Análisis de Algoritmos

Natalia Baeza - Nadina Martínez Carod

1 de septiembre de 2023

Índice general

Índice de cuadros

Índice de figuras

Capítulo 1

Eficiencia y Notación Asintótica

1.1. Conceptos Generales

1.1.1. Introducción

Un algoritmo se puede definir como una secuencia de instrucciones que representan un modelo de solución para determinado tipo de problemas. Por lo tanto podemos decir que es un conjunto ordenado y finito de pasos que nos permite solucionar un problema.

Los algoritmos son independientes de los lenguajes de programación. El algoritmo es la infraestructura de cualquier solución, escrita luego en cualquier lenguaje de programación.

Un algoritmo es una secuencia bien determinada de acciones elementales que transforman los datos de entrada en datos de salida con el objetivo de resolver un problema computacional. El problema computacional consiste de una especificación de un conjunto de datos de entrada junto con una especificación de un conjunto de datos de salida en base a la entrada. La especificación de un algoritmo por medio de un lenguaje de programación se realiza mediante un programa de programación, de modo tal que se pueda ejecutar en una computadora.

1.1.1.1. Solución de un algoritmo

La solución al algoritmo se puede presentar como un modelo matemático que le da el soporte para demostrar la correctitud y eficiencia de la solución alcanzada a partir de propiedades y demostraciones matemáticas. La solución a un problema dado puede consistir en diversos algoritmos basados en ideas muy diversas, los cuales pueden resolver el problema con velocidades de diferencias abismales en algunos casos.

Los datos de la entrada a un algoritmo especifican una instancia del problema a resolver, por lo tanto es muy importante especificar exactamente el conjunto de casos, dominio, que el algoritmo debe contemplar.

En el proceso de elegir la solución se distinguen entre soluciones exactas y soluciones aproximadas. Existen problemas importantes que simplemente no pueden ser resueltos con exactitud

para la mayoría de sus casos; por ejemplo los problemas de extracción de raíces cuadradas, la resolución de ecuaciones no lineales, integrales, el cálculo de logaritmos, etc.

También hay problemas complejos donde el tiempo requerido para su resolución se considera inaceptable o el algoritmo una una parte de otro algoritmo más sofisticado que resuelve un problema exactamente.

1.1.1.2. Análisis de un algoritmo

Cuando se resuelve un problema, con frecuencia se necesita elegir entre varios algoritmos. ¿Cómo se debe elegir? Hay dos enfoques que suelen colisionar:

1. Que el algoritmo sea fácil de entender, codificar y depurar.

2. Que el algoritmo use eficientemente los recursos de la computadora y, en especial, que se ejecute con la mayor rapidez posible.

Cuando se escribe un programa cuya ejecución no es habitual pero su complejidad es alta, conviene el primer enfoque. Por ejemplo, resolver estrategias de juego o simulaciones de procesos. En este caso es probable que el costo del tiempo de programación, es decir resolver el problema, exceda en mucho al costo de ejecución del programa, de modo que el costo a optimizar es el de la escritura.

En cambio, cuando se resuelve un problema cuya solución va a utilizarse muchas veces, el costo de ejecución del programa puede superar en mucho al de escritura, en especial para ejecuciones que tienen entradas de gran tamaño. Por ejemplo, realizar búsquedas de claves, reconocimiento de patrones, ordenación de elementos. Entonces es más ventajoso desarrollar un algoritmo más complejo buscando conseguir que el tiempo de ejecución del programa resultante sea mucho mejor. Y aún en situaciones como ésa, quizá sea conveniente construir primero un algoritmo simple, con el objeto de determinar el beneficio real que se obtendría con un programa más complicado.

En la construcción de software se emplean prototipos, y en general se busca implementar un prototipo sencillo sobre el que se puedan ejecutar simulaciones y mediciones antes de llegar al diseño definitivo. Por eso un/a programador/a no solo debe conocer sobre eficiencia de un programa, sino que también debe saber cuándo aplicar esas técnicas y cuándo no.

1.1.1.3. Eficiencia

Para calcular el tiempo de ejecución siempre partimos de la base que el algoritmo es correcto, es decir, que produce el resultado deseado en un tiempo nito. Para comparar dos algoritmos correctos que resuelven el mismo problema, se determina su eficiencia de acuerdo a una medida de tamaño N, que representa la cantidad de datos que manipulan dichos algoritmos. Respecto al uso eficiente de los recursos, éste suele medirse en función de dos parámetros:

- *espacio*: memoria que utiliza

- *tiempo*: lo que tarda en ejecutarse

Ambos representan los costes que supone encontrar la solución al problema planteado mediante un algoritmo. Dichos parámetros van a servir además para comparar algoritmos entre sí, permitiendo determinar el más adecuado de entre varios que solucionan un mismo problema. En este capítulo nos centraremos solamente en la eficiencia temporal. El tiempo de ejecución de un algoritmo va a depender de diversos factores como son: los datos de entrada que le suministremos, la calidad del código generado por el compilador para crear el programa objeto, la naturaleza y rapidez de las instrucciones máquina del procesador concreto que ejecute el programa, y la complejidad intrínseca del algoritmo.

Hay dos estudios posibles sobre el tiempo:

1. Uno que proporciona una medida *teórica* (a priori), que consiste en obtener una función que acote (por arriba o por abajo) el tiempo de ejecución del algoritmo para unos valores de entrada dados.

2. Y otro que ofrece una medida *real o empírico* (a posteriori), consistente en medir el tiempo de ejecución del algoritmo para unos valores de entrada dados y en un ordenador concreto.

Ambas medidas son importantes puesto que, si bien la primera nos ofrece estimaciones del comportamiento de los algoritmos de forma independiente del ordenador en donde serán implementados y sin necesidad de ejecutarlos, la segunda representa las medidas reales del comportamiento del algoritmo. Estas medidas son *funciones temporales* de los datos de entrada.

