Análisis y Diseño de Algoritmos

Tema 1: Algoritmos y Complejidad

Titulación: Grado en Ingeniería del Software

UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA

Contexto de la asignatura

Primer Curso: Toma de contacto con el computador

- Conocimiento de la herramienta: electrónica, tecnología de Computadores
- Comunicación con la misma: lenguajes y fundamentos de programación.

Segundo Curso: Formalizar y Sistematizar

- Algoritmos eficaces (correctos): Verificación formal 2
- Algoritmos eficientes:
  - Medida de eficiencia: Complejidad del algoritmo 1
- Técnicas de diseño de algoritmos: 3
  - Divide y Vencerás
  - Algoritmos voraces
  - Programación Dinámica
  - Vuelta a atrás.
  - Ramificación y Poda.

Contenido

- Complejidad
  - Coste computacional de un algoritmo
  - Clases de complejidad y notación asintótica
  - Análisis de algoritmos no recursivos: algoritmos de ordenación
  - Resolución de ecuaciones de recurrencia
  - Análisis de Algoritmos Recursivos
  - El teorema maestro

Calcular 2^{16}

- Método directo: 16 multiplicaciones

$$2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2$$

```
x= 1;
for (int i = 1; i<=16){ x = x * 2;}
```

- ¿Se podría hacer sólo con 4 multiplicaciones?

$$2 \cdot 2 = 2^2$$

$$2^2 \cdot 2^2 = 2^4$$

$$2^4 \cdot 2^4 = 2^8$$

$$2^8 \cdot 2^8 = 2^{16}$$

```
x= 2;
for (int i = 1; i<=4){ x = x * x;}
```

Calcular 2^n

- Si n = 32

  Método directo:

  32 multiplicaciones

  Método eficiente:

  5 multiplicaciones

  $2 \cdot 2 = 2^2$

  $2^2 \cdot 2^2 = 2^4$

  $2^4 \cdot 2^4 = 2^8$

  $2^8 \cdot 2^8 = 2^{16}$

  $2^{16} \cdot 2^{16} = 2^{32}$

- Si n = 64

  Método directo:

  64 multiplicaciones

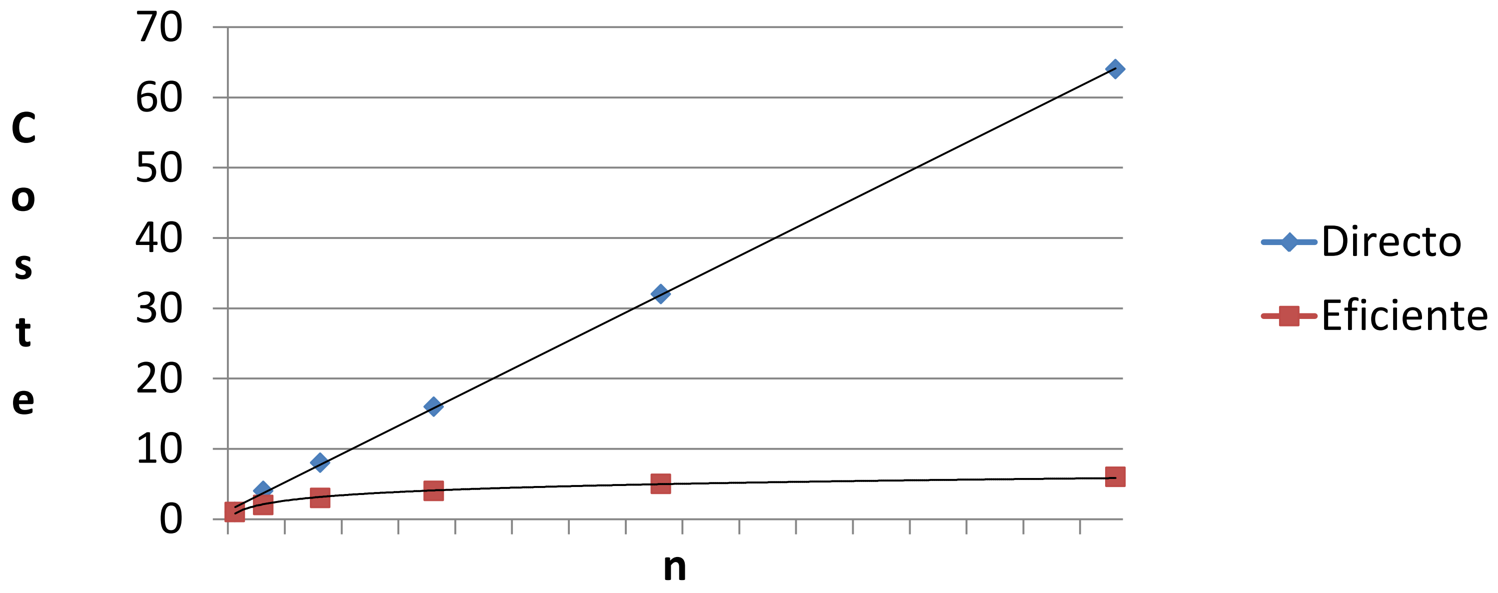
  Método eficiente:

