Análisis de la eficiencia de los algoritmos

Obtenido del Tema 1 de los apuntes de la asignatura de EDA.
Ricardo Peña es el autor principal de este tema

Facultad de Informática - UCM

28 de septiembre de 2013

Biliografía Recomendada

- Diseño de Programas: Formalismo y Abstracción. *Ricardo Peña*. Tercera edición, Pearson Prentice-Hall, 2005
- Algoritmos correctos y eficientes: Diseño razonado ilustrado con ejercicios. *Matí-Oliet, N.; Segura Diaz, C. M., Verdejo Lopez, A.*. Ibergarceta Publicaciones, 2012.
- Especificación, Derivación y Análisis de Algoritmos: ejercicios resueltos. *Narciso Martí Oliet, Clara María Segura Díaz y Jose Alberto Verdejo López*. Colección Prentice Práctica, Pearson Prentice-Hall, 2006

Introducción

OBJETIVOS DEL TEMA:

1. *medir la* **eficiencia** *en tiempo/espacio de los algoritmos.*
2. *comparar la eficiencia de distintos algoritmos para un mismo problema.*

PROBLEMA: Calcular el n-ésimo término de la sucesión de Fibonacci.

$$f(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ f(n-1) + f(n-2) & \text{if } n > 1 \end{cases}$$

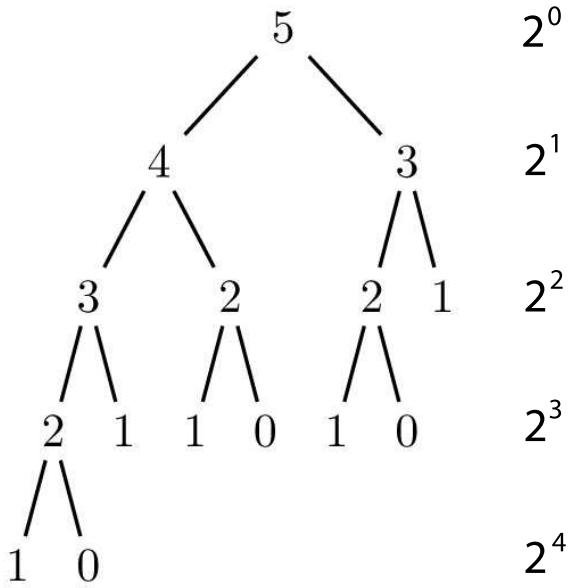
1º SOLUCIÓN: Algoritmo recursivo según la definición matemática

```
1 unsigned long FibRecInefic (unsigned int n)
2 {
3   if (n == 0 || n == 1) return n;
4   else
5     return FibRecInefic(n-1) + FibRecInefic(n-2);
6 }
```

n	Nº llamadas recursivas	n	Nº llamadas recursivas
5	7	30	1.346.268
10	88	40	165.580.140
20	10.945		

Porque el algoritmo no es eficiente?

Árbol de activación



Número de llamadas recursivas:

$$min = \sum_{i=1}^{h \text{ arbol completo}} 2^i = 2^{h_1+1} - 2$$

$$max = \sum_{i=1}^{h \text{ total arbol}} 2^i = 2^{h_2+1} - 2$$

$$S_n = \frac{a_n r - a_1}{r-1}$$

2º SOLUCIÓN: Algoritmo recursivo de complejidad lineal

```
1  unsigned long FibRecEfic2 (unsigned int n,
2       unsigned long p0, unsigned long p1)
3  {
4     if (n == 1) return p1;
5     else return FibRecEfic2(n-1, p1, p0 + p1);
6  }
7  unsigned long FibRecEfic (unsigned int n)
8  {
9     if (n == 0) return 0;
10    else return FibRecEfic2(n,0,1);
11 }
```

memoize, o guardar las variables en el argumento

n	Nº llamadas recursivas	n	Nº llamadas recursivas
5	4	30	29
10	9	40	39
20	19		

3º SOLUCIÓN: Algoritmo recursivo de complejidad logarítmica

Ejercicio 5 del tema 5 de Divide y Vencerás

n	Nº llamadas recursivas	n	Nº llamadas recursivas
10	3,33	1.000	10
20	4,34	1.000.000	20
40	5,34		

Introducción

PROBLEMA: Decidir si un determinado elemento pertenece a un vector ordenado de forma creciente.

