Capítulo 1

Análisis de la eficiencia de los algoritmos [1]

No puede haber orden cuando hay mucha prisa

Seneca

Resumen: En este tema se muestra la importancia de contar con algoritmos eficientes, se define qué se entiende por coste de un algoritmo y se enseña a comparar las funciones de coste de distintos algoritmos con el fin de decidir cuál de ellos es preferible. Se define una jerarquía de órdenes de complejidad que ilustra la separación entre algoritmos eficientes y los que no lo son.

1. Introducción

⋆ Aproximadamente cada año y medio se duplica el número de instrucciones por segundo que son capaces de ejecutar los computadores. Ello puede inducir a pensar que basta con esperar algunos años para que problemas que hoy necesitan muchas horas de cálculo puedan resolverse en pocos segundos.

⋆ Sin embargo hay algoritmos tan ineficientes que ningún avance en la velocidad de las máquinas podrá conseguir para ellos tiempos aceptables. El factor predominante que delimita lo que es soluble en un tiempo razonable de lo que no lo es, es precisamente el **algoritmo** elegido para resolver el problema.

⋆ En este capítulo enseñaremos a medir la **eficiencia** de los algoritmos y a comparar la eficiencia de distintos algoritmos para un mismo problema. Después de la **corrección**, conseguir eficiencia debe ser el principal objetivo del programador. Mediremos principalmente la eficiencia en **tiempo de ejecución**, pero los mismos conceptos son aplicables a la medición de la eficiencia en **espacio**, es decir a medir la memoria que necesita el algoritmo.

⋆ El siguiente programa ordena un vector $a[0..n-1]$ por el método de selección:

[1] Ricardo Peña es el autor principal de este tema.

```
1  int a[n];
2  int i, j, pmin, temp;
3  for (i = 0; i < n-1; i++)
4   // pmin calcula la posición del mínimo de a[i..n-1]
5   {pmin = i;
6    for  (j = i+1; j < n; j++)
7      if (a[j] < a[pmin]) pmin = j;
8   // ponemos el mínimo en a[i]
9    temp = a[i]; a[i] = a[pmin]; a[pmin] = temp;
10  }
```

⋆ Una manera de medir la eficiencia en tiempo de este programa 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. Sean

$$\begin{aligned} t_a &= \text{tiempo de una asignación entre enteros} \\ t_c &= \text{tiempo de una comparación entre enteros} \\ t_i &= \text{tiempo de incrementar un entero} \\ t_v &= \text{tiempo de acceso a un elemento de un vector} \end{aligned} \tag{1.1}$$

- La línea (3) da lugar a una asignación, a $n-1$ incrementos y a n comparaciones, es decir, a un tiempo $t_a + (n-1)t_i + nt_c$.
- La línea (5) da lugar a un tiempo $(n-1)t_a$.
- El bucle interior `for` se ejecuta $n-1$ veces, cada una con un valor diferente de i. Para cada valor de i y siguiendo el cálculo hecho para la (3), la línea (6) da lugar a un tiempo $t_a + (n-i-1)t_i + (n-i)t_c$.
- La línea (7) da, para cada valor de i, un tiempo mínimo de $(n-i-1)(2t_v + t_c)$, suponiendo que la instrucción `pmin = j` nunca se ejecuta. A ello hay que sumar $(n-i-1)t_a$ en el caso más desfavorable en que dicha rama se ejecute todas las veces. El caso promedio tendrá un tiempo de ejecución entre estos dos.
- Finalmente, la línea (9) dará lugar a un tiempo $(n-1)(4t_v + 3t_a)$.
- Por tanto, el tiempo del bucle interior `for`, en el caso más desfavorable, se calcula mediante el siguiente sumatorio:

$$\sum_{i=0}^{n-2}(t_a + t_c + (n-i-1)(t_i + 2t_v + t_a + 2t_c)) = P(n-1) + \frac{1}{2}Qn(n-1)$$

siendo $P = t_a + t_c$ y $Q = t_i + 2t_v + t_a + 2t_c$.

Para no cansar al lector con tediosos cálculos, concluiremos que la suma de todos estos tiempos da lugar a dos polinomios de la forma:

$$\begin{aligned} T_{min} &= An^2 - Bn + C \\ T_{max} &= A'n^2 - B'n + C' \end{aligned}$$

donde A, A', B, B', C y C' son expresiones racionales positivas que dependen linealmente de los tiempos elementales descritos en 1.1.