  6 multiplicaciones

- ¿Hay alguna relación entre el coste de los dos métodos?

  $CosteEficiente = \log_2(CosteDirecto)$

Análisis coste 2^n



- Al crecer el tamaño del problema, el coste computacional del primer algoritmo crece mucho más rápido que el del segundo.
- La complejidad del primer algoritmo parece mayor que la del segundo.

REGLAS PRÁCTICAS PARA HALLAR EL COSTE DE UN ALGORITMO

- Las instrucciones simples
- La composición de instrucciones
- Las instrucciones de selección
- Los bucles
- Los subprogramas

INSTRUCCIONES SIMPLES

- Se considera que se ejecuta en tiempo constante (*k* segundos)*:*
  - La evaluación de las expresiones aritméticas siempre que los datos sean de tamaño constante así como las comparaciones de datos simples.
  - Las operaciones de asignación, lectura y escritura de datos simples.
  - Las operaciones de acceso a una componente de un array, a un campo de un registro y a la siguiente posición de un registro de un archivo.

- Todas estas operaciones se consideran operaciones elementales de coste 1.

COMPOSICIÓN DE INSTRUCCIONES

Si suponemos que las instrucciones I_1 y I_2 poseen coste computacional (complejidad temporal), en el peor de los casos, de $T_1(n)$ y $T_2(n)$ respectivamente, entonces el coste de la composición de ambas instrucciones será:

$$T_{I1;I2}(n) = T_1(n) + T_2(n)$$

INSTRUCCIONES DE SELECCIÓN

– if <condición> then I_1 else I_2

$$T_{selección}(n)=T_{condición}(n)+max(T_1(n),T_2(n))$$

– Case <expresión> of

caso1: I_1;
caso2: I_2;
.....
.....
cason: I_n;
end; {case}

$$T_{selección}(n)=T_{expresión}(n)+max(T_1(n),....,T_n(n))$$

BUCLES CON ITERACIONES FIJAS

En los bucles con contador explícito, se distinguen dos casos: que el tamaño del problema *n* forme parte de los límites o que no.

Si el bucle se realiza un número fijo de veces, *K*, independiente de *n*, entonces la repetición sólo introduce una constante multiplicativa.

Ejemplo 1.- `for (int i= 0; i < K; i++) { bloque con coste T }`

Entonces el coste computacional es

$$Coste(\text{"int i=0"}) +$$

Inicialización variable control

$$\left(\sum_{i=0}^{K-1} Coste(\text{"i<K"}) + \mathrm{Coste}(\text{"i ++"}) + \mathrm{T}\right) +$$

Coste iteraciones

$$\mathrm{Coste}(\text{"i<K"}) =$$

Última evaluación de i<K

$$1 + \left(\sum_{i=0}^{K-1} 1 + 2 + T\right) + 1 = 1 + (3 + T) \cdot K + 1 = Constante$$

BUCLE CON ITERACIONES FIJAS

Si el tamaño *n* aparece como límite de iteraciones ...