SOLUCIÓN 1: Recorrer el vector de izquierda a derecha. Ineficiente.

```
1 bool IneficSearch (int A[], int n, int x)
2 {
3     int i = 0;
4     while (i < n && A[i] != x) i++;
5     return i < n;
6 }
```

SOLUCIÓN 2: Búsqueda binaria. Eficiente.

```
1  bool BinarySearchIt (int A[], int n, int x)
2  {
3      int a = 0; int b = n - 1; bool enc = false;
4      while (a <= b && !enc) {
5          int m = (a + b) / 2;
6          if (A[m] == x) enc = true;
7          else if (A[m] > x) b = m - 1;
8          else a = m + 1;
9      }
10     return enc;
11 }
```

COSTE DE LAS DOS SOLUCIONES:

Estudio: Vector con números pares consecutivos empezando en 2. Se buscan números impares consecutivos empezando en el 3. Se cuenta el número de vueltas que da el bucle del algoritmo.

Tam. vector	Nº Búsquedas	Solución 1	Solución 2
100.000	10	1.000.000	167
100.000	10.000	1.000.000.000	166.891
500.000	10	5.000.000	190
500.000	10.000	5.000.000.000	189.514

Solución 1: Coste lineal respecto al tamaño del vector $\mathcal{O}(n)$
Solución 2: Coste logarítmico respecto al tamaño del vector $\mathcal{O}(\log n)$

Introducción

La tabla muestra el tiempo que tardarían en ejecutarse algoritmos con las funciones de complejidad y el tamaño de datos que se indica, suponiendo que el tiempo de proceso para un dato de entrada es de *1ms*.

n	$\log_{10} n$	n	n^2	n^3	2^n
10	$1\ ms$	$10\ ms$	$0{,}1\ s$	$1\ s$	$1{,}02\ s$
10^3	$3\ ms$	$1\ s$	$16{,}67\ m$	$11{,}57\ d$	$3{,}4 * 10^{291}\ sig$
10^6	$6\ ms$	$1{,}67\ h$	$31{,}71\ a$	$317\,097{,}9\ sig$	$3{,}1 * 10^{301020}\ sig$

Hay algoritmos tan ineficientes que ningún avance en la velocidad de las máquinas podrá conseguir para ellos tiempos aceptables

Para calcular la eficiencia en tiempo de un programa iterativo: **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.

Programa que comprueba si un valor pertenece a un vector recorriéndolo de izquierda a derecha.

```
1 bool IneficSearch (int A[], int n, int x)
2 {
3     int i = 0;
4     while (i < n && A[i] != x)
5         i++;
6     return i < n;
7 }
```

Introducción

Sean

$$\begin{aligned} t_a &= \text{tiempo de una asignación entre enteros} \quad (1)\\ t_c &= \text{tiempo de una comparación entre enteros}\\ t_i &= \text{tiempo de incrementar un entero}\\ t_v &= \text{tiempo de acceso a un elemento de un vector} \end{aligned}$$

- Coste línea (3) : t_a.
- Coste líneas (4) y (5):
  - Si el elemento no pertenece al vector (caso peor):
    línea (4) $nt_c + (n-1)(t_v + t_a)$.
    línea (5) $(n-1)t_i$.
  - Si el elemento está en la primera posición del vector (caso mejor):
    línea (4) $t_c + t_v + t_a$.
    línea (5) No se ejecuta.
- Coste línea (6) : t_c.

En el caso peor el coste del algoritmo es:

$$t_a + nt_c + (n-1)(t_v + t_a) + (n-1)t_i + t_c = (n+1)t_c + nt_a + (n-1)(t_v + t_i) \in \mathcal{O}(n)$$

siendo n el número de elementos del vector.

En el caso mejor el coste del algoritmo es:

$$t_a + t_c + t_v + t_a + t_c = 2t_c + 2t_a + t_v \in \mathcal{O}(1)$$

EJERCICIO: Calcular el coste del algoritmo de búsqueda binaria

```
1  bool BinarySearchIt (int A[], int n, int x)
2  {
3      int a = 0; int b = n - 1; bool enc = false;
4      while (a <= b && !enc) {
5          int m = (a + b) / 2;
6          if (A[m] == x) enc = true;
7          else if (A[m] > x) b = m - 1;
8          else a = m + 1;
9      }
10     return enc;
11 }
```

Nota: El número de veces que se ejecuta la instrucción `while` en el caso peor puede aproximarse a $\lceil \log_2 n \rceil$

Introducción

Otro ejemplo: Programa que ordena un vector $a[0..n-1]$ por el método de selección:

```
1  int a[n];
2  int i, j, pmin, temp;
3  for (i = 0; i < n-1; i++)
4   // pmin calcula la posicion del minimo 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 minimo en a[i]
9     temp = a[i]; a[i] = a[pmin]; a[pmin] = temp;
10  }
```

Introducción

- Coste línea (3) : $t_a + (n-1)t_i + nt_c$.
- Coste línea (5): $(n-1)t_a$.
- El bucle interior **for** se ejecuta $n-1$ veces, cada una con un valor diferente de i. Para cada valor de i:
- Coste línea (6): $t_a + (n-i)t_i + (n-i+1)t_c$.
- Coste línea (7)
  - Mínimo (la instrucción nunca se ejecuta): $(n-i)(2t_v + t_c)$.
  - Máximo: $(n-i)(2t_v + t_c) + (n-i)t_a$.
- Coste línea (9): $(n-1)(4t_v + 3t_a)$.