Entonces podemos resumir que para comparar los algoritmos se puede utilizar una estrategia empírica, teórica, o una mezcla de ambas:

El enfoque empírico consiste en programar los algoritmos y ejecutarlos en una computadora sobre varios casos de prueba, y comparar los tiempos reales de ejecución.

El enfoque teórico consiste en determinar matemáticamente la cantidad de recursos (tiempo, espacio, etc) que necesitará el algoritmo en función de la cantidad de datos manipulados por el algoritmo. Esta estrategia tiene la ventaja de que no depende de la computadora ni del lenguaje de programación utilizado. Además, evita el esfuerzo de programar algoritmos ineficientes y de desperdiciar tiempo de máquina para ejecutarlos. También permite conocer la eficiencia de un algoritmo para cualquier tamaño de entrada.

Comenzaremos con el enfoque teórico. Para ello, el tiempo de ejecución de un algoritmo se expresará como una función de la longitud de entrada en relación con un número de pasos a realizar.

Entendemos por *tamaño de la entrada* el número de componentes sobre los que se va a ejecutar el algoritmo. Por ejemplo, la dimensión del vector a ordenar o el tamaño de las matrices a multiplicar. La unidad de tiempo a la que debe hacer referencia estas medidas de eficiencia no puede ser expresada en segundos o en otra unidad de tiempo concreta, pues no existe un ordenador estándar al que puedan hacer referencia todas las medidas. Así, el tiempo de ejecución de un programa será una función de una cantidad n de elementos, que se la denomina $T(n)$.

En las sentencias condicionales (alternativas y repetitivas de tipo mientras o hasta) los algoritmos dependen de valores concretos, y generalmente se pueden diferenciar el mejor caso, el peor caso, el caso promedio y otro menos utilizados que son el probabilístico y el de análisis amortizado.

Mejor caso T_{mejor} ***(n)***:

Es el menor de los tiempos de ejecución sobre todas las entradas de tamaño n. Puede dar lugar a errores en algoritmos lentos que trabajan rápido sobre unas pocas entradas. Por ejemplo, en el método de ordenamiento burbuja mejorado, el mejor caso es cuando el arreglo ya está ordenado y en una sola pasada termina sin hacer ningún intercambio.

Caso Promedio $T_{promedio}$ ***(n)***:

Es el promedio de los tiempos de ejecución sobre todas las entradas de tamaño n. Es el más fiel, pero puede ser muy difícil de determinar. En el ejemplo del método burbuja mejorado, el caso promedio es cuando el arreglo está aleatoriamente desordenado.

Peor caso T_{peor} ***(n)***:

Es el máximo de los tiempos de ejecución sobre todas las entradas de tamaño n, puede no ser muy el, pero consideramos que es una cota superior del tiempo promedio. En el mismo ejemplo del método burbuja mejorado, el peor caso es cuando el arreglo está ordenado de manera inversa a la deseada. En este caso se produce la mayor cantidad de comparaciones e intercambios.

El ***análisis probabilístico*** asume una distribución de probabilidad sobre las posibles entradas.

El ***análisis amortizado*** utiliza el tiempo promedio de ejecución sobre una secuencia de ejecuciones sucesivas.

En algunos algoritmos se utiliza el tiempo promedio, principalmente cuando para fines prácticos la probabilidad de que ocurra el peor caso es muy poca, o cuando el tiempo de ejecución es prácticamente el mismo que el del peor caso (y si se tarda más, no nos afecta mucho).

Como el análisis debe realizarse con independencia de los valores que determinan las sentencias condicionales, en general se prefiere el cálculo más pesimista, es decir el peor caso. Para ello se evalúa la estructura del algoritmo bajo la suposición de una memoria infinita (es decir que no habrá demoras por intercambio de información entre memoria primaria y secundaria).

el análisis amortizado estudia el tiempo requerido para ejecutar una secuencia de operaciones sobre una estructura de datos I si usamos el análisis normal en el peor caso, ejecutar N operaciones sobre una estructura de datos de n elementos lleva tiempo en O(Nf(n)), donde f(n) es el tiempo en el peor caso de la operación I en muchos casos esa cota no es ajustada debido a que el peor caso puede NO ocurrir las N veces, o incluso ninguna de esas veces I entonces se introducen las técnicas de análisis amortizado para tratar de obtener una cota menor para la serie de operaciones Algoritmos y Complejidad I el análisis amortizado se diferencia del análisis en el caso promedio en que no involucra probabilidades, y en que garantiza el tiempo en el peor caso de las N operaciones I el análisis probabilístico produce un tiempo esperado que una determinada ejecución puede sobrepasar o no I el análisis amortizado produce una cota en el tiempo de ejecución de la serie de operaciones (es decir, sigue siendo un análisis del peor de los casos)

1.1.1.4. Análisis en cada tipo de sentencia

Consideraremos que las operaciones de asignación, impresión en pantalla, operaciones lógicas (and, or, not, $<$, $>$, =), operaciones matemáticas simples ($+, -, *, /, \%$), los accesos a un elemento de un arreglo y las instrucciones de retornos de métodos se cumplen en un tiempo 1. Este tiempo es independiente de las máquinas y de los lenguajes utilizados.

