Análisis de Algoritmos

M. Andrea Rodríguez-Tastets
Ayudante: Erick Elejalde

Universidad de Concepción,Chile
www.inf.udec.cl\ ∼andrea
andrea@udec.cl

I Semestre - 2013

Origen de la Palabra Algoritmo

El concepto de algoritmo es fundamental en ciencias de la computación. La palabra proviene de al-Khowarizmi, el sobrenombre de un matemático y astrónomo medieval de Uzbekistán, Mohammed bin-Musa, quien definió las reglas para efectuar las 4 operaciones aritmáticas básicas. Khowarizmi significa “de Khowarizm”, lugar de procedencia de Mohammed bin-Musa. Su trabajo dió originen a un esquema de numeración que fue nombrado ‘algorismi” y finalmente algoritmo.

Algoritmo (1/2)

Es un conjunto finito de instrucciones no ambiguo y efectivo (termina en un intervalo finito de tiempo) que solucionan un problema. Sus características son:

- Input: cero o más cantidades o valores son ingresados
- Output: Al menos un valor es entregado
- Cada instrucción en computable (propiedad de definición)
- El algoritmo se detiene: entrega resultado o dice que los resultados no existen (propiedad de detención o efectividad): Si un conjunto de instrucciones cumple con las propiedades de un algoritmo excepto por ser finito en tiempo, se denomina *proceso computacional*.

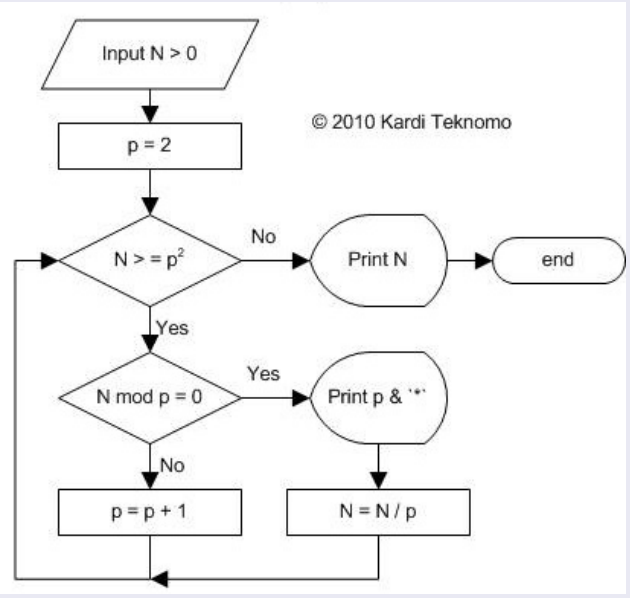
Formas de expresar un algoritmo

Para poder analizar un algoritmo debe existir una descripción clara de los pasos que se deben llevar a cabo por el algoritmo. Las formas más comunes de expresar un algoritmo son:

- Lenguaje natural que suele ser impreciso.
- Lenguaje de programación que es preciso pero difícil de entender
- Seudo-código que es un nivel intermedio que se puede describir como un lenguaje que nunca reclama por errores sintácticos.

En esta asignatura usaremos los tres métodos, usando lenguaje natural para describir ideas general y lenguajes más preciso para describir detalles necesarios.

Algoritmos: seudo-código



Algoritmo **F(N)**

```
     begin
1      p ← 2
2      while (N > p^2) do
3       if N mod p = 0then
4         Print p
5         N = N/p
6      else
7         p = p + 1
6      endif
8      endWhile
9      Print N
     end
```

Qué revisaremos: Técnicas de diseño de algoritmos

- Dividir y vencer
- Backtracking
- Métodos greedy
- Programación dinámica
- Búsqueda básica y técnicas de recorrido
- Grafos
- Problemas NP y Algoritmos de aproximación
- Programación lineal (como algoritmo de aproximación)

Formas de análisis

- Estructura de un algoritmo
- Validación de correctitud
- Análisis de tiempo y uso de espacio
- Cómo transformar un algoritmo para solucionar un problema
- Cómo implementar y evaluar un algoritmo: mantener el código corto y cercano al algoritmo

Demostrar que un algoritmo es incorrecto (1/2)

La mejor forma de demostrar que un algoritmo es incorrecto es dar una instancia del problema para el cual el algoritmo entrega una respuesta incorrecta. Estas instancias se llaman contra ejemplos. Las propiedades de un buen contra ejemplo son:

- Verificación. Se necesita saber la respuesta del algoritmo y la que debería dar.
- Simplicidad

Demostrar que un algoritmo es incorrecto (1/2)

Considere el problema de la mochila: Dado un conjunto de enteros $S = \{s_1, s_2, \ldots, s_n\}$, y un número T, encontrar el subconjunto de S más grande para el cual la suma sea T.
Mostrar que dado un S y T, los siguientes algoritmos no encuentran una solución correcta al problema de la mochila

- Poner los elementos en la mochila de menor a mayor
- Poner los elementos en la mochila de izquierda a derecha si ellos calzan, sino no.

