2^n *no* es armónica, ya que $2^{2n} = 4^n \notin \Theta(2^n)$.

Teorema: Sea $b \geq 2$, $f(n)$ una función de coste armónica y $T(n)$ una función de coste eventualmente no decreciente. Si

$$T(n) \in \Theta(f(n)) \qquad \text{para } n \text{ potencia de } b$$

entonces

$$T(n) \in \Theta(f(n)).$$

Ejemplo: Supongamos que para un algoritmo obtenemos la siguiente recurrencia

$$T(n) = \begin{cases} 1 & n = 1 \\ T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 1 & n > 1. \end{cases}$$

Cuando n es una potencia de 2, $\left\lfloor \frac{n}{2} \right\rfloor = \frac{n}{2}$. Por el teorema de la división,

$$T(n) \in \Theta(\log n) \qquad \text{para } n \text{ potencia de 2.}$$

Como $\log n$ es armónica y $T(n)$ es eventualmente no decreciente (se puede demostrar por inducción sobre n), podemos concluir que $T(n) \in \Theta(\log n)$ para todo n.

Recurrencias con historia

$$T(n) = \begin{cases} 1 & n = 1 \\ \sum_{i=1}^{n-1} T(i) + n^2 & n \geq 2. \end{cases}$$

Buscamos primero una recurrencia equivalente en cuyo caso recursivo solamente aparezca el valor inmediatamente anterior $T(n-1)$.

Restamos $T(n-1)$ de $T(n)$. Para $n \geq 3$,

$$\begin{aligned} T(n) - T(n-1) &= \sum_{i=1}^{n-1} T(i) + n^2 - \left(\sum_{i=1}^{n-2} T(i) + (n-1)^2\right) \\ &= \sum_{i=1}^{n-2} T(i) + T(n-1) + n^2 - \sum_{i=1}^{n-2} T(i) - (n^2 - 2n + 1) \\ &= T(n-1) + 2n - 1. \end{aligned}$$

Despejando tenemos

$$T(n) = 2T(n-1) + 2n - 1.$$

Recurrencias con historia

Si $n = 2$, la definición original de la recurrencia $T(n)$ da

$$T(2) = \sum_{i=1}^{2-1} T(i) + 2^2 = T(1) + 4 = 1 + 4 = 5,$$

y también

$$2T(n-1) + 2n - 1 = 2T(1) + 2 \cdot 2 - 1 = 2 + 4 - 1 = 5,$$

por lo que la fórmula anterior vale para todo $n \geq 2$.

Aplicando desplegado

$$\begin{array}{rcl} T(n) & \stackrel{1}{=} & 2T(n-1) + 2n - 1 \\ & \stackrel{2}{=} & 2^2T(n-2) + 2^2(n-1) - 2 + 2n - 1 \\ & \stackrel{3}{=} & 2^3T(n-3) + 2^3(n-2) - 2^2 + 2^2(n-1) - 2 + 2n - 1 \\ & \stackrel{i}{=} & 2^iT(n-i) + \sum_{j=0}^{i-1} 2^{j+1}(n-j) - \sum_{j=0}^{i-1} 2^j \\ & = & 2^iT(n-i) + (2n-1)\sum_{j=0}^{i-1} 2^j - 2\sum_{j=0}^{i-1} j2^j. \end{array}$$

Recurrencias con historia

Al caso básico se llega cuando $n - i = 1$; entonces $i = n - 1$ y

$$T(n) \stackrel{n-1}{=} 2^{n-1} + (2n-1)\sum_{j=0}^{n-2} 2^j - 2\sum_{j=0}^{n-2} j2^j.$$

Para simplificar las sumas recordamos las igualdades

$$\begin{array}{rcl} \sum_{j=0}^{n-2} 2^j & = & 2^{n-1} - 1 \\ \sum_{j=0}^{n-2} j2^j & = & (n-3)2^{n-1} + 2. \end{array}$$

Sustituyendo en la expresión anterior queda

$$\begin{array}{rcl} T(n) & = & 2^{n-1} + (2n-1)(2^{n-1} - 1) - 2((n-3)2^{n-1} + 2) \\ & = & (1 + 2n - 1 - 2n + 6)2^{n-1} - (2n-1) - 4 \\ & = & 3 \cdot 2^n - 2n - 3 \\ & \in & \Theta(2^n). \end{array}$$

Limitaciones prácticas

Si tenemos dos algoritmos con costes $T_1(n) = 3n^3$ y $T_2(n) = 600n^2$, ¿cuál es mejor?

En principio es mejor el segundo, ya que $O(n^2) \subset O(n^3)$.

Pero esto es "para valores de n suficientemente grandes".

Aquí $n \geq 200$.

Complejidad en promedio

- Para analizar el coste en el caso promedio necesitamos conocer el tiempo de ejecución de cada posible ejemplar y la frecuencia con que se presenta, es decir, su distribución de probabilidades.
  Definimos la complejidad de un algoritmo A en el caso promedio como
  $$TM_A(n) = \sum_{\overline{x} \text{ de tamaño } n} p(\overline{x}) t_A(\overline{x})$$
  siendo $p(\overline{x}) \in [0..1]$ la probabilidad de que la entrada sea $\overline{x}$.
- **NO** se debe interpretar el análisis del caso medio como el análisis del caso típico. Una media solo es típica si los casos no se desvían demasiado de dicha media, es decir si la desviación típica es pequeña.
- Un análisis en el caso medio es útil porque nos dice cuánto tarda el algoritmo cuando se ejecuta **muchas veces sobre entradas muy diferentes**.
- Por ejemplo, el uso de un algoritmo de ordenación que se usa repetidamente para ordenar todas las posibles entradas. Se puede tolerar una ordenación relativamente lenta (para ciertos casos) si en media el tiempo de ordenación es bueno, ej: quicksort
- En sistemas críticos es mejor el análisis del caso peor.