Ejemplo:

$$\begin{array}{lll} (1) & a = 1 & T_{S1} = 1_{asig} = 1 \\ (2) & b = x * 21 - 5 & T_{S2} = 1_{asig} + 2_{OperMat} = 3 \\ (3) & c = arr[i] * 9 - 2 & T_{S3} = 1_{asig} + 2_{OperMat} + 1_{acceso} = 4 \end{array}$$

Secuencia: el tiempo del bloque completo es la suma de los tiempos de cada sentencia que lo componen. Ejemplo:

$$\begin{array}{ll} (1) & a = 1; \\ (2) & b = x * 21 - 5; \\ (3) & c = arr[i] * 9 - 2; \end{array}$$

$T_{Secuencia} = T_{S1} + T_{S2} + T_{S3} = 1 + 3 + 4 = 8$

Alternativa: El tiempo de ejecución de la alternativa nunca es más grande que el tiempo empleado por la evaluación de la condición más el mayor de los tiempos de *S1* y *S2*.

Ejemplo:

```
1  if (i+1 ≤ 2*p)
2      a[i+1] ← 2*i+5;
3  else
4      exito ← false;
```

$T_{S1} = T_{Condición} = 2_{OpMat} + 1_{OperLog} = 3$
$T_{S2} = 1_{asig} + 1_{OpMat} + 1_{Acceso} = 5$
$T_{S4} = 1_{asign} = 1$
$T_{Alternativa} = T_{S1} + max(T_{S2}; T_{S4})$
$T_{Alternativa} = 3 + max(5; 1) = 3 + 5 = 8$

Repetitiva: Se considera el tiempo de iniciación, el de incremento, el de evaluación de la condición.

- **Repetitiva *PARA***: se tiene en cuenta el tiempo de la inicialización de la variable de control (T_{ini}), luego por cada repetición del ciclo $\sum_{min}^{max} T_{CuerpoRep}$, donde min es el valor de inicialización y max es el último valor para el que la condición da $true$, sumamos el tiempo para evaluar la condición (T_{cond}), el tiempo del cuerpo de sentencias interno

del bucle (T_{int}), y el tiempo del incremento de la variable de control (T_{incr}). Además consideramos que la evaluación de la condición (T_{cond}) se ejecuta una vez más que las que se ejecuta el bloque interno (cuando la condición es falsa). Ejemplo:

```
1  for (i ← 3; i < 9; i++){
2      a[i] ← a[i+1];
3  }
```

$T_{ini} = 1_{asig} = 1$
$T_{cond} = 1_{opLog} = 1$
$T_{incr} = 1_{opMat} + 1_{asig} = 2$
$T_{int} = 1_{asig} + 2_{acceso} + 1_{opMat} = 4$
$T_{for} = T_{ini} + \sum_{min}^{max}(T_{cond} + T_{int} + T_{incr}) + T_{cond}$
$T_{for} = 1 + \sum_{3}^{8}(1 + 4 + 2) + 1$
$T_{for} = 1 + (8 - 3 + 1) * (1 + 4 + 2) + 1 = 1 + 6 * 7 + 1 = 44$

- **Repetitiva *MIENTRAS***: consideramos la cantidad de repeticiones (las veces que la condición da true); por cada repetición sumamos el tiempo para evaluar la condición (T_{cond}) y el tiempo del cuerpo de sentencias interno del bucle (T_{int}), y consideramos que la evaluación de la condición (T_{cond}) se ejecuta una vez más que las que se ejecuta el bloque interno (cuando la condición es falsa).

  Ejemplo:

```
1  i ← 0;
2  while (i < n){
3      a[i] ← a[i+1];
4      i ← i+1;
5  }
```

$T_{cond} = 1_{opLog} = 1$
$T_{S3} = 1_{asig} + 2_{acceso} + 1_{opMat} = 4$
$T_{S4} = 1_{asig} + 1_{opMat} = 2$
$T_{int} = T_{S3} + T_{S4}$
$T_{int} = 4 + 2 = 6$
$T_{mientras} = cantrep(T_{cond} + T_{int}) + T_{cond} = n * (1 + 6) + 1 = 7n + 1$

- **Repetitiva *HASTA***: consideramos la cantidad de repeticiones (las veces que la condición da $true$); por cada repetición sumamos el tiempo para evaluar la condición (T_{cond}) y el

tiempo del cuerpo de sentencias interno del bucle (T_{int}). A diferencia del $mientras$ y el $para$, esta sentencia evalúa la condición sólo la cantidad de veces que se ejecuta el cuerpo interno.

Ejemplo

$T_{int} = T_{S3} + T_{S4}$

```
1  i ← 0;
2  do {
3      a[i] ← a[i+1];
4      i ← i+1;
5  } while i ≤ n;
```

$T_{S3} = 1_{asig} + 2_{acceso} + 1_{opMat} = 4$
$T_{S4} = 1_{asig} + 1_{opMat} = 2$
$T_{int} = 4 + 2 = 6$
$T_{cond} = 1_{opLog} = 1$
$T_{hasta} = cantRep(T_{cond} + T_{int}) = n(1 + 6) = 7n$

1.2. Notación asintótica

La notación asintótica permite estudiar el comportamiento de un algoritmo cuando está dado en función de la magnitud de la entrada y ésta es lo suficientemente grande. No considera lo que ocurre con entradas pequeñas y obvia el comportamiento con factores constantes. Se dice que:

> Un algoritmo tiene un tiempo de ejecución de $O(g(n))$, para una función dada f, si existe una constante positiva c y una implementación del algoritmo capaz de resolver cada caso del problema en un tiempo acotado superiormente por $c \cdot g(n)$, donde n es el tamaño del problema considerado.