- El tiempo del bucle interior **for**, en el caso más desfavorable, se calcula mediante el sumatorio:

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

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

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).

Introducción

Factores de los que depende el tiempo de ejecución:

1. El **tamaño** de los datos de entrada:
   - longitud del vector (algoritmos de ordenación de vectores...),
   - valor del dato de entrada (sucesión de fibonacci, cálculo de la potencia de un número...) ,
   - número de cifras del dato de entrada (cambio de base binaria, decimal ....)
   - ...
2. El **contenido** de los datos de entrada. Fijado un tamaño de los datos de entrada, dependiendo del valor de los datos el tiempo es diferente. Por ejemplo, para vectores de un mismo tamaño dependiendo del valor de las componentes del vector obtenemos un tiempo u otro (T_{min} y T_{max}).
3. El código generado por el **compilador** y el **computador** concreto utilizados, que afectan a los tiempos elementales $(t_a, t_c, t_i \ldots)$.

Introducción

Abstracciones que se realizan en el cálculo de la complejidad:

La comparación de los algoritmos es independiente del valor de los datos de entrada. Para ello se puede:

1. Medir sólo el **caso peor**, es decir la ejecución que tarde más tiempo de todos los ejemplares de tamaño n.
   - Establece una cota superior fiable para **todos** los casos del mismo tamaño.
   - Es fácil de calcular
2. Medir todos los casos de tamaño n y calcular el tiempo del **caso promedio**.
   - Es más difícil de calcular pero a veces es más informativo.
   - Exige conocer la probabilidad con la que se va a presentar cada caso. Si no se conoce se considera uniforme.
3. Medir el **caso mejor**. Es una *cota inferior* al coste de cualquier otro ejemplar de ese tamaño. Muy raramente resulta útil.

- Consideramos todos los tiempos elementales iguales, independientemente de la máquina o el compilador utilizado.

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.

Medidas asintóticas de la eficiencia

El criterio asintótico para medir la eficiencia de los algoritmos se basa en tres principios:

1. El coste o eficiencia es una función que solo depende del tamaño de la entrada, e.g. $f(n) = n^2$. Para cada problema habrá que definir qué se entiende por tamaño del mismo.
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.

Medidas asintóticas de la eficiencia

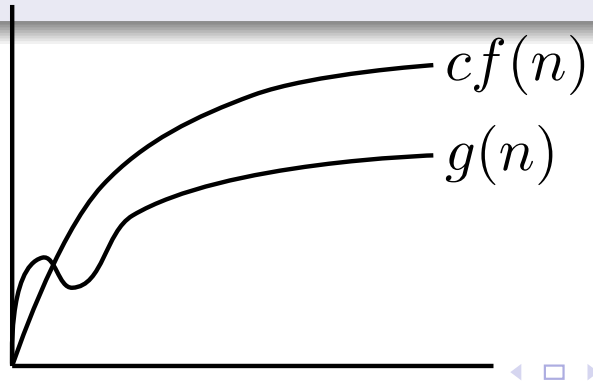
Sea $\mathbb{N}$ el conjunto de los números naturales y $\mathbb{R}^+$ el conjunto de los reales estrictamente positivos.

Definición 0.1

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* $\mathcal{O}(f(n))$.



- Se admiten funciones negativas o indefinidas para un número finito de valores de n si eligiendo n_0 suficientemente grande, satisface la definición.
- Diferentes implementaciones del mismo algoritmo que difieran en el lenguaje, el compilador, o/y la máquina empleada, son del mismo orden.
- La definición 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 (Quicksort).
- 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.

Medidas asintóticas de la eficiencia

Las clases $\mathcal{O}(f(n))$ para diferentes funciones $f(n)$ se denominan **clases de complejidad**, u **órdenes de complejidad**.

Se elije como representante del orden $\mathcal{O}(f(n))$ la función $f(n)$ **más sencilla** posible dentro del mismo.

Esta función da nombre al orden:

1. $\mathcal{O}(1)$: **constantes**
2. $\mathcal{O}(\log n)$: **logarítmico**
3. $\mathcal{O}(n)$: **lineal**
4. $\mathcal{O}(n^2)$: **cuadrático**
5. $\mathcal{O}(n^k)$: **polinomial**
6. $\mathcal{O}(2^n)$: **exponencial**
7. $\mathcal{O}(2!)$: **factorial**

Medidas asintóticas de la eficiencia

Para demostrar que una función pertenece a un orden se aplica directamente la definición.