Prueba de Correctitud (1/3)

Frecuentemente inducción matemática es la forma adecuada de verificar si un algoritmo recursivo o de inserción incremental es correcto.
Considere el caso de un ordenamiento por inserción:

- El caso base de inducción consiste de un elemento, y por definición, el arreglo está ordenado.
- Se asume que los primeros $n-1$ elementos están ordenados después de $n-1$ iteraciones del algoritmo.
- Para insertar un último elemento x en el arreglo A, encontramos donde debe ir. Esto se hace moviendo todos los elementos mayores hacia el final de una posición, dejando el espacio para insertar x. $\square$

Prueba de Correctitud (2/3)

Considere el siguiente algoritmo de multiplicación que es recursivo para $c \geq 2$.

Function **Multiplicar(y,z)**

```
1   if z = 0 then
2     return 0
3   else
4     return(Multiplicar(c * y, ⌊z/c⌋) + y * (z mod c))
```

La prueba de que el algoritmo es correcto por inducción es:

- La base de inducción es que $z = 0$, la que entrega obviamente el valor 0.
- Asumamos ahora que funciona bien para valores de $z \leq n - 1$.
- Para $z = (n-1)+1$, considere primero que z es un múltiplo de c, en cuyo caso, $((n-1)+1) = cm$ mod c es 0 y la expresión se resuelve por:

$$\begin{aligned} Multiplicar(c * y, \lfloor((n-1)+1)/c\rfloor) + y * (((n-1)+1) mod c)) = \\ Multiplicar(c * y, ((n-1)+1)/c) = \\ c * y * (n-1)/c + c * y * 1/c = y * (n-1) + y \\ Multiplicar(y, n-1) + y \end{aligned} \quad (1)$$

□

Prueba de Correctitud (3/3)

Function **Multiplicar(y,z)**

```
1   if z = 0 then
2     return 0
3   else
4     return(Multiplicar(c * y, ⌊z/c⌋) + y * (z mod c))
```

Para el caso que $z = (n-1)+1$ no es múltiplo de c, asuma que $z = cm + d = (n-1)+1$, con $d < c$. Entonces, la expresión es

$$Multiplicar(c * y, \lfloor((n-1)+1)/c\rfloor + y * (((n-1)+1) mod c)) =$$
$$Multiplicar(c * y, m + y * d) = cy * (m + d) = y * (cm + d - 1) + y =$$
$$y * (n-1) + y = Multiplicar(y, n-1) + y$$

□

Concepto de complejidad de algoritmos

El análisis de la complejidad de algoritmos es el estudio teórico del rendimiento y uso de recursos de un algoritmo.

Análisis de algoritmos: temas importantes

- Complejidad de tiempo: dependencia del tiempo respecto al tamaño de entrada
- Complejidad de espacio
- Influencia del tamaño de palabra del procesador: procesador único o multiple)
- Calidad de la solución: solución exacta o aproximada.

Análisis de algoritmos: RAM model

- Un procesador ejecuta una instrucción en cada paso
- Cada instrucción toma una unidad de tiempo (lo cual no siempre es cierto)
- Los operadores son tamaño fijo
- El almacenamiento es de tamaño fijo (RAM y disco)

¿Por qué importa el rendimiento?

- Rendimiento distingue lo que es posible de lo imposible.
- Algoritmos definen una comunicación en un lenguaje que permite definir costos.
- Permite la búsqueda de soluciones eficientes (rápidas) y no solo eficaces.