1.2.1. Orden de eficiencia

Para indicar el orden de eficiencia de una función $f(n)$ se define una cota superior para la función de crecimiento. Para un valor pequeño de n no tiene sentido analizar $f(n) < c \cdot g(n)$. Sin embargo, si existe una *cota de crecimiento* $O()$ se puede afirmar $f(n) < c \cdot g(n)$ a partir de un determinado valor n grande. Matemáticamente hablando, esto es cuando n tiende a infinito. Es decir, lo que nos interesa conocer es el comportamiento asintótico de la función del tiempo de ejecución. El conjunto de funciones que comparten un mismo comportamiento asintótico comparten un orden de complejidad. El orden de complejidad de $f(n)$ lo escribimos $O(g(n))$

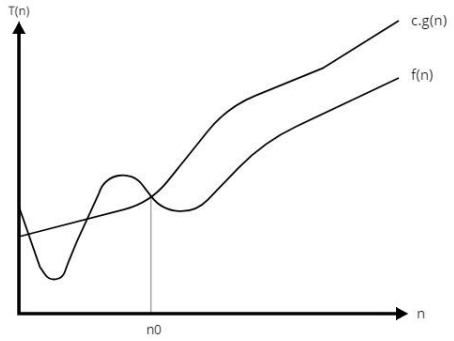

Figura 1.1: Orden de una función f: $f(n) = O(g(n))$

donde $f(n)$ representa la velocidad de crecimiento de la función del tiempo de ejecución, $g(n)$ crece tan deprisa como $f(n)$.

Es decir dada una función $f(n)$, queremos estudiar aquellas funciones $t(n)$ que a lo sumo crecen tan deprisa como f. Al conjunto de tales funciones se le llama cota superior de f y lo denominamos $O(g(n))$. Conociendo la cota superior de un algoritmo podemos asegurar que, en ningún caso, el tiempo empleado será de un orden superior al de la cota. En la Figura 1.1 el valor n_0 es la constante a partir de la cual

$$f(n) = O(g(n))$$

Abuso de notación: Aunque $O(g(n))$ es una clase de funciones se acostumbra escribir $f(n) = O(g(n))$ en vez de $f(n) \in O(g(n))$. Cabe observar que escribir $O(g(n)) = f(n)$ no tiene sentido.

Específicamente hablando de tiempos de ejecución de un algoritmo, la notación asintótica resulta:

Decimos que una función $T(n)$ es $O(g(n))$ si existen constantes n_0 y c tales que $T(n) \leq c \cdot g(n)$ para todo $n \geq n_0$

$$T(n) \text{ es } O(g(n)) \Leftrightarrow \exists\, c \in \mathbb{R}^+, \exists\, n_0 \in \mathbb{N} | \forall n \in \mathbb{N}, n > n_0, T(n) \leq c \cdot g(n)$$

1.2.1.1. Definición de Orden

Sea $g : \mathbb{N} \to [0, \infty)$. Se define el conjunto de funciones de orden O (Omicron) de g como:

$$O(g(n)) = \{t : \mathbb{N} \to [0, \infty) | \exists\, c \in \mathbb{R}, c > 0, \exists\, n_0 \in \mathbb{N} : t(n) \leq c \cdot g(n), \forall n \geq n_0\}$$

Diremos que una función $t : \mathbb{N} \to [0, \infty)$ es de orden O de g si $t \in O(g)$.

1.2.1.2. Principio de Invarianza

Dado un algoritmo y dos implementaciones suyas I_1 e I_2, que tardan $T_1(n)$ y $T_2(n)$ tiempos respectivamente, el $Principio\ de\ Invarianza$ afirma que existe una constante real $c > 0$ y un número natural n_0 tales que para todo $n > n_0$ se verifica que $T_1(n) <= c \cdot T_2(n)$.

Es decir, el tiempo de ejecución de dos implementaciones distintas de un mismo algoritmo dado, ya sea del mismo código ejecutado por dos máquinas de distinta velocidad, no va a diferir más que en una constante multiplicativa.

Si dos implementaciones tardan $T_1(n)$ y $T_2(n)$ tiempos respectivamente, se dice que

$$\exists c, d \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}, \text{ tal que } n > n_0, T_1(n) \leq c \cdot T_2(n) \text{ y } T_2(n) \leq d \cdot T_1(n)$$

1.2.1.3. Comparativa de órdenes

Siempre escogeremos el representante más sencillo para cada clase; así los órdenes de complejidad constante serán expresados por $O(1)$, los lineales por $O(n)$, etc.

Según el Cuadro 1.1 y la Figura 1.2 resulta que:

$$O(1) \subset O(log\ n) \subset O(n) \subset O(n \cdot log\ n) \subset O(n^2) \subset O(n^3) \subset O(2^n) \subset O(n!)$$

N	$O(log_2\ n)$	$O(n)$	$O(n\ log_2\ n)$	$O(n^2)$	$O(2^n)$	$O(n!)$
10	$3\mu s$	$10\mu s$	$30\mu s$	0,1 ms	1 ms	4 s
25	$5\mu s$	$25\mu s$	0,1 ms	0,6 ms	33 s	10^{11} $años$
50	$6\mu s$	$50\mu s$	0,3 ms	2,5 ms	36 $años$	...
100	$7\mu s$	$100\mu s$	0,7 ms	10 ms	10^{17} $años$	...
1000	$10\mu s$	1 ms	10 ms	1 s	...	...
10000	$13\mu s$	10 ms	0,1 s	100 s	...	...
100000	$17\mu s$	100 ms	1,7 s	3 $horas$	...	...
1000000	$20\mu s$	1 s	20 s	12 $días$	...	...

Cuadro 1.1: Comparativa de órdenes

Se puede calcular el orden de un algoritmo de dos maneras:

1. A partir de la función de tiempo de ejecución cuando se conoce la función del tiempo de ejecución, se aísla el término que crece más rápido, despreciando las constantes que lo multiplican. Ejemplo, si $T(n) = 4n^2 + 5n + 8$ su orden es $O(n^2)$.