Ejemplo 2.-

```
for (int i= 0; i < n; i++) { bloque con coste T }
```

$$\mathrm{Coste(n)} = 1 + \left(\sum_{i=0}^{n-1} 1 + 2 + T\right) + 1 = 1 + (3 + T) \cdot n + 1$$

$$\approx T \cdot n$$

BUCLE CON ITERACIONES FIJAS

Si el tamaño *n* aparece como límite de iteraciones ...

Ejemplo 3.

```
for (int i= 0; i < n; i++) {
    for (int j= 0; j < n; j++) {
        bloque con coste T
    }
}
```

$$Coste(n) = 1 + \left[\sum_{i=0}^{n-1} 1 + 2 + \left(1 + \left(\sum_{j=0}^{n-1} 1 + 2 + T \right) + 1 \right) \right] + 1 =$$

$$2 + \left[\sum_{i=0}^{n-1} 1 + 2 + (1 + (1 + 2 + T)n + 1) \right] =$$

$$2 + \left[\sum_{i=0}^{n-1} 5 + (3 + T)n \right] = 2 + [5 + (3 + T)n] \cdot n \approx Tn^2$$

BUCLES CON ITERACIONES VARIABLES

Ejemplo 4.-

```
for (int i= 0; i < n; i++) {
    for (int j= 0; j < i; j++) {
        bloque de coste T
    }
}
```

el bucle exterior se realiza *n* veces, mientras que el interior se realiza 0, 1, 2 ... n-1 veces respectivamente.

$$Coste(n) = 1 + \left[\sum_{i=0}^{n-1} 1 + 2 + \left(1 + \left(\sum_{j=0}^{i-1} 1 + 2 + T \right) + 1 \right) \right] + 1$$

$$= 2 + \left[\sum_{i=0}^{n-1} 5 + (3+T) \cdot i \right] = 2 + \sum_{i=0}^{n-1} 5 + \sum_{i=0}^{n-1} (3+T) \cdot i$$

$$= 2 + 8n + (3+T) \frac{(n-1)(n)}{2} \approx \frac{T}{2} n^2$$

INSTRUCCIONES DE ITERACIÓN: BUCLES

Bucle con iteraciones variables:

A veces aparecen bucles multiplicativos, donde la evolución de la variable de control no es lineal (como en los casos anteriores)

Ejemplo 5.-

```
c= 1;
while (c < n) {
        bloque de coste T
        c= 2*c;
}
```

El valor inicial de "c" es 1, siendo "2^k" al cabo de "k" iteraciones. El número de iteraciones k es tal que $2^k >= n \Rightarrow k = eis\ (log_2\ (n))$ [entero inmediato superior]

$Coste(n) = 1 + (1 + T + 2) \cdot eis(log_2\ (n)) + 1 \approx T \cdot log_2(n)$

Ejercicio 1

Ejercicio 1.- Dado el siguiente segmento de código

```
c= n;
while (c > 1) {
        bloque de coste T
        c= c / 2;
 }
```

Explicar de forma justificada como evoluciona la variable c, cuantas veces se ejecuta el bucle y el coste computacional de ese conjunto de instrucciones.

Ejercicio 2

Ejercicio 2.- Dado el código siguiente:

```
for (int i= 0; i < n; i++) {
      c= n;
      while (c > 1) {
          bloque de coste T
           c= c/2;
      }
  }
```

tenemos un bucle interno de cuya complejidad se ha calculado en el ejercicio anterior. Se pide:

a) Determinar el coste computacional de este código

b) Determinar el coste computacional al cambiar “c=n;” por “c=i;”

Complejidad: tamaño de la entrada

- Dado un algoritmo *A*, la complejidad se establece como una función que asocia a cada entrada del algoritmo su tiempo de ejecución.

Tamaño de la entrada

$$T_A : \mathbb{N} \rightarrow \mathbb{N}$$

Tiempo de ejecución

- Una cuestión importante es cómo medimos la entrada de un algoritmo
  - Habitualmente, cuanto más grande es la entrada, el algoritmo tarda más en ejecutarse.
- En muchos casos es fácil decidir cómo medir la entrada de un algoritmo, por ejemplo:
  - Ordenación o búsqueda en una lista: longitud de la lista
  - Evaluación de un polinomio en un punto: grado del polinomio (los coeficiente no afectan de forma determinante a la complejidad)
- En otros casos, la decisión no es tan fácil
  - Multiplicación de matrices cuadradas de tamaño *nxn*:
    - La dimensión *n* de la matriz
    - El número de elementos de la matriz *n**n*
- De forma general puede utilizarse el número de bits de la representación binaria de la entrada
- En otras ocasiones puede ser que la función de la complejidad dependa de más de un parámetro de entrada.