Métodos de análisis de complejidad de algoritmos

- Análisis asintótico. Diferentes medidas de interes son el peor caso, el mejor caso, y el caso promedio.
- Análisis teórico que crea una relación recurrente que relaciona la solución del problema original a la solución de problemas más pequeños.
- Implementación en papel: sumar los pasos del agoritmo en función del tamaño de entrada.
- Implementación en un lenguaje de computación y ejecución

Algoritmo: ejemplo

- Problema: Dado una lista de números, entregar el k-ésimo más grande.
- Algoritmo 1: Almacenar los números en un arreglo, ordernarlos y luego sacar el k-ésimo.
- Algoritmo 2: Almacenar los primeros k números en un arreglo y ordenarlos en forma descendiente. Por cada número remanente, si el número es mayor que el k-ésimo, insertarlo en la posición correcta del arreglo. Recuperar el número en la posición k.

¿Cuál es mejor?

Función de complejidad

La función de complejidad $f(n)$, donde n es el tamaño del problema, es una medida de la cantidad de recursos que requiere un algoritmo. Puesto que la cantidad de recursos crece con el tamaño del problema, entonces $f(n) > f(m) \iff n > m$ (monótona creciente). La función de complejidad puede ser espacial (si se mide el espacio ocupado) o temporal (si se mide tiempo de ejecución).

Elección del tamaño

Problema	Tamaño
Búsqueda de un elemento en el conjunto	Número de elementos en el conjunto
Multiplicar dos matrices	Dimensión de las matrices
Recorrer un árbol	Número de nodos en el árbol
Resolver un sistema de ecuaciones lineales	Número de ecuaciones y/o incógnitas
Ordenar un conjunto de valores	Número de elementos en el conjunto

Análisis espacial

El análisis espacial determina la cantidad de memoria que utiliza un algoritmo, para lo cual se suman todas las celdas de memoria que utiliza. En general hay dos tipos de celdas de memoria que son requeridas por los algoritmos:

- *Celdas estáticas*. Son las que se utilizan en todo el tiempo que dura la ejecución de un algoritmo.
- *Celdas dinámicas*. Son las que se utilizan durante un momento de la ejecución, y por lo tanto pueden ser asignadas y devueltas conforme se ejecuta el algoritmo.

Análisis temporal

El análisis temporal se expresa en el número de operaciones que realiza un algoritmo. Se asume que cada operación requiere una cierta cantidad constante de tiempo para ser ejecutada.

Ejemplo 1: Búsqueda lineal

	Algoritmo	
	BusquedaLineal(valor, A,n): integer	
	begin	
1	$i \leftarrow 1$;	1
2	**while** $(i \leq n) \wedge (A[i] \neq valor)$ **do**	$k+1$
3	$i \leftarrow i+1$;	k
	endWhile	
4	**return** i	1
	end	

La variable k depende de la entrada de datos.

Ejemplo 1 continuación

En este ejemplo se puede notar que el número de veces que se ejecutan algunas operaciones, para valores sucesivamente mayores que k, presenta un modelo de crecimiento similar al que tiene el número total de operaciones que ejecuta el algoritmo. Para estimar el tiempo total entonces, no se necesita sumar todo, sino identificar una *operación básica* que observe el comportamiento del algoritmo y contar el número de veces que se ejecuta esa operación. En este algoritmos existen alternativas de operaciones básicas: sumas, asignaciones, comparaciones del índice, comparaciones con elementos de A y acceso a elementos de A. En este caso, consideraremos como operación básica la comparación entre elementos del arreglo y el valor buscado.

Sea $I = \{I_1, \ldots, I_m\}$ el conjunto de instancias del problema, conjunto de tamaño es m. Sea $O = \{O_1 \ldots, O_m\}$ el conjunto formado por el número de operaciones básicas que un algoritmo realiza para resolver cada instancia. Entonces, O_j es el número de operaciones ejecutadas para resolver la instance I_j con $1 \leq j \leq m$. Se distinguen los siguientes casos:

- Caso peor: $f(n) = max(\{O_1, \ldots, O_m\})$
- Caso mejor: $f(n) = min(\{O_1, \ldots, O_m\})$
- Caso medio: $f(n) = \sum_{i=1}^{m} O_i P(i)$, con $P(i)$ la probabilidad de que ocurra instancia I.