⋆ En este sencillo ejemplo se observan claramente los tres factores de los que en general depende el tiempo de ejecución de un algoritmo:

1. El **tamaño** de los datos de entrada, simbolizado aquí por la longitud n del vector.
2. El **contenido** de los datos de entrada, que en el ejemplo hace que el tiempo para diferentes vectores del mismo tamaño esté comprendido entre los valores T_{min} y T_{max}.
3. El código generado por el **compilador** y el **computador** concreto utilizados, que afectan a los tiempos elementales 1.1.

⋆ Como el objetivo es poder comparar algoritmos independientemente del valor de los datos de entrada, el segundo factor podemos eliminarlo de dos maneras:

- O bien midiendo solo el **caso peor**, es decir la ejecución que tarde más tiempo de todos los ejemplares de tamaño n.
- O bien midiendo todos los casos de tamaño n y calculando el tiempo del **caso promedio**.

En esta asignatura nos concentraremos en el caso peor por dos razones:

1. El caso peor establece una cota superior fiable para **todos** los casos del mismo tamaño.
2. El caso peor es más fácil de calcular.

El caso promedio es más difícil de calcular pero a veces es más informativo. Además exige conocer la probabilidad con la que se va a presentar cada caso. Muy raramente puede ser útil conocer el **caso mejor** de un algoritmo para un tamaño n dado. Ese coste es una *cota inferior* al coste de cualquier otro ejemplar de ese tamaño.

⋆ El tercer factor impediría comparar algoritmos escritos en diferentes lenguajes, traducidos por diferentes compiladores, o ejecutados en diferentes máquinas. El criterio que seguiremos es **ignorar** estos factores.

⋆ Por tanto solo mediremos la eficiencia de un algoritmo en función del **tamaño** de los datos de entrada. Este criterio está en la base de lo que llamaremos **medida asintótica** de la eficiencia.

2. Medidas asintóticas de la eficiencia

⋆ El **criterio asintótico** para medir la eficiencia de los algoritmos tiene como objetivo comparar algoritmos **independientemente** de los lenguajes en que están escritos, de las máquinas en que se ejecutan y del valor concreto de los datos que reciben como entrada. Tan solo considera importante el **tamaño** de dichos datos. Para cada problema habrá que definir qué se entiende por tamaño del mismo.

⋆ Se basa en tres principios:

1. El coste o eficiencia es una función que solo depende del tamaño de la entrada, e.g. $f(n) = n^2$.
2. Las constantes multiplicativas o aditivas no se tienen en cuenta, e.g. $f(n) = n^2$ y $g(n) = 3n^2 + 27$ se consideran costes equivalentes.

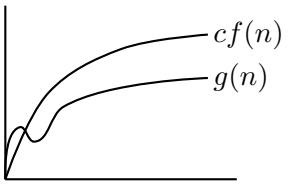
3. La comparación entre funciones de coste se hará para valores de n **suficientemente grandes**, es decir los costes para tamaños pequeños se consideran irrelevantes.

⋆ Sea $\mathbb{N}$ el conjunto de los números naturales y $\mathbb{R}^+$ el conjunto de los reales estrictamente positivos.

Definición 1.1 *Sea* $f : \mathbb{N} \longrightarrow \mathbb{R}^+ \cup \{0\}$. *El conjunto de las funciones* **del orden de** $f(n)$, *denotado* $\mathcal{O}(f(n))$, *se define como:*

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

Asímismo, diremos que una función g **es del orden de** $f(n)$ *cuando* $g \in \mathcal{O}(f(n))$. *También diremos que* g **está** *en* $\mathcal{O}(f(n))$.



⋆ Generalizando, admitiremos también que una función negativa o indefinida para un número finito de valores de n pertenece al conjunto $\mathcal{O}(f(n))$ si eligiendo n_0 suficientemente grande, satisface la definición.

⋆ Esta garantiza que, si el tiempo de ejecución $g(n)$ de una implementación concreta de un algoritmo es del orden de $f(n)$, entonces el tiempo $g'(n)$ de cualquier otra implementación del mismo que difiera de la anterior en el lenguaje, el compilador, o/y la máquina empleada, también será del orden de $f(n)$. Por tanto, el coste $\mathcal{O}(f(n))$ expresa la eficiencia del algoritmo *per se*, no el de una implementación concreta del mismo.