Complejidad: órdenes de crecimiento

	complejidad logarítmica	complejidad lineal	complejidad cuadrática	Complejidad exponencial
n	$log_2 n$ ms	n ms	n^2 ms	2^n ms
10	0.000003 s	0.00001 s	0.0001 s	0.001 s
100	0.000007 s	0.0001 s	0.001 s	10^{14} ???
1000	0.00001 s	0.001 s	1 s	astronómica
10000	0.000013 s	0.01 s	1.7 min	astronómica
100000	0.000017 s	0.1 s	2.8 h	astronómica

La base de los logaritmos y de las exponenciales no es relevante

Los algoritmos con una complejidad exponencial sólo son prácticos para resolver problemas de pequeño tamaño

Principio de invarianza

Dos implementaciones de un mismo algoritmo no diferirán más que en una constante multiplicativa.

Si $T_1(n)$ y $T_2(n)$ son los tiempos consumidos por dos implementaciones de un mismo algoritmo, se verifica que:

$$\exists\, c, d \in \mathbf{R},$$

$$T_1(n) \leq c \cdot T_2(n)$$

$$T_2(n) \leq d \cdot T_1(n)$$

Clases de complejidad y notación asintótica: *O(g(n))*

Definición

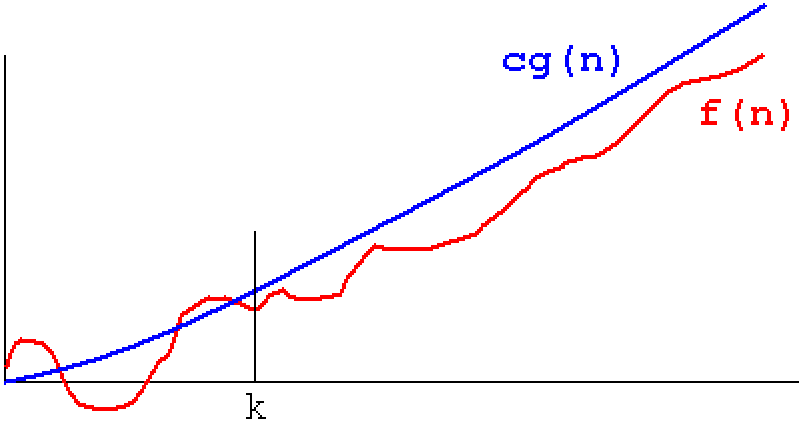
$$O(g(n)) = \{f : \mathbb{N} \to \mathbb{N} | \exists k \geq 0, c > 0, \forall n \geq k. f(n) \leq cg(n)\}$$

- Informalmente, *O(g(n))* es el conjunto de funciones acotadas superiormente por un múltiplo de *g*.
- Se utiliza para probar que la complejidad de un algoritmo como muy mal se va a comportar como la función *g*, que se toma como referencia.
- Observa que los valores iniciales de ambas funciones no importan, lo que es relevante es que a partir de una cierto número, *f* se comporte mejor o igual que un múltiplo de *f*.

Clases de complejidad y notación asintótica: *O(g(n))*

Definición

$$O(g(n)) = \{f : \mathbb{N} \to \mathbb{N} | \exists k \geq 0, c > 0, \forall n \geq k . f(n) \leq cg(n)\}$$



Ejemplo Por ejemplo

$$n \in O(n^2) \qquad 100n + 5 \in O(n^2) \qquad \frac{1}{2}n(n-1) \in O(n^2)$$

$$n^3 \notin O(n^2) \quad 0.00001n^3 \notin O(n^2) \quad n^4 + n + 1 \notin O(n^2)$$

Clases de complejidad y notación asintótica: $\Omega(g(n))$

Definición

$$\Omega(g(n)) = \{f : \mathbb{N} \rightarrow \mathbb{N} | \exists k \geq 0, c > 0, \forall n \geq k. f(n) \geq cg(n)\}$$

- Informalmente, $\Omega(g(n))$ es el conjunto de funciones acotadas inferiormente por un múltiplo de *g*.