Continuación ejemplo 1

- Caso peor ocurre cuando el valor no se encuentra: $f(n) = n + 1$
- Caso mejor ocurre cuando el valor está en la primera posición: $f(n) = 1$
- Caso medio: $f_t(n) = 1P(1) + 2P(2) + \ldots nP(n) + (n+1)P(n+1)$, donde $P(i)$ es la probabilidad de que el valor se encuentre en la localidad i, $(1 \leq i \leq n)$ y $P(n+1)$ es la probabilidad de que no esté en el arreglo. Se supone que todos son igualmente probables, entonces $P(i) = 1/(n+1)$. Así:

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

Ejemplo 2

- Problema: dado un arreglo de valores, encontrar el producto de los dos números mayores.
- Tamaño del problema: n, el número de elementos en el arreglo.
- Operación básica: comparación

Ejemplo 2: Producto2Mayores

```
        Algoritmo
        Producto2Mayores(A,n): real
        begin
 1        if A[1] > A[2] then
 2          mayor1 ← A[1];
 3          mayor2 ← A[2];
 4        else
 5          mayor2 ← A[1];
 6          mayor1 ← A[2];
          endif
 7        i ← 3;
 8        while i ≤ n do
 9          if A[i] > mayor1 then
10          mayor2 ← mayor1;
11          mayor1 ← A[i];
12          else if A[i] > mayor2 then
13            mayor2 ← A[i];
            endif
14          i + 1;
          endWhile
15        return mayor1 × mayor2;
        end
```

Continuación ejemplo 2

Este algoritmo realiza las siguientes operaciones:

1. Comparación entre $mayor1$, $mayor2$ y los elementos del arreglo
2. Asignación a $mayor1$ y $mayor2$
3. Asignación al índice
4. Return
5. Producto de los mayores
6. Incremento del índice
7. Comparación entre el índice i y la longitud del arreglo n

(3) ,(6) y (7) no las considero básicas por realizarse entre índices, (4) y (5) no por realizarse solo una vez. Entonces solo escojo entre (1) o (2).

Continuación ejemplo 2

- Caso peor ocurre cuando el arreglo tiene en las dos primera posiciones los valores mayores (se hacen dos comparaciones por iteración): $f(n) = (2 \times (n-2)) = 2n - 3$
- El caso mejor ocurre cuando el arreglo está ordenado en forma ascendente (en cada iteración se hace solo la primera comparación): $f(n) = (1 \times (n-2)) = n - 1$

Continuación ejemplo 2

- Este algoritmo tiene $\binom{|U|}{n}$ $n!$ posibles entradas, donde U es el conjunto del que se extraen los elementos del arreglo. Considerar todas las posibles entradas no tiene sentido para efectos de determinar la función de complejidad, sino lo que se necesita es determinar las instancias de casos diferentes de ejecución del algoritmo.

  El algoritmo hace siempre una comparación al inicio y en el interior del ciclo puede ser que se realice una comparación o dos: (i) $A[i] > mayor1$ se hace una comparación, (2) si $mayor1 \geq A[i] > mayor2$ se hacen dos comparaciones y (iii) si $mayor2 \geq A[i]$ se hacen dos comparaciones. Si cada paso tiene la misma probabilidad de ocurrencia, entonces en promedio, se harán $1/3(1+2+2) = 5/3$ comparaciones por iteración, por lo tanto el algoritmo realiza en promedio $f(n) = 5/3(n-2)$ comparaciones.

Ejemplo de operaciones básicas

Problema	**Operación Básica**
Búsqueda de un elemento en un conjunto	Comparación entre el valor y los elementos del conjunto
Multiplicar matrices	Producto de elementos de las matrices
Recorrer el árbol	Visitar nodos
Resolver un sistema de ecuaciones lineales	Suma
Ordenar un conjunto de valores	Comparación entre valores

Demostraciones

- Prueba por contradicción: Se asume que un teorema es falso, se muestra que esta presunción implica que una propiedad que se sabe verdades sea falsa, entonces la hipótesis original debe ser verdadera.
- Inducción matemática: Se prueba el caso base trivial, se asume verdad para un k y se muestra que la hipótesis es verdadera para un $k + 1$. Esto se usa para algoritmos recursivos.

Dominio asintótico

Cuando el tamaño del problema es grande, la cantidad de recursos que el algoritmo necesite puede crecer tanto que haga que su ejecución no sea posible. Por lo tanto, lo que interesa es sabe cómo se comporta un algoritmo para problemas grandes. De esta manera se estudia la velocidad de crecimiento de la cantidad de recursos en función del tamaño del problema, es decir, se estudiará el comportamiento asintótico de la función de complejidad $f(n)$.