- Demostrar 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 $n+1$:

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

Medidas asintóticas de la eficiencia

Para demostrar que una función no pertenece a un orden se hace por reducción al absurdo.

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 $t(n) \in \mathcal{O}(f(n))$.

Jerarquía de órdenes de complejidad

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

$$\underbrace{\ldots \subset O(2^n) \subset O(n!)}_{\text{intratables}}$$

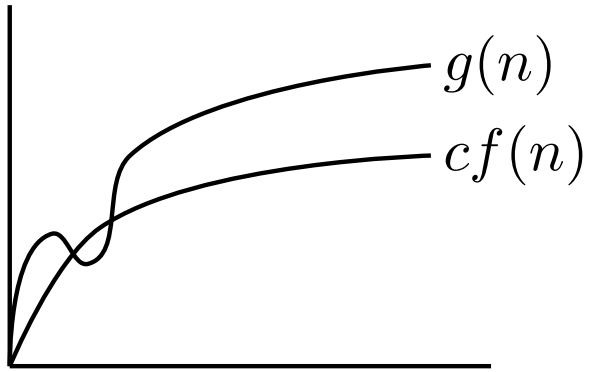
Medidas asintóticas de la eficiencia

Una forma de realizar un análisis más completo es encontrar además la **mayor** función $g(n)$ que sea una cota inferior de $t(n)$.

Definición 0.2

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

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



- NO confundir la medida $\mathcal{O}(f(n))$ como aplicable al caso peor y la medida $\Omega(f(n))$ como aplicable al caso mejor.
  Se aplican **ambas medidas a cada caso** (caso peor, caso promedio, o 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.
- 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.

Medidas asintóticas de la eficiencia

Para tratar los casos en que la cota superior e inferior del tiempo de un algoritmo es la misma función se define la siguiente medida.

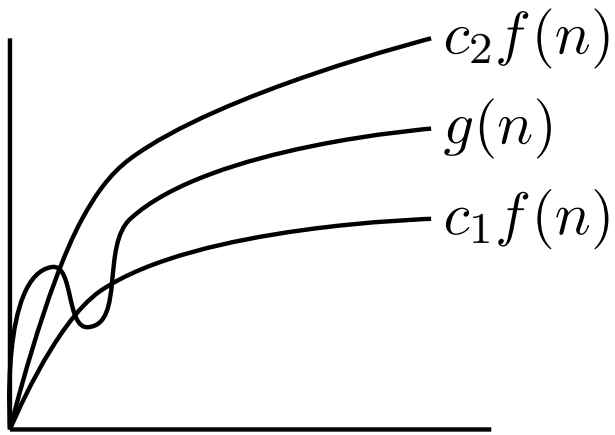
Definición 0.3

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

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

- 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.

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 $f \in O(h)$.

$f \in O(g) \Rightarrow \exists c_1 \in \mathbb{R}^+, n_1 \in \mathbb{N}$ 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 $f \in O(h)$.

- 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:

  $$h(n) \leq c \cdot (\max(f(n),g(n)) + \max(f(n),g(n)))$$

  $$= 2 \cdot c \cdot \max(f(n),g(n))$$

  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 $g_1 \cdot g_2 \in O(f_1 \cdot f_2)$. 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 (ejercicio 1), 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:

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

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f \in O(g) \text{ pero } f \notin \Theta(g)$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty \Rightarrow g \in O(f) \text{ pero } g \notin \Theta(f)$$

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 $k > 1$):

$$O(1) \subset O(\log n) \subset O(n^k) \subset O(n^k) \subset O(2^n) \subset O(n!)$$

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:
   1. $\sum_{i=1}^{n} i = n(n+1)/2$.
   2. $\sum_{i=1}^{n} i^2 = n(n+1)(2n+1)/6$.
   3. $\sum_{i=1}^{n} 2^i i = (n-1)2^{n+1} + 2$.
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?
    1. $2^n + n^{99} \in O(n^{99})$.
    2. $2^n + n^{99} \in \Omega(n^{99})$.
    3. $2^n + n^{99} \in \Theta(n^{99})$.
    4. Si $f(n) = n^2$, entonces $f(n)^3 \in O(n^5)$.
    5. Si $f(n) \in O(n^2)$ y $g(n) \in O(n)$, entonces $f(n)/g(n) \in O(n)$.
    6. Si $f(n) = n^2$, entonces $3f(n) + 2n \in \Theta(f(n))$.
    7. 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:
    1. $2^{n+1},\ 2^n$.
    2. $(n+1)!,\ n!$.
    3. $\log n,\ \sqrt{n}$.
    4. 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:
    1. $t_1(n) + t_2(n) \in \mathcal{O}(f(n))$.
    2. $t_1(n) \cdot t_2(n) \in \mathcal{O}(f(n^2))$.
    3. $t_1(n)/t_2(n) \in \mathcal{O}(1)$.