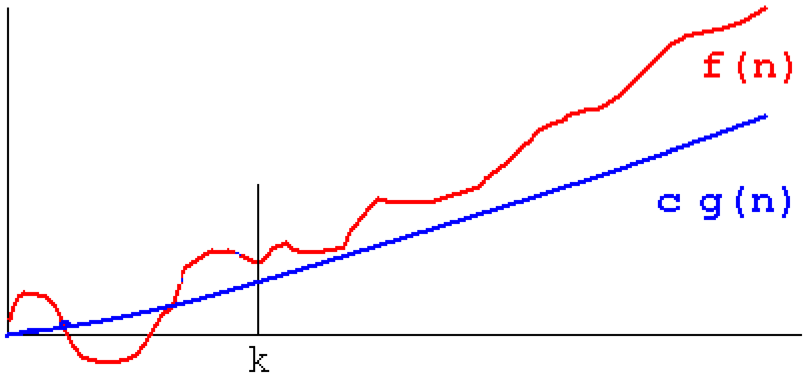
- Se utiliza para probar que la complejidad de un algoritmo como muy bien se va a comportar como la función *g*, que se toma como referencia.

- Como en el caso de *O*, los valores iniciales de ambas funciones no importan, lo que es relevante es que a partir de una cierto número, *f* se comporte peor o igual que un múltiplo de *g*.

Clases de complejidad y notación asintótica: *Ω(g(n))*

Definición

$$\Omega(g(n)) = \{f : \mathbb{N} \to \mathbb{N} | \exists k \geq 0, c > 0, \forall n \geq k. f(n) \geq cg(n)\}$$



Ejemplo Por ejemplo

$$n^3 \in \Omega(n^2) \quad 100n + 5 \in \Omega(n) \quad \tfrac{1}{2}n(n-1) \in \Omega(n^2)$$

$$n \notin \Omega(n^2) \quad 0.01n^2 \in \Omega(n) \quad n^2 + n + 1 \notin \Omega(n^3)$$

Clases de complejidad y notación asintótica: *Θ(g(n))*

Definición

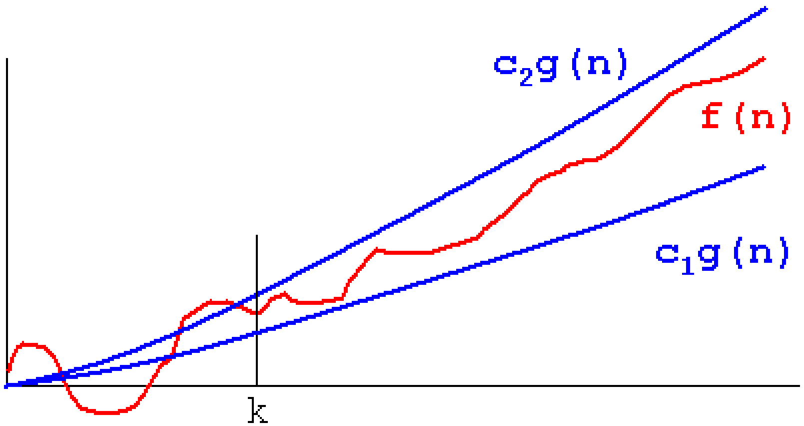
$$\Theta(g(n)) = \{f : \mathbb{N} \to \mathbb{N} | \exists k \geq 0, c_1 > 0, c_2 > 0 \forall n \geq k. c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$

- Informalmente, *Θ(g(n))* es el conjunto de funciones con el mismo orden de complejidad que g.
- Se utiliza para probar que la complejidad de un algoritmo es igual asintóticamente a la función *g*, que se toma como referencia.
- Como en los dos casos anteriores, los valores iniciales de ambas funciones no importan, lo que es relevante es que a partir de una cierto número, *f* crezca de forma *similar* a *g*.