⋆ Las clases $\mathcal{O}(f(n))$ para diferentes funciones $f(n)$ se denominan **clases de complejidad**, u **órdenes de complejidad**. Algunos órdenes tienen nombre propio. Así, al orden de complejidad $\mathcal{O}(n)$ se le llama **lineal**, al orden $\mathcal{O}(n^2)$, **cuadrático**, el orden $\mathcal{O}(1)$ describe la clase de las funciones **constantes**, etc. Eligiremos como representante del orden $\mathcal{O}(f(n))$ la función $f(n)$ **más sencilla** posible dentro del mismo.

⋆ Nótese que la definición 1.1 se puede aplicar tanto a un análisis en el caso peor, como a un análisis en el caso promedio. Por ejemplo, hay algoritmos cuyo coste en tiempo está en $\mathcal{O}(n^2)$ en el caso peor y en $\mathcal{O}(n \log n)$ en el caso promedio.

⋆ Nótese también que las unidades en que se mide el coste en tiempo (horas, segundos, milisegundos, etc.), o en memoria (octetos, palabras, celdas de longitud fija, etc.) **no son relevantes** en la complejidad asintótica: dos unidades distintas se diferencian en una constante multiplicativa (e.g. $120\, n^2$ segundos son $2\, n^2$ minutos, ambos en $\mathcal{O}(n^2)$).

⋆ Aplicando directamente la definición de $\mathcal{O}(f(n))$, demostremos que $(n+1)^2 \in O(n^2)$. Un modo de hacerlo es por inducción sobre n. Elegimos $n_0 = 1$ y $c = 4$, es decir demostraremos $\forall n \geq 1 \,.\, (n+1)^2 \leq 4n^2$:

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

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

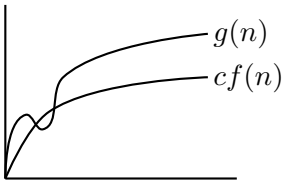
$$\begin{aligned} (n+1+1)^2 &\leq 4(n+1)^2 \\ (n+1)^2 + 1 + 2(n+1) &\leq 4n^2 + 4 + 8n \\ (n+1)^2 &\leq 4n^2 + \underbrace{6n+1}_{\geq 0} \end{aligned}$$

⋆ También podemos probar que $3^n \notin O(2^n)$. Si perteneciera, existiría $c \in \mathbb{R}^+$, $n_0 \in \mathbb{N}$ tales que $3^n \leq c \cdot 2^n$ para todo $n \geq n_0$. Esto implicaría que $(\frac{3}{2})^n \leq c$ para todo $n \geq n_0$. Pero esto es falso porque dado un c cualquiera, bastaría tomar $n > \log_{1,5} c$ para que $(\frac{3}{2})^n > c$, es decir $(\frac{3}{2})^n$ no se puede acotar superiormente.

⋆ La notación $\mathcal{O}(f(n))$ nos da una cota superior al tiempo de ejecución $t(n)$ de un algoritmo. Normalmente estaremos interesados en la **menor** función $f(n)$ tal que $t(n) \in \mathcal{O}(f(n))$. Una forma de realizar un análisis más completo es encontrar además la **mayor** función $g(n)$ que sea una cota inferior de $t(n)$. Para ello introducimos la siguiente medida.

Definición 1.2 *Sea* $f : \mathbb{N} \longrightarrow \mathbb{R}^+ \cup \{0\}$. *El conjunto* $\Omega(f(n))$, *leído* **omega de** $f(n)$, *se define como:*

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



⋆ Es frecuente confundir la medida $\mathcal{O}(f(n))$ como aplicable al caso peor y la medida $\Omega(f(n))$ como aplicable al caso mejor. Esta idea es **errónea**. Aplicaremos **ambas medidas al caso peor** (también podríamos aplicar ambas al caso promedio, o al caso mejor). Si el tiempo $t(n)$ de un algoritmo en el caso peor está en $\mathcal{O}(f(n))$ y en $\Omega(g(n))$, lo que estamos diciendo es que $t(n)$ no puede valer más que $c_1 f(n)$, ni menos que $c_2 g(n)$, para dos constantes apropiadas c_1 y c_2 y valores de n suficientemente grandes.

⋆ Es fácil demostrar (ver ejercicios) el llamado **principio de dualidad**: $g(n) \in \mathcal{O}(f(n))$ si y sólo si $f(n) \in \Omega(g(n))$.