2. A partir del código (o pseudocódigo), simplificando las sentencias de tiempo constante (orden $O(1)$) y aplicando las siguientes reglas: Sean $T_1(n)$ y $T_2(n)$ las funciones que

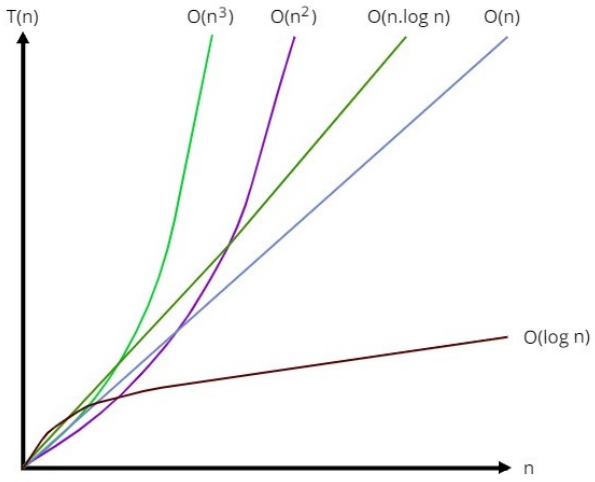

Figura 1.2: Crecimiento de O

expresan los tiempos de ejecución de dos fragmentos de un programa, y sus órdenes $T_1(n) \in O(g_1(n))$ y $T_2(n) \in O(g_2(n))$.

- **Regla de la suma**: El orden de la suma de dos bloques de código secuenciales es el máximo de sus órdenes.

$$O(g_1(n)) + O(g_2(n)) = O(max(g_1(n), g_2(n)))$$

- **Regla del producto**: El orden del producto de ambos bloques es el producto de sus órdenes.

$$O(g_1(n)) \cdot O(g_2(n)) = O(g(n) \cdot g_2(n))$$

Formalmente estas son las propiedades de O:

Propiedades de O

1. Para cualquier función f se tiene que $f \in O(f)$.
2. $f \in O(g) \Rightarrow O(f) \subseteq O(g)$.
3. $O(f) = O(g) \Leftrightarrow f \in O(g) \wedge g \in O(f)$.
4. Si $f \in O(g) \wedge g \in O(h) \Rightarrow f \in O(h)$
5. Si $f \in O(g) \wedge f \in O(h) \Rightarrow f \in O(min(g, h))$.
6. Regla de la suma: Si $f_1 \in O(g) \wedge f_2 \in O(h) \Rightarrow f_1 + f_2 \in O(max(g, h))$.

7. Regla del producto: Si $f_1 \in O(g) \wedge f_2 \in O(h) \Rightarrow f_1 \cdot f_2 \in O(g \cdot h)$.
8. Si existe $\lim_{n\to\infty} \frac{f(n)}{g(n)} = k$ según los valores de k tenemos:
   a) si $k \neq 0 \wedge k < \infty$ entonces $O(f) = O(g)$
   b) si $k = 0$ entonces $f \in O(g)$, es decir $O(f) \subset O(g)$, pero sin embargo se verifica que $g \notin O(f)$.

La regla más potente para demostrar que unas funciones están en el orden de otras es utilizando la regla del límite:
Sea $f, g : \mathbb{N} \to \mathbb{R}^+$

- Si $\lim_{n\to\infty} \frac{f(n)}{g(n)} \in \mathbb{R}$ entonces $f(n) \in O(g(n)) \wedge g(n) \in O(f(n))$
- Si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ entonces $f(n) \in O(g(n)) \wedge g(n) \notin O(f(n))$
- Si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$ entonces $f(n) \notin O(g(n)) \wedge g(n) \in O(f(n))$

1.2.2. Cota inferior Ω

Dada una función $f(n)$, queremos estudiar aquellas funciones $g(n)$ que a lo sumo crecen tan lentamente como $f(n)$. Al conjunto de tales funciones se le llama cota inferior de $f(n)$ y lo denominamos $\Omega(g(n))$. Conociendo la cota inferior de un algoritmo podemos asegurar que, en ningún caso, el tiempo empleado será de un orden inferior al de la cota.

Notación Ω (cota inferior)

$$T(n) \text{ es } \Omega(g(n)) \text{ cuando } \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N} : \forall n \geq n_0 \Rightarrow T(n) \geq c \cdot g(n)$$

Formalmente se define el conjunto de funciones de orden Ω (Omega) de g como:

$$\Omega(g(n)) = \{t : \mathbb{N} \to [0, \infty) | \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N} : \forall n \geq n_0, t(n) \geq c \cdot g(n)\}$$

Diremos que una función $f : \mathbb{N} \to [0, \infty)$ es de orden Ω de g si $f \in \Omega(g(n))$. Ver Figura 1.3.

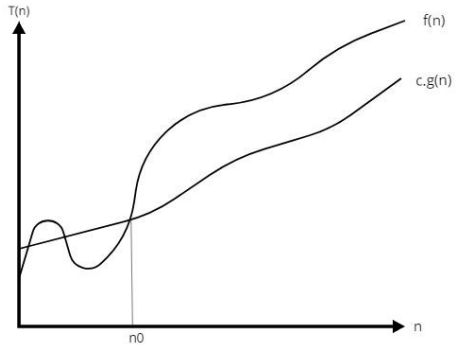

Figura 1.3: Cota inferior: $f(n) = \Omega(g(n))$

Intuitivamente, $f \in \Omega(g)$ indica que f está acotada inferiormente por algún múltiplo de g. Normalmente estaremos interesados en la mayor función g tal que f pertenezca a $\Omega(g)$, a la que denominaremos su cota inferior. Obtener buenas cotas inferiores es en general muy difícil, aunque siempre existe una cota inferior trivial para cualquier algoritmo: al menos hay que leer los datos y luego escribirlos, de forma que ésa sería una primera cota inferior. Así, para ordenar n números una cota inferior sería n, y para multiplicar dos matrices de orden n sería n^2; sin embargo, los mejores algoritmos conocidos son de órdenes $n \cdot log\ n$ y n^2 respectivamente.
Formalmente estas son las propiedades de Ω:

Propiedades de Ω

1. Para cualquier función f se tiene que $f \in \Omega(f)$.
2. $f \in \Omega(g) \Rightarrow \Omega(f) \subseteq \Omega(g)$.
3. $\Omega(f) = \Omega(g) \Leftrightarrow f \in \Omega(g) \wedge g \in \Omega(f)$.
4. Si $f \in \Omega(g) \wedge g \in \Omega(h) \Rightarrow f \in \Omega(h)$
5. Si $f \in \Omega(g) \wedge f \in \Omega(h) \Rightarrow f \in \Omega(min(g, h))$.
6. Regla de la suma: Si $f_1 \in \Omega(g) \wedge f_2 \in \Omega(h) \Rightarrow f_1 + f_2 \in \Omega(g + h)$.
7. Regla del producto: Si $f_1 \in \Omega(g) \wedge f_2 \in \Omega(h) \Rightarrow f_1 \cdot f_2 \in \Omega(g \cdot h)$.
8. Si existe $\lim\limits_{n\to\infty} \dfrac{f(n)}{g(n)} = k$ según los valores de k tenemos:
   a) si $k \neq 0 \wedge k < \infty$ entonces $\Omega(f) = \Omega(g)$
   b) si $k = 0$ entonces $g \in \Omega(f)$, es decir $\Omega(g) \subset \Omega(f)$, pero sin embargo se verifica que $f \notin \Omega(g)$.

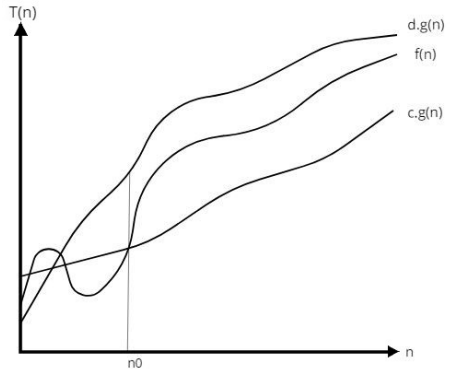

Figura 1.4: Orden exacto: $f(n) = \Theta(g(n))$

De las propiedades anteriores se deduce que la relación de equivalencia $\approx$, definida por $f \approx g$ si y sólo si $\Omega(f) = \Omega(g)$, es una relación de equivalencia. Al igual que hacíamos para el caso de la cota superior O, siempre escogeremos el representante más sencillo para cada clase. Así los órdenes de complejidad Ω constante serán expresados por $\Omega(1)$, los lineales por $\Omega(n)$, etc.

1.2.3. Orden Exacto. Notación Θ

Como última cota asintótica, definiremos los conjuntos de funciones que crecen asintóticamente de la misma forma.

Se define el conjunto de funciones de orden Θ (Theta) como:

$$\Theta(g(n)) = \{t : \mathbb{N} \to [0, \infty) | \exists c, d \in \mathbb{R}^+; \exists n_0 \in \mathbb{N} : \forall n \geq n_0, c \cdot g(n) \leq t(n) \leq d \cdot g(n)\}$$

Diremos que una función $f : \mathbb{N} \to [0, \infty)$ es de orden Θ de g si $f \in \Theta(g)$. Intuitivamente, $f \in \Theta(g)$ indica que f está acotada tanto superior como inferiormente por múltiplos de g, es decir, que f y g crecen de la misma forma.
Formalmente estas son las propiedades de Θ:

Propiedades de Θ

1. Para cualquier función f se tiene que $f \in \Theta(f)$.
2. $f \in \Theta(g) \Rightarrow \Theta(f) \subseteq \Theta(g)$.
3. $\Theta(f) = \Theta(g) \Leftrightarrow f \in \Theta(g) \wedge g \in \Theta(f)$.
4. Si $f \in \Theta(g) \wedge g \in \Theta(h) \Rightarrow f \in \Theta(h)$

5. Regla de la suma: Si $f_1 \in \Theta(g) \land f_2 \in \Theta(h) \Rightarrow f_1 + f_2 \in \Theta(max(g,h))$.
6. Regla del producto: Si $f_1 \in \Theta(g) \land f_2 \in \Theta(h) \Rightarrow f_1 \cdot f_2 \in \Theta(g \cdot h)$.
7. Si existe $\lim_{n\to\infty} \frac{f(n)}{g(n)} = k$ según los valores de k tenemos:
   a) si $k \neq 0 \land k < \infty$ entonces $\Theta(f) = \Theta(g)$
   b) si $k = 0$ entonces los órdenes de f y g son diferentes.

Capítulo 2

Demostraciones

2.1. Formas de demostración

Para demostrar que una función dada no pertenece al orden de otra función tendremos estas formas:

- **Demostración por contradicción**: Es la forma más sencilla. Consiste en demostrar la veracidad de una sentencia demostrando que su negación da lugar a una contradicción.
- **La regla del umbral generalizado**: Implica la existencia de una constante real y positiva c tal que $f(n) \leq c \cdot g(n)$ para todos los $n \geq 1$ (tomaremos n_0 como 1, nos interesa más la definición dada por la regla del umbral sin generalizar).
- **La regla del límite**: Lo definiremos completamente tras analizar la cota superior y el coste exacto. Nos permite comparar dos funciones en cuanto a la notación asintótica se refiere. Tendremos que calcular el siguiente límite:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)}$$

$$(1)\ \lim_{n\to\infty} \frac{f(n)}{g(n)} = c \in \mathbb{R}^+ \left\{ \begin{array}{ccc} f(n) \in O(g(n)) & f(n) \in \Omega(g(n)) & f(n) \in \Theta(g(n)) \\ g(n) \in O(f(n)) & g(n) \in \Omega(f(n)) & g(n) \in \Theta(f(n)) \end{array} \right\}$$

$$(2)\ \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \left\{ \begin{array}{ccc} f(n) \notin O(g(n)) & f(n) \in \Omega(g(n)) & f(n) \notin \Theta(g(n)) \\ g(n) \in O(f(n)) & g(n) \notin \Omega(f(n)) & g(n) \notin \Theta(f(n)) \end{array} \right\}$$

$$(3)\ \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \left\{ \begin{array}{ccc} f(n) \in O(g(n)) & f(n) \notin \Omega(g(n)) & f(n) \notin \Theta(g(n)) \\ g(n) \notin O(f(n)) & g(n) \in \Omega(f(n)) & g(n) \notin \Theta(f(n)) \end{array} \right\}$$