Clases de complejidad y notación asintótica: $\Theta(g(n))$

Definición

$$\Theta(g(n)) = \{f : \mathbb{N} \to \mathbb{N} | \exists k \geq 0, c_1 > 0, c_2 > 0 \forall n \geq k. c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$



Ejemplo Por ejemplo

$$100n^2 \in \Theta(n^2) \quad 100n + 5 \in \Theta(n) \quad \tfrac{1}{2}n(n-1) \in \Theta(n^2)$$

$$n \notin \Theta(n^2) \quad 0.01n \notin \Theta(n^2) \quad n^2 + n + 1 \notin \Theta(n^3)$$

Principio de invarianza

Dos implementaciones de un mismo algoritmo no diferirán más que en una constante multiplicativa.

Si $T_1(n)$ y $T_2(n)$ son los tiempos consumidos por dos implementaciones de un mismo algoritmo, se verifica que:

$$T_1(n) \in \Theta(T_2(n))$$

$$T_2(n) \in \Theta(T_1(n))$$

Notación Asintótica: propiedades

Si $f_1(n) \in O(g_1(n))$ y $f_2(n) \in O(g_2(n))$ entonces $f_1(n)+f_2(n) \in O(max(g_1(n), g_2(n)))$

Ejemplo $\frac{1}{2}n(n+1) \in O(max(n^2, n)) = O(n^2)$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \begin{cases} < \infty & \Leftrightarrow f(n) \in O(g(n)) \\ 0 < c < \infty & \Leftrightarrow f(n) \in \Theta(g(n)) \\ > 0 & \Leftrightarrow f(n) \in \Omega(g(n)) \end{cases}$$

Ejemplo

$$\lim_{n\to\infty} \frac{1}{n^2} = 0 \Rightarrow 1 \in O(n^2)$$

$$\lim_{n\to\infty} \frac{2n}{n} = 2 > 0 \Rightarrow 2n \in \Theta(n)$$

$$\lim_{n\to\infty} \frac{2n^2}{n} = \infty \Rightarrow 2n^2 \in \Omega(n)$$

Ejemplo $$\lim_{n\to\infty} \frac{\frac{1}{2}n(n-1)}{n^2} = \frac{1}{2} \lim_{n\to\infty} \frac{n^2-n}{n^2} = \frac{1}{2} \lim_{n\to\infty} (1 - \frac{1}{n}) = \frac{1}{2}$$

Notación asintótica: propiedades

$O(\log_a n) = O(\log_b n)$

Ya que, por la propiedad de cambio de base de los logaritmos

$$\log_a n = \frac{\log_b n}{\log_b a} = \frac{1}{\log_b a}\log_b n = Cte \cdot \log_b n$$

Por esta razón no es necesario especificar la base del logaritmo: $O(\log n)$.

Notación asintótica: propiedades

Los comportamientos asintóticos de más frecuente aparición se pueden ordenar de menor a mayor crecimiento de la siguiente forma:

$1 \ll \log n \ll n \ll n \cdot \log n \ll n^2 \ll n^3 \ll \ldots \ll 2^n \ll n!$

Complejidad logarítmica

- Sabemos que $n^0 = 1 \ll \log n \ll n = n^1$
- ¿Cómo podríamos calcular lo cerca que está el orden *log n* del orden *n*?

$$\lim_{n\to\infty}\frac{n^{0.0625}}{\log n} = \lim_{n\to\infty}\frac{0.0625\cdot n^{-0.9375}}{1/n} = \lim_{n\to\infty}\frac{0.0625\cdot n}{n^{0.9375}} = \lim_{n\to\infty} 0.0625\cdot n^{0.125} = \infty$$