Definición 1

Sean f y g funciones de $\mathbb{N}$ a $\mathbb{R}$. Se dice que g domina asintóticamente (d.a.) a f o que f es dominada asintóticamente por g, si $\exists k \geq 0$ y $m \geq 0$ tales que:

$$|f(n)| \leq m|g(n)|, \forall n \geq k$$

Puesto que las funciones de complejidad son funciones con dominio $\mathbb{N}$, y contradominio $\mathbb{R}$, los conceptos y propiedades del dominio asintótico proporcionan una manera conveniente de expresarlas y manipularlas. Note que en el caso de funciones de complejidad, las funciones $f(n)$ y $g(n)$ son siempre positivas, por lo que usualmente no se considera el valor absoluto de la función y $m > 0$.

Ejemplo 3

Sean $f(n) = n$ y $g(n) = n^3$:

1. Demostrar que g domina asintóticamente a f. Esto es, demostrar que

$$\exists m \geq 0, k \geq 0 \text{ tales que } |f(n)| \leq m|g(n)|, \forall n \geq k$$

2. Demostrar que f no domina asintóticamente a g. Expresado matemáticamente:

$$\neg(\exists m \geq 0, k \geq 0 \text{ tales que } |g(n)| \leq m|f(n)|, \forall n \geq k)$$

lo que es equivalente a

$$\forall m \geq 0, k \geq 0 \exists n \geq k \text{ tales que } |g(n)| > m|f(n)|$$

Ejemplo 3 (cont.)

Sean $f(n) = n$ y $g(n) = n^3$:
(1) Demostrar

$$\exists m > 0, k \geq 0 \text{ tales que } |f(n)| \leq m|g(n)|, \forall n \geq k$$

Sustituyendo $f(n)$ y $g(n)$ da

$$|n| \leq m|n^3|, \forall n \geq k$$

Si se toma $m = 1$ y $k = 0$, las desigualdades anteriores se cumplen, por lo tanto, m y k existen, y en consecuencia g domina asintóticamente a f.

Ejemplo 3 (cont.)

Sean $f(n) = n$ y $g(n) = n^3$:
(2) Demostrar

$$\forall m \geq 0, k \geq 0 \exists n \geq k \text{ tales que } |g(n)| > m|f(n)|$$

Sustituyendo $f(n)$ y $g(n)$ en cada lado de la desigualdad

$$|n^3| > m|n|$$

Simplificando,

$$n > \sqrt{m}$$

Si se toma $n > max(\sqrt{m}, k)$ ambas desigualdades se cumplen, y $\therefore$ f no domina asintóticamente a g.

Relación: complejidad y análisis asintótico

Si $t(n)$ es la cantidad real de recurso que se consume para una implementación específica del un algoritmo, se tiene entonces que:

$$\begin{aligned} t(n) &\propto f(n) \text{ i.e., proporcional} \\ t(n) &= kf(n) \\ |t(n)| &\leq k|f(n)| \end{aligned}$$

Es decir $f(n)$ domina asintóticamente a cualquier $t(n)$, lo que es lo mismo a decir que $t(n)$ se rige por el modelo de crecimiento de $f(n)$.

Teorema 1

Sea M el conjunto de funciones de $\mathbb{N}$ to $\mathbb{R}$. La relación binaria sobre este conjunto definida como

$$\{< f, g > | f, g \in M \text{ y } g \text{ domina asintóticamente a } f\}$$

es reflexiva y transitiva.

Teorema 1: Demostración

(1) Se debe demostrar que la relación es reflexiva, es decir,
$f \in M \Rightarrow f \ \mathrm{d.a.}\ f$.
Si tomamos $m = 1$ y $k = 0$ la desigualdad $|f(n)| \leq m|f(n)|, \forall n \geq k$ se cumple, $\therefore$ la relación es reflexiva.

Teorema 1: Demostración (cont.)

(2) Se debe demostrar que la relación es transitiva, es decir, si $f, g, h \in M$ tal que f d.a. g y g d.a. $h \Rightarrow f \,\mathrm{d.a.}\, h$.
Sabemos que $\exists m_1, m_2, k_1$ y $k_2 \geq 0$ tales que:

$$|g(n)| \leq m_1|f(n)|, \forall n \geq k_1 \text{ y } |h(n)| \leq m_2|g(n)|, \forall n \geq k_2$$

Entonces

$$|h(n)| \leq m_1 m_2|f(n)|, \forall n \geq max(k_1, k_2)$$

con $m = m_1 m_2$ y $k = max(k_1, k_2)$, se satisface la definición, $\therefore$ f d.a. h y la relación es transitiva.