⋆ Sucede con frecuencia que una misma función $f(n)$ es a la vez cota superior e inferior del tiempo $t(n)$ (peor, promedio, etc.) de un algoritmo. Para tratar estos casos, introducimos la siguiente medida.

Definición 1.3 *El conjunto de funciones* $\Theta(f(n))$, *leído* **del orden exacto de** $f(n)$, *se define como:*

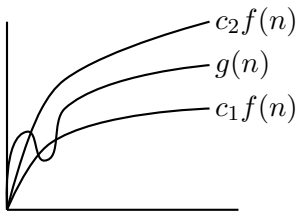
$$\Theta(f(n)) = \mathcal{O}(f(n)) \cap \Omega(f(n))$$

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

Figura 1: Crecimiento de distintas funciones de complejidad

También se puede definir como:

$$\Theta(f(n)) = \{g : \mathbb{N} \longrightarrow \mathbb{R}^+ \cup \{0\} \mid \exists c_1, c_2 \in \mathbb{R}^+, n_0 \in \mathbb{N} \,. \\ \forall n \geq n_0 \,.\, c_1 f(n) \leq g(n) \leq c_2 f(n)\}$$



⋆ Siempre que sea posible, daremos el orden exacto del coste de un algoritmo, por ser más informativo que dar solo una cota superior.

3. Jerarquía de órdenes de complejidad

⋆ Es importante visualizar las implicaciones prácticas de que el coste de un algoritmo pertenezca a una u otra clase de complejidad. La Figura 1 muestra el crecimiento de algunas de estas funciones, suponiendo que expresan un tiempo en milisegundos (ms = milisegundos, s = segundos, m = minutos, h = horas, etc.).

⋆ Se aprecia inmediatamente la **extraordinaria eficiencia** de los algoritmos de coste en $\mathcal{O}(\log n)$: pasar de un tamaño de $n = 10$ a $n = 1\,000\,000$ solo hace que el tiempo crezca de 1 milisegundo a 6. La búsqueda binaria en un vector ordenado, y la búsqueda en ciertas estructuras de datos de este curso, tienen este coste en el caso peor.

⋆ En sentido contrario, los algoritmos de coste $\mathcal{O}(2^n)$ son **prácticamente inútiles**: mientras que un problema de tamaño $n = 10$ se resuelve en aproximadamente un segundo, la edad del universo conocido ($1,4 \times 10^8$ siglos) sería totalmente insuficiente para resolver uno de tamaño $n = 100$. Algunos algoritmos de *vuelta atrás* que veremos en este curso tienen ese coste en el caso peor.

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

Figura 2: Jerarquía de órdenes de complejidad

⋆ Esta tabla confirma la afirmación hecha al comienzo de este capítulo de que para ciertos algoritmos es inútil esperar a que los computadores sean más rápidos. Es más productivo invertir esfuerzo en diseñar **mejores algoritmos** para ese problema.

⋆ Para mejorar la intuición anterior, hagamos el siguiente experimento: supongamos seis algorimos con los costes anteriores, tales que tardan todos ellos 1 hora en resolver un problema de tamaño $n = 100$. ¿Qué ocurre si duplicamos la velocidad del computador? O lo que es lo mismo, ¿qué ocurre si duplicamos el tiempo disponible?

$t(n)$	$t = 1h.$	$t = 2h.$
$k_1 \cdot \log n$	$n = 100$	$n = 10\,000$
$k_2 \cdot n$	$n = 100$	$n = 200$
$k_3 \cdot n \log n$	$n = 100$	$n = 178$
$k_4 \cdot n^2$	$n = 100$	$n = 141$
$k_5 \cdot n^3$	$n = 100$	$n = 126$
$k_6 \cdot 2^n$	$n = 100$	$n = 101$

⋆ Observamos que mientras el de coste logarítmico es capaz de resolver problemas 100 veces más grandes, el de coste exponencial resuelve un tamaño practicamente igual al anterior. Obsérvese que los de coste $\mathcal{O}(n)$ y $\mathcal{O}(n \log n)$ se comportan de acuerdo a la intuición de un usuario no informático: al duplicar la velocidad del computador (o el tiempo disponible), se duplica aproximadamente el tamaño del problema resuelto. En los de coste $\mathcal{O}(n^k)$, al duplicar la velocidad, el tamaño se multiplica por un factor $\sqrt[k]{2}$.