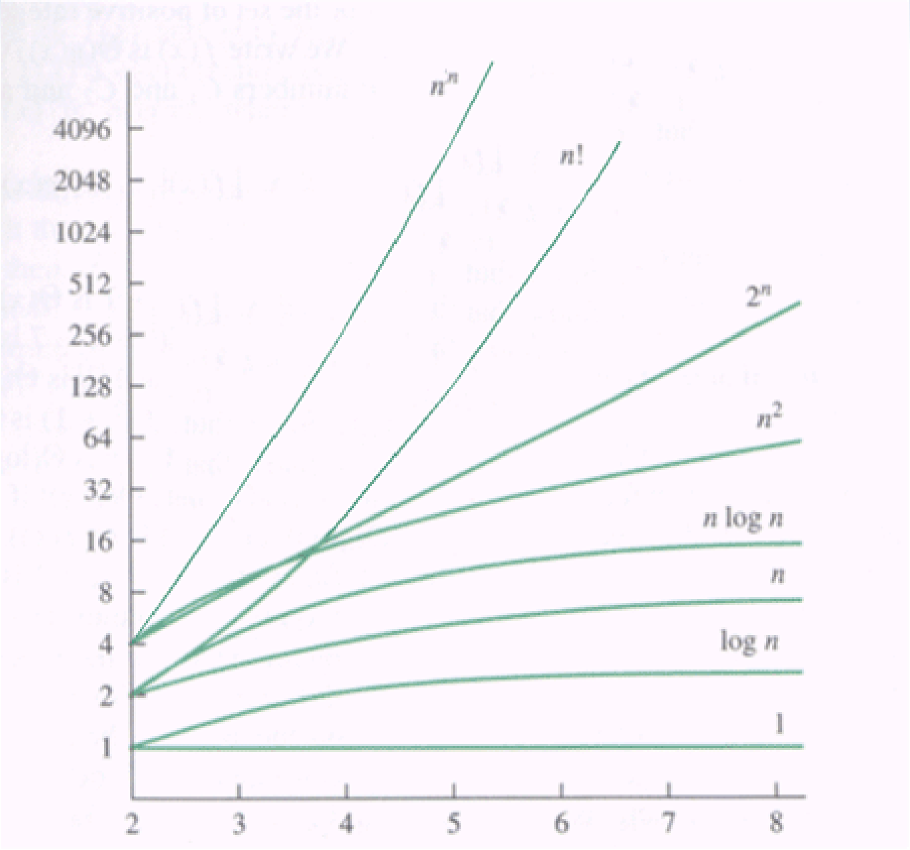
Complejidad logarítmica

- En general, para 0 < k

$$\lim_{n\to\infty}\frac{n^k}{\log n}=\lim_{n\to\infty}\frac{k\cdot n^{k-1}}{1/n}=\lim_{n\to\infty}\frac{k\cdot n^{k-1}}{n^{-1}}=\lim_{n\to\infty}k\cdot n^{k-1-(-1)}=\infty$$

- El comportamiento asintótico de *log n* se puede considerar casi constante.

Complejidad: comparando las complejidades



Comparativa de complejidades

O(1)	Constante	No depende del tamaño del problema	Eficiente
O(log n)	Logarítmica	Búsqueda binaria	
O(n)	Lineal	Búsqueda lineal	
O(n•log n)	Casi lineal	Quick-sort	
O(n^2)	Cuadrática	Algoritmo de la burbuja	Tratable
O(n^3)	Cúbica	Producto de matrices	
O(n^k) k>3	Polinómica		
O(k^n) k>1	Exponencial	Algunos algoritmos de grafos	Intratable
O(n!)	Factorial		

Caso de estudio

- A veces al calcular la complejidad resulta:

$$T(n) = 4^{log_3 n}$$

- ¿Es una complejidad exponencial?
- No
- Porque $4^{log_3 n} = n^{log_3 4} \approx n^{1,26}$
- Propiedad de los logaritmos:

$$a^{log_c b} = b^{log_c a}$$

Caso de estudio

- Demostración:

$$a^{\log_c b} = z; \quad [1]$$

$$\log_c\left(a^{\log_c b}\right) = \log_c z \quad ; \quad \log_c b \cdot \log_c a = \log_c z \quad ;$$

$$\log_c\left(b^{\log_c a}\right) = \log_c z \quad ; \quad \text{(Por inyectividad)}$$

$$b^{\log_c a} = z; \quad [2]$$

$$[1] \wedge [2] \Rightarrow a^{\log_c b} = b^{\log_c a}$$