2.1.1. Para tener en cuenta, regla de L'Hopital

Sean f y g dos funciones continuas definidas en el intervalo $[a, b]$, derivables en (a, b) y sea c perteneciente a (a, b) tal que $f(c) = g(c) = 0$ y $g'(x) \neq 0$ si $x \neq c$.
Si existe el límite L de f'/g' en c, entonces existe el límite de f/g (en c) y es igual a L.
Por lo tanto,

$$\lim_{x \to c} \frac{f(x)}{g(x)} = \lim_{x \to c} \frac{f'(x)}{g'(x)} = L$$

2.2. Observaciones sobre las cotas asintóticas

La utilización de las cotas asintóticas para comparar funciones de tiempo de ejecución se basa en la hipótesis de que son suficientes para decidir el mejor algoritmo, prescindiendo de las constantes de proporcionalidad. Sin embargo, esta hipótesis puede no ser cierta cuando el tamaño de la entrada es pequeño.

Para un algoritmo dado se pueden obtener tres funciones que miden su tiempo de ejecución, que corresponden a sus casos mejor, medio y peor, y que denominaremos respectivamente $T_{mejor}(n)$, $T_{prom}(n)$ y $T_{peor}(n)$. Para cada una de ellas podemos dar tres cotas asintóticas de crecimiento, por lo que se obtiene un total de nueve cotas para el algoritmo.

Para simplificar, dado un algoritmo diremos que su orden de complejidad es $O(t_p)$ si su tiempo de ejecución para el peor caso es de orden O de t_p, es decir, $T_{peor}(n)$ es de orden $O(t_p)$. De forma análoga diremos que su orden de complejidad para el mejor caso es $\Omega(t_m)$ si su tiempo de ejecución para el mejor caso es de orden Ω de t_m, es decir, $T_{mejor}(n)$ es de orden $\Omega(t_m)$. Por último, diremos que un algoritmo es de orden exacto $\Theta(t_e)$ si su tiempo de ejecución en el caso medio $T_{prom}(n)$ es de este orden.

2.3. Ejemplos

2.3.1. Ejemplo: Demostrar que $f(n) = 5 \cdot 2^n + n^2$ está en el orden exacto de 2^n

Sean $f(n) = 5 \cdot 2^n + n^2$ y $g(n) = 2^n$
Entonces

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{5 \cdot 2^n + n^2}{2^n} \xrightarrow{L'Hopital}$$

$$\lim_{n\to\infty} \frac{5 \cdot 2^n \cdot log\ 2 + 2 \cdot n}{2^n \cdot log\ 2} \xrightarrow{L'Hopital} \lim_{n\to\infty} \frac{5 \cdot 2^n \cdot (log\ 2)^2 + 2}{2^n \cdot (log\ 2)^2} \xrightarrow{L'Hopital}$$

$$\lim_{n\to\infty} \frac{5 \cdot 2^n \cdot \cancel{(log\ 2)^3}}{\cancel{2^n \cdot (log\ 2)^3}} = 5$$

Como el límite da un valor constante $5 \neq 0 \wedge 5 < \infty$ entonces por el caso **a)** de la propiedad **7** de Orden Exacto podemos decir que $\Theta(5 \cdot 2^n + n^2) = \Theta(2^n)$. Y por propiedad 3, decimos que $f(n) = 5 \cdot 2^n + n^2 \in \Theta(2^n)$

2.3.2. Ejemplo: Demostrar que $f(n) = n! \notin \Theta((2 \cdot n + 1)!)$

Sean $f(n) = n!$ y $g(n) = (2n + 1)!$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n!}{(2n+1)!} \xrightarrow{L'Hopital}$$

$$\lim_{n\to\infty} \frac{\cancel{n!}}{(2n+1) \cdot (2n) \cdot (2n-1) \cdot ... \cdot \cancel{n!}} = 0$$

Por propiedad 7 de orden exacto, nos encontramos en el caso b) donde $k = 0$ entonces los órdenes exactos de $f(n) = n!$ y $g(n) = (2n + 1)!$ son distintos. Concluimos que $f(n) = n! \notin \Theta((2 \cdot n + 1)!)$

2.3.3. Ejemplo: Análisis de eficiencia del MCD

El problema del cálculo del máximo común divisor de dos valores enteros m y n (ambos mayores a cero), que se denota $mcd(m, n)$ corresponde al mayor entero que divide a la vez a m y n de manera uniforme, es decir, con un resto cero.

Una de las primeras soluciones fue dada por Euclides de Alejandría (siglo 300 ac), donde básicamente analiza la igualdad $mcd(m, n) = mcd(n, mmodn)$, donde $m\ mod\ n$ es el resto de

la división de m por n, hasta $m \; mod \; n$ es igual a 0. Cuando se llega a $mcd(m, 0) = m$, resulta este último valor ser el común divisor de los valores de m y n dados inicialmente.