⋆ En la Figura 2 se muestra la **jerarquía de órdenes de complejidad**. Las inclusiones estrictas expresan que se trata de clases distintas. Los algoritmos cuyos costes están en la parte izquierda resuelven problemas que se denominan **tratables**. Estos costes se denominan en su conjunto **polinomiales**. Hay problemas que sólo admiten algoritmos de complejidad exponencial o superior. Se llaman **intratables**.

⋆ También hay muchos problemas interesantes cuyos mejores algoritmos conocidos son exponenciales en el caso peor, pero no se sabe si existirán para ellos algoritmos polinomiales. Se llaman **NP-completos** y se verán en el próximo curso. El más conocido de todos es el problema **SAT** que consiste en determinar si una fórmula de la lógica proposicional es satisfactible.

4. Propiedades de los órdenes de complejidad

⋆ $O(a \cdot f(n)) = O(f(n))$ 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 $a, b > 1$. La demostración es inmediata sabiendo que:

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

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

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

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 $f \in O(h)$.

⋆ Regla de la suma: $O(f + g) = O(\text{máx}(f, g))$.

($\subseteq$) $h \in O(f + g) \Rightarrow \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N} \,.\, \forall n \geq n_0 \,.\, h(n) \leq c \cdot (f(n) + g(n))$. Pero $f \leq \text{máx}(f, g)$ y $g \leq \text{máx}(f, g)$, luego:

$$h(n) \leq c \cdot (\text{máx}(f(n), g(n)) + \text{máx}(f(n), g(n))) = 2 \cdot c \cdot \text{máx}(f(n), g(n))$$

Tomando $c' = 2 \cdot c$ se cumple que $\forall n \geq n_0 \,.\, h(n) \leq c' \cdot \text{máx}(f(n), g(n))$ y por tanto $h \in O(\text{máx}(f, g))$.

($\supseteq$) $h \in O(\text{máx}(f, g)) \Rightarrow \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N} \,.\, \forall n \geq n_0 \,.\, h(n) \leq c \cdot \text{máx}(f(n), g(n))$. Pero $\text{máx}(f, g) \leq f + g$, 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 $g_1 \cdot g_2 \in O(f_1 \cdot f_2)$. La demostración es similar.

⋆ Teorema del límite

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

⋆ Por el principio de dualidad, también tenemos:

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

⋆ Aplicando la definición de $\Theta(f)$, también tenemos:

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

Notas bibliográficas

Se recomienda ampliar el contenido de estas notas estudiando el Capítulo 1 de (Peña, 2005) en el cual se han basado. También la Sección 1.4 de (Rodriguez Artalejo et al., 2011).

El Capítulo 3 de (Martí Oliet et al., 2012) contiene material válido para este capítulo y material adicional que será utilizado en los Capítulos 3 y 4. También tiene numerosos ejemplos resueltos.

Ejercicios

1. Demostrar el Principio de dualidad, es decir $g(n) \in \mathcal{O}(f(n)) \Leftrightarrow f(n) \in \Omega(g(n))$.

2. Demostrar que todo polinomio $a_m n^m + \cdots + a_1 n + a_0$, en n y de grado m, cuyo coeficiente a_m correspondiente al mayor grado sea positivo, está en $\mathcal{O}(n^m)$.

3. Demostrar
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow \mathcal{O}(f(n)) \subset \mathcal{O}(g(n))$$
Dar un ejemplo de que la implicación inversa puede no ser cierta.

4. Demostrar
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = k > 0 \Rightarrow f(n) \in \Theta(g(n))$$

5. Usar el teorema del límite para demostrar las siguientes inclusiones estrictas (suponemos $k > 1$):
$$O(1) \subset O(\log n) \subset O(n) \subset O(n^k) \subset O(2^n) \subset O(n!)$$

6. Si tenemos dos algoritmos con costes $t_1(n) = 3n^3$ y $t_2(n) = 600n^2$, ¿cuál es mejor en términos asintóticos? ¿A partir de que umbral el segundo es mejor que el primero?

7. Si el coste de un algoritmo está en $\mathcal{O}(n^2)$ y tarda 1 segundo para un tamaño $n = 100$, ¿de qué tamaño será el problema que puede resolver en 10 segundos?

8. Demostrar por inducción sobre $n \geq 0$ las siguientes igualdades:

   a) $\sum_{i=1}^{n} i = n(n+1)/2$.
   b) $\sum_{i=1}^{n} i^2 = n(n+1)(2n+1)/6$.
   c) $\sum_{i=1}^{n} 2^i i = (n-1)2^{n+1} + 2$.