Teorema 2

Sean f y g funciones en M, la relación de equivalencia binaria $f \equiv g \Leftrightarrow f$ y g se dominan asintóticamente es una relación de equivalencia sobre M.

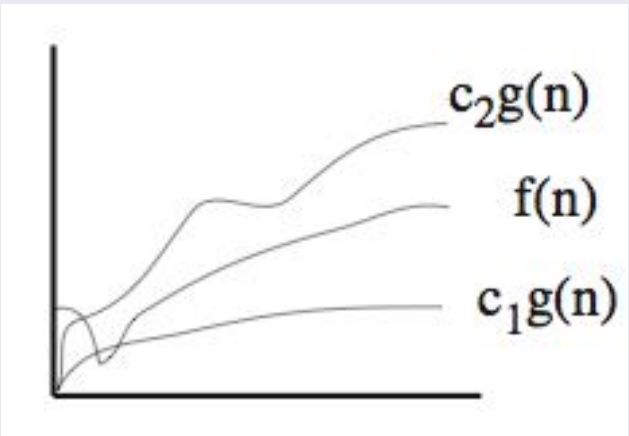
Teorema 2: Demostración

Se requiere demostrar que la relación es simétrica, transitiva y reflexiva.
(i) Puesto que $f \equiv g \Rightarrow f$ d.a. g y g d.a. f entonces por definición $f \equiv g$ y $\therefore$ la relación es simétrica.
(ii) La relación es reflexiva de Teorema 1.
(iii) Si $f \equiv g$ y $g \equiv h$, entonces f d.a. g y g d.a. h, por lo que f d.a. h. Además h d.a. g y g. d.a. $f \Rightarrow f \equiv h$, $\therefore$ la relación es transitiva.

Notación Θ

Para una función dada $g(n)$:

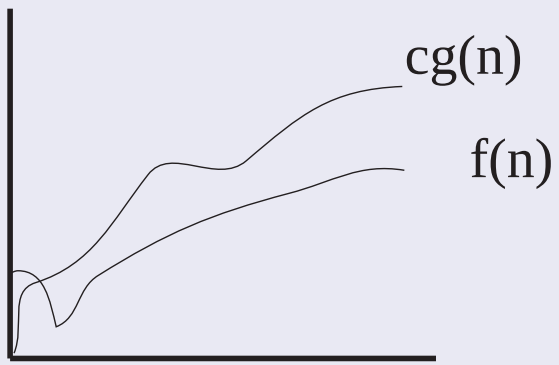
$$\Theta(g(n)) = \{f(n) : \text{existen las constantes positivas } c_1,\ c_2 \text{ y } n_0 \text{ tal que } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{para todo } n \geq n_0\}$$



Notación O

Para una función dada $g(n)$:

$$O(g(n)) = \{f(n) : \text{existen las constantes positivas } c \text{ y } n_0 \text{ tal que } 0 \leq f(n) \leq cg(n) \text{para todo } n \geq n_0\}$$



Notación *o*

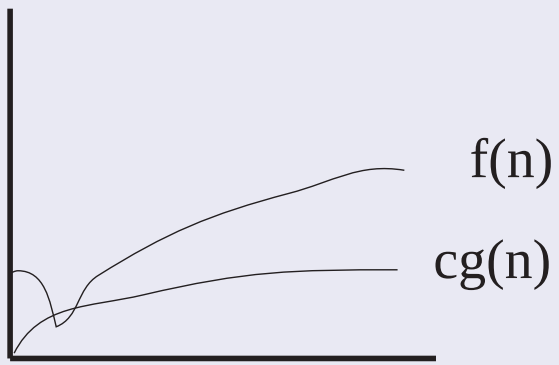
A diferencia de la notiación O donde la cota superior puede ser ajustada o no, la notación o denota una cota superior no ajustada. Para una función dada $g(n)$:

$$o(g(n)) = \{f(n) : \text{para cualquier constante positivas } c \text{ , existe un } n_0 \text{ tal que } 0 \leq f(n) \leq cg(n) \text{para todo } n \geq n_0\}$$

Notación Ω

Para una función dada $g(n)$:

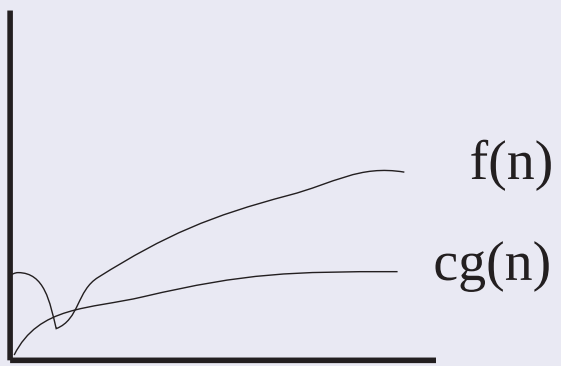
$$\Omega(g(n)) = \{f(n) : \text{existen las constantes positivas } c \text{ y } n_0 \text{ tal que } 0 \leq cg(n) \leq f(n) \text{para todo } n \geq n_0\}$$



Notación ω

Análogo a lo que sucede entre O y o, la notación w es una cota inferior no ajustada. Para una función dada $g(n)$:

$$\begin{aligned}\omega(g(n)) \quad = \quad & \{f(n) : \text{para cualquier constantes positivas } c \text{ y } n_0 \text{ tal que} \\ & 0 \leq cg(n) \leq f(n) \text{para todo } n \geq n_0\}\end{aligned}$$



Ejemplos de O

$$3n^2 - 100n + 6 = O(n2) \text{ porque } 3n^2 > 3n^2 - 100n + 6$$
$$3n^2 - 100n + 6 = O(n3) \text{ porque } ,01n^3 > 3n^2 - 100n + 6$$
$$3n^2 - 100n + 66 \neq O(n) \text{ porque } c * n < 3n^2 \text{ cuando } n > c$$

Se debe pensar en igualdad como en el conjunto de funciones

Ejemplos de Ω

$$3n^2 - 100n + 6 = \Omega(n2) \text{ porque } 2 : 99n^2 < 3n^2 - 100n + 6$$
$$3n^2 - 100n + 6 \neq \Omega(n3) \text{ porque } 3n^2 - 100n + 6 < n^3$$
$$3n^2 - 100n + 66 = \Omega(n) \text{ porque } 10^{10^{10}} n < 3n^2 - 100 + 6$$

Ejemplos de Θ

$$3n^2 - 100n + 6 = \Theta(n2) \text{ porque se cumple } O, \Omega$$
$$3n^2 - 100n + 6 \neq \Theta(n3) \text{ porque se cumple sólo } O$$
$$3n^2 - 100n + 66 \neq \Theta(n) \text{ porque se cumple sólo } \Omega$$

Teorema 3

Sean $f, g, h \in M$, entonces: (a) f es $O(f)$.
(b) Si f es $O(g) \Rightarrow cf$ es $O(g)$ para un $c \in \mathbb{R}$. Es decir, el conjunto $O(g)$ es cerrado bajo la manipulación por una constante.
(c) Si f es $O(g)$ y h es $O(g) \Rightarrow (f + h)$ es $O(g)$ donde $(f + g)(n) = f(n) + g(n)$ i.e. $O(g)$ es cerrado bajo la suma de funciones.

Teorema 3: demostración

Sean $f, g, h \in M$, entonces:
(a) f es $O(f)$; dado que f d.a. $f \Rightarrow f$ es $O(f)$.

(b) Debemos demostrar que cf es $O(g) \Rightarrow \exists m, k \geq 0$ tales que $cf(n) \leq mg(n), \forall n \geq k$, con $c \in \mathbb{R}$. Si $m = cm_1$

$$\begin{aligned} cf(n) &\leq cm_1 g(n), \forall n \geq k_1 \\ f(n) &\leq m_1 g(n), \forall n \geq k_1 \end{aligned}$$

Entonces $m = |c|m_1$ y $k = k_1$ la desigualdad $f(n) \leq m_1 g(n)$ se cumple y en consecuencia cf es $O(g)$.

Teorema 3: demostración (cont.)

(c) Si f es $O(g)$ y h es $O(g)$ entonces $(f + h)$ es $O(g)$.
Sabemos que existen $m_1, m_2, k_1, k_2 \geq 0$ tales que
$f(n) \leq m_1 g(n), \forall n \geq k_1$ y que $h(n) \leq m_2 g(n), \forall n \geq k_2$.

Sumando las desigualdades término a término:
$f(n) + h(n) \leq m_1 g(n) + m_2 g(n), \forall n \geq max(k_1, k_2)$. Entonces:
$f(n) + h(n) \leq (m_1 + m_2) g(n), \forall n \geq max(k_1, k_2)$.