Complejidad en promedio: búsqueda secuencial

fun `búsqueda-sec`$(V[1..n]$ **de** *ent*$, x :$ *ent*$)$ **dev** $i :$ *nat*
  $i := 1$;
  mientras $i \leq n \wedge_c V[i] \neq x$ **hacer**
    $i := i + 1$
  fmientras
ffun

Caso peor: $\Theta(n)$

Caso mejor: $\Theta(1)$

¿Caso medio?

$$TM_A(n) = \sum_{\overline{x} \text{ de tamaño } n} p(\overline{x}) t_A(\overline{x})$$

Primer análisis: x está con seguridad en V. Todos los elementos son distintos y x puede aparecer en cualquier posición con la misma probabilidad, $\frac{1}{n}$.

Hacemos clases de vectores, según lo que tarde el algoritmo.

Clases
  C_1 si $V[1] = x$
  C_2 si $V[1] \neq x$ y $V[2] = x$
  $\vdots$
  C_n si $(\forall i : 1 \leq i < n : V[i] \neq x) \wedge V[n] = x$

i	probabilidad	comparaciones
1	$1/n$	1
2	$1/n$	2
	$\vdots$	
i	$1/n$	i
	$\vdots$	
n	$1/n$	n

$$TM_A(n) = \frac{1}{n} \sum_{i=1}^{n} i = \frac{n+1}{2}$$

Segundo análisis: x está con probabilidad p, $0 \leq p \leq 1$.

La probabilidad de que esté en la posición i es $\frac{p}{n}$.

Una clase más, $n+1$, $(\forall i : 1 \leq i \leq n : V[i] \neq x)$. Se realizan n comparaciones.

$$\begin{aligned} TM_A(n) &= (\sum_{i=1}^{n} \tfrac{p}{n} i) + (1-p)n = p\tfrac{n+1}{2} + (1-p)n \\ &= \tfrac{(2-p)n+p}{2} \in \Theta(n) \end{aligned}$$

Si $p = 1/2$ entonces $TM_A(n) = 3n/4 + 1/4$, es decir, en media se recorre al menos los $3/4$ del vector.

Complejidad en promedio: ordenación por inserción

proc `ord-inserción`$(V[1..n]$ **de** $ent)$
    para $i = 2$ **hasta** n **hacer**
        $elem := V[i]$;
        $j := i - 1$;
        mientras $j > 0 \wedge_c elem < V[j]$ **hacer**
            $V[j+1] := V[j]$;
            $j := j - 1$
        fmientras ;
        $V[j+1] := elem$
    fpara
fproc

Caso peor: $\Theta(n^2)$

Caso mejor: $\Theta(n)$

¿Caso medio?

Suposiciones:

- Todos los elementos son distintos: $n!$ posibilidades.
- Todas las posibilidades igualmente probables.

1ª posibilidad: Sumar los tiempos de todas las posibilidades y dividir por $n!$.

2ª posibilidad: Analizar el tiempo requerido razonando *probabilísticamente* a medida que vayamos avanzando.

- Definimos el **rango parcial** de $V[i]$ $(2 \leq i \leq n)$ como la posición que ocuparía $V[i]$ si ordenásemos $V[1..i]$.
  Ejemplo: el rango de $V[4]$ en $V = [3, 6, 2, 5, 1, 7, 4]$ es 3.
- El rango parcial de $V[i]$ no depende del orden de los elementos en $V[1..i-1]$.
- Si las $n!$ permutaciones son equiprobables, el rango parcial de $V[i]$ puede variar en $1..i$ con la misma probabilidad para todos los valores.

Fijemos i y supongamos que vamos a entrar en el bucle **mientras** y que el rango parcial de $V[i]$ es k.

Entonces la comprobación del bucle se realiza $i - k + 1$ veces.

El número medio de veces que se efectúa la comprobación para cada i es:

$$c_i = \sum_{k=1}^{i} \frac{1}{i}(i - k + 1) = \frac{i+1}{2}$$

Por ser sucesos independientes para distintos valores de i, el número total medio de comparaciones es:

$$\sum_{i=2}^{n} c_i = \sum_{i=2}^{n} \frac{i+1}{2} = \frac{(n+4)(n-1)}{4} \in \Theta(n^2)$$

Fórmulas útiles para el análisis de algoritmos

Sumatorios

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \approx \frac{1}{2}n^2$$

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \approx \frac{1}{3}n^3$$

$$\sum_{i=1}^{n} i^k \approx \frac{1}{k+1}n^{k+1}$$

$$\sum_{i=0}^{n} a^i = \frac{a^{n+1}-1}{a-1} \qquad \text{si } a \neq 1$$

$$\sum_{i=1}^{n} i2^i = (n-1)2^{n+1} + 2$$

$$\sum_{i=1}^{n} \log i \approx n \log n$$

Fórmulas útiles para el análisis de algoritmos

Propiedades de los logaritmos, $a, b > 1$

$$\log_a 1 = 0$$

$$\log_a a = 1$$

$$\log_a x^y = y \log_a x$$

$$\log_a xy = \log_a x + \log_a y$$

$$\log_a \frac{x}{y} = \log_a x - \log_a y$$

$$a^{\log_b x} = x^{\log_b a}$$

$$\log_a x = \frac{\log_b x}{\log_b a}$$