9. Demostrar que $\sum_{i=1}^{n} i^k \in \Theta(n^{k+1})$.

10. Demostrar que $\log n \in O(\sqrt{(}n))$ pero que $\sqrt{(}n) \notin O(\log n)$.

11. ¿Verdadero o falso?

   a) $2^n + n^{99} \in O(n^{99})$.
   b) $2^n + n^{99} \in \Omega(n^{99})$.
   c) $2^n + n^{99} \in \Theta(n^{99})$.
   d) Si $f(n) = n^2$, entonces $f(n)^3 \in O(n^5)$.
   e) Si $f(n) \in O(n^2)$ y $g(n) \in O(n)$, entonces $f(n)/g(n) \in O(n)$.
   f) Si $f(n) = n^2$, entonces $3f(n) + 2n \in \Theta(f(n))$.
   g) Si $f(n) = n^2$ y $g(n) = n^3$, entonces $f(n)g(n) \in O(n^6)$.

12. Comparar con respecto a O y Ω los siguientes pares de funciones:

   a) $2^{n+1},\ 2^n$.
   b) $(n+1)!,\ n!$.
   c) $\log n,\ \sqrt{n}$.
   d) Para cualquier $a \in \mathbb{R}^+$, $\log n, n^a$.

13. Supongamos que $t_1(n) \in \mathcal{O}(f(n))$ y $t_2(n) \in \mathcal{O}(f(n))$. Razonar la verdad o falsedad de las siguientes afirmaciones:

   a) $t_1(n) + t_2(n) \in \mathcal{O}(f(n))$.
   b) $t_1(n) \cdot t_2(n) \in \mathcal{O}(f(n^2))$.
   c) $t_1(n)/t_2(n) \in \mathcal{O}(1)$.

14. (Martí Oliet et al. (2012)) Demostrar o refutar cada una de las siguientes afirmaciones:

   a) $n^2 + n + \log n \in O(n^2)$.
   b) $n^2 + n + \log n \in \Omega(n)$.
   c) $n^2 + n + \log n \in \Theta(\log n)$.
   d) $21n^3 + 7n^2 + n\log n - 17n + 8 \in O(n^4)$.
   e) $2^n + 3^n + n^{59} \in O(n^{59})$.
   f) $2^n + 3^n + n^{59} \in \Omega(2^n)$.
   g) Si $f(n) = n^2$, entonces $f(n)^3 \in \Theta(f(n)^3)$.
   h) $\Theta(2^n) = \Theta(2^{n+2}) = \Theta(4^n)$.
   i) $\log_2 n \in O(\log_3 n)$.

15. (Martí Oliet et al. (2012)) Para cada una de las siguientes funciones $f(n)$, obtener el menor número natural k tal que $f(n) \in O(n^k)$.

a) $f(n) = 2n^3 + n^2 \log n$.

b) $f(n) = 3n^5 + (\log n)^4$.

c) $f(n) = (n^4 + n^2 + 1)/(n^3 + 1)$.

d) $f(n) = (n^4 + n^2 + 1)/(n^4 + 1)$.

e) $f(n) = (n^4 + 5\log n)/(n^4 + 1)$.

f) $f(n) = (n^3 + 5\log n)/(n^4 + 1)$.

16. (Martí Oliet et al. (2012)) Para cada una de las siguientes funciones $f(n)$, encontrar una función $g(n)$ del menor orden posible tal que $f(n) \in O(g(n))$.

a) $f(n) = (n \log n + n^2)(n^3 + 2)$.

b) $f(n) = (2^n + n^2)(n^3 + 3^n)$.

c) $f(n) = n \log(n^2 + 1) + n^2 \log n$.

d) $f(n) = n^{2^n} + n^{n^2}$.

e) $f(n) = (n! + 2^n)(n^3 + \log(n^2 + 1))$.

17. (Martí Oliet et al. (2012)) Comparar con respecto a O y Ω los siguientes pares de funciones:

a) $n^2 + 3n + 7, n^2 + 10$.

b) $n^2 \log n, n^3$.

c) $n^4 + \log(3n^8 + 7), (n^2 + 17n + 3)^2$.

d) $(n^3 + n^2 + n + 1)^4, (n^4 + n^3 + n^2 + n + 1)^3$.

e) $\log(n^2 + 1), \log n$.

f) $2^{n+3}, 2^{n+7}$.

g) $2^{2^n}, 2^{n^2}$.