Com $m = m_1 + m_2$ y $k = max(k_1, k_2)$ se cumple que g d.a. $f(n) + h(n)$.

Teorema 4

Sean $f, g \in M$, f es $O(g) \Leftrightarrow O(f) \subset O(g)$.

Dem.

Debemos demostrar: $\Rightarrow$ Si f es $O(g)$ entonces $O(f) \subset O(g)$. Si f es $O(g)$, por defínción g d.a. f. Sea h una función arbitraria tal que f d.a. h, por transitividad g d.a h. y luego entonces h es $O(g)$. Dado que h puede ser cualquier elemento de $O(f)$, podemos concluir que $O(f) \subset O(g)$.

$\Leftarrow$ Si $f \subset O(g)$ entonces f es $O(g)$. f es $O(f)$ y como $O(f) \subset O(g) \Rightarrow f$ es $O(g)$.

Teorema 5

Sean $f, g, h_1, h_2 \in M$. Si f es $O(h_1)$ y g es $O(h_2)$ entonces $f + g$ es $O(max(h_1, h_2))$, donde una función es mayor que otra si la domina asintótcamente.

Dem. Si f es $O(h_1)$ y g es $O(h_2)$ entonces
$f(n) + g(n) \leq m_1 h_1(n) + m_2 h_2(n), \forall n \geq max(k_1, k_2)$.
Por el teorema 3 y sustituyendo,
$f(n) + g(n) \leq (m_1 + m_2) max(h_1(n), h_2(n)), \forall n \geq max(k_1, k_2)$.

Si $h(n) = max(h_1(n), h_2(n))$ y $k = max(k_1, k_2)$, entonces $(f + g)$ es $O(max(h_1, h_2)$.

Teorema 6

Sean $c, d \in \mathbb{R}$ y $0 < c < d$, entonces $O(n^c) \subset O(n^d)$, y la inclusión es propia.

Como consecuencia de este teorema, *para determinar el grado de un polinomio se elige el término de mayor grado o* para estimar el orden de una expresión asintótica se usa el término más representativo.

Propiedades Notaciones

- $\Theta(g(n)) \subseteq O(g(n))$
- $\Theta(g(n)) \subseteq \Omega(g(n))$
- *Transitividad:* $(f(n) \in O(g(n)) \wedge g(n) \in O(h(n))) \rightarrow f(n) \in O(h(n))$ (esto para todas las notaciones)
- *Reflexividad:* $f(n) \in O(f(n))$ (para notaciones O, Ω y Θ
- *Simetría :* $f(n) \in \Theta(g(n))$ iff $g(n) \in \Theta(f(n))$
- *Simetría Transpuesta:* $f(n) \in O(g(n))$ iff $g(n) \in \Omega(f(n))$ y $f(n) \in o(g(n))$ iff $g(n) \in \omega(f(n))$

Tipos de Complejidad

La mayoría de los algoritmos caen en alguno de los siguientes casos:

$O(1)$	Complejidad constante
$O(logn)$	Complejidad logarítmica
$O(n)$	Complejidad lineal
$O(nlogn)$	Complejidad "n log n"
$O(n^2)$	Complejidad cuadrática
$O(c^n)$, $c > 1$	Complejidad exponencial
$O(n!)$	Complejidad factorial

Estas complejidades tienen una relación de orden dada por:
$O(1) \subset O(logn) \subset O(n) \subset O(nlognn) \subset O(n^2) \subset O(c^n) \subset O(n!)$

Funciones dominantes

Las siguiente tabla muestra razones de crecimiento de funciones comunes medidas en nanosegundos (μs).

n $f(n)$	lgn	n	$nlgn$	n^2	2^n	$n!$
10	$0.003\mu s$	$0.01\mu s$	$0.033\mu s$	$0.1\mu s$	$1\mu s$	3.63 ms
20	$0.004\mu s$	$0.02\mu s$	$0.086\mu s$	$0.4\mu s$	$1ms$	77.1 años
30	$0.005\mu s$	$0.03\mu s$	$0.147\mu s$	$0.9\mu s$	$1s$	$8{,}4x10^{15}$ años
100,000	0.017 μs	0.1 ms	167 ms	16 s		

Conjunto de funciones de $\mathbb{N}$ a $\mathbb{R}$