Por ejemplo, $mcd(60, 24)$ se puede calcular de la siguiente manera:

$$mcd(60, 24) = mcd(24, 12) = mcd(12, 0) = 12$$

```
1 MODULO Euclides (m, n) RETORNA ENTERO
2     MIENTRAS que n <> 0 HACER
3         r ← m mod n
4         m ← n
5         n ← r
6     FIN MIENTRAS
7 RETORNAR m
```

El valor de n se va reduciendo en cada iteración hasta convertirse en cero, luego finaliza retornando el valor que queda en la variable m.

Un segundo algoritmo podría consistir en definir un contador que comienza en el menor valor m o n, y se detiene cuando alcanza un valor divisor de ambos números. El procedimiento consiste en ir dividiendo ambos números por el contador, guardando el valor de este cuando se detecte que divida a ambos números.

Por último, una tercera solución algorítmica podría basarse en la descomposición de factores primos. En el Cuadro 2.1 se muestra un análisis de eficiencia para cada caso.

2.3.4. Ejemplo: Demostrar que $f \in O(g) \Leftrightarrow g \in \Omega(f)$

Por la definición de O, sabemos que

$$f \in O(g)$$

$$f \in \{t : \mathbb{N} \to [0, \infty) | \exists \, c \in \mathbb{R}, c > 0, \exists n_0 \in \mathbb{N} : t(n) \leq c \cdot g(n), \forall n \geq n_0\}$$

Por la definición de Ω tenemos que

$$g \in \Omega(f)$$

$$g \in \{t : \mathbb{N} \to [0, \infty) | \exists \, d \in \mathbb{R}, d > 0, \exists n_1 \in \mathbb{N} : t(n) \geq d \cdot f(n), \forall n \geq n_1\}$$

1. $\Rightarrow$) Si $f \in O(g)$ entonces $f(n) \leq c \cdot g(n)$ con un $c > 0$. Tenemos también que $1/c \cdot f(n) \leq g(n)$.
   $\therefore$ si tomamos $d = 1/c$ podemos afirmar que $d \cdot f(n) \leq g(n)$ y que $g \in \Omega(f)$.

2. $\Leftarrow$) En forma recíproca, si $g \in \Omega(f)$ entonces $g(n) \geq d \cdot f(n)$ con un $d > 0$. Tenemos también que $1/d \cdot g(n) \geq f(n)$.
   $\therefore$ si tomamos $c = 1/d$ podemos afirmar que $c \cdot g(n) \leq f(n)$ y que $f \in O(g)$.

Soluciones algorítmicas	Clásico	A partir de un contador	Euclides

Esquema algorítmico	• Descomponer en factores primos
	• Tomar el producto de los factores primos comunes de m y n, (cada factor primo en la menor potencia de los dos argumentos)
		`mcd(m,n)`
		`i <- min(m,n) + 1`
		`repetir mientras`
		`  i <- i - 1`
		`hasta i divide a`
		`  m y n exactamente`
		`retornar i`
			`euclides (m,n)`
			`while m > 0 do`
			`  t <- n mod m`
			`  n <- m`
			`  m <- t`
			`retornar n`

Tiempo consumido en relación a la entrada de datos	Requiere factorizar m y n, una operación que necesita como máximo un recorrido hasta la mitad de cada argumento, y luego combinar los factores obtenidos por tanto el tiempo es lineal $((m + n)/2)$	Es proporcional a la diferencia entre el menor de los dos argumentos y su máximo común divisor. Cuando m y n son de tamaño similar y primos entre sí, toma por tanto un tiempo lineal (n)	Consume un tiempo en el orden del logaritmo de sus argumentos, siendo el mejor en eficiencia.

Cuadro 2.1: Comparativa de tiempos en diferentes soluciones algorítmicas

de $\Rightarrow$) y $\Leftarrow$) podemos concluir que

$$f \in O(g) \Leftrightarrow g \in \Omega(f)$$

2.3.5. Ejemplo: Demostrar la propiedad de transitividad para Ω

Demostraremos la propiedad transitividad para la cota inferior Ω:

$$f(n) \in \Omega(g(n)) \wedge g(n) \in \Omega(h(n)) \Rightarrow f(n) \in \Omega(h(n))$$

Por la definición de Ω sabemos que

$$f \in \Omega(g) \wedge g \in \Omega(h) \Rightarrow$$

$$f \in \{t : \mathbb{N} \to [0, \infty) | \exists\, c_1 \in \mathbb{R}, c_1 > 0, \exists\, n_0 \in \mathbb{N} : t(n) \geq c_1 \cdot g(n), \forall n \geq n_0\} \wedge$$

$$g \in \{t' : \mathbb{N} \to [0, \infty) | \exists\, c_2 \in \mathbb{R}, c_2 > 0, \exists\, n_1 \in \mathbb{N} : t'(n) \geq c_2 \cdot h(n), \forall n \geq n_1\}$$

En particular, si $f \in \Omega(g)$ y $g \in \Omega(h)$ tenemos que

$$f(n) \geq c_1 \cdot g(n) \wedge g(n) \geq c_2 \cdot h(n), \text{ con } c_1, c_2 > 0$$

$$\frac{1}{c_1} \cdot f(n) \geq g(n) \wedge g(n) \geq c_2 \cdot h(n)$$

Por propiedad transitiva del operador $\geq$, tenemos

$$\frac{1}{c_1} \cdot f(n) \geq c_2 \cdot h(n)$$

$$f(n) \geq c_1 \cdot c_2 \cdot h(n)$$

Si tomamos $d = c_1 \cdot c_2$ tenemos

$$f(n) \geq d \cdot h(n)$$

Por lo tanto podemos afirmar que

$f \in \{t : \mathbb{N} \to [0, \infty) | \exists\, d \in \mathbb{R}, d > 0, \exists\, n_0 \in \mathbb{N} : t(n) \geq d \cdot h(n), \forall n \geq n_0\} \Rightarrow f \in \Omega(h)$