

Análisis de eficiencia - Tiempo de ejecución

Estructuras de Datos y Algoritmos /
Algoritmos y Estructuras de Datos II

Año 2025

Dr. Pablo Ponzio

Universidad Nacional de Río Cuarto
CONICET



CREER... CREAR... CRECER



Atributos de calidad de los programas

- Corrección (resp. de la especificación)
- Eficiencia en el uso de recursos computacionales
 - Eficiencia en tiempo de ejecución
 - Eficiencia en uso de memoria
- Mantenibilidad
 - Extensibilidad y reparabilidad
- Reusabilidad
- Comprensibilidad

Atributos de calidad de los programas

- Corrección (resp. de la especificación)
- Eficiencia en el uso de recursos computacionales
 - Eficiencia en tiempo de ejecución
 - Eficiencia en uso de memoria
- Mantenibilidad
 - Extensibilidad y reparabilidad
- Reusabilidad
-

En esta clase nos enfocaremos en la eficiencia en tiempo de ejecución

Importancia de la Eficiencia

- Aunque los procesadores son cada vez más rápidos, la eficiencia es importante porque:
 - Existen soluciones a problemas simples que son extremadamente lentas (para cualquier procesador)
 - Los algoritmos eficientes nos permiten resolver más problemas en menos tiempo
 - La tesis extendida de Church-Turing: El avance en hardware es solo polinomial
 - Mientras que la cantidad de operaciones que tiene que realizar un algoritmo puede crecer exponencialmente

Un ejemplo

- Supongamos que queremos calcular 2^n
- El siguiente programa es correcto pero ineficiente

```
public static long exp(long n){  
    if (n == 0)  
        return 1;  
    else  
        return exp(n-1) + exp(n-1);  
}
```

- Es demasiado lento incluso para valores relativamente chicos de n
- Este es un ejemplo de un programa exponencial
 - La cantidad de operaciones que realiza crece exponencialmente respecto del tamaño de la entrada (n)

Midiendo el tiempo de ejecución

- El tiempo de ejecución de un programa depende de muchos factores:
 - La entrada del programa
 - La calidad del código generado por el compilador
 - La naturaleza y la velocidad de las instrucciones de la máquina usada para ejecutar el programa
 - La complejidad en tiempo del algoritmo implementado
- Una alternativa sería realizar un análisis experimental: Correr el programa y medir la cantidad de tiempo que tarda para distintas entradas
 - Esto es dependiente del compilador, la computadora en la que se ejecuta, etc...

Midiendo el tiempo de ejecución

- Otra alternativa es calcular la cantidad de instrucciones que ejecutará el programa en una computadora idealizada
 - $T(n)$: cantidad de instrucciones que ejecuta el programa para una entrada de tamaño n
 - Esto nos permitirá sacar conclusiones del estilo "el tiempo de ejecución del algoritmo es proporcional a n^2 "
 - Donde la constante de proporcionalidad depende de los factores restantes (el compilador, la computadora, etc...)
 - Nos da una forma comparar la eficiencia de distintos algoritmos que es muy útil en la práctica

Análisis del peor caso

- Para muchos programas, el tiempo de ejecución no es sólo una función del tamaño de la entrada, sino que depende de la entrada particular
- En muchos casos, estaremos interesados en el tiempo de ejecución del programa en el peor caso
 - El tiempo que tarda el programa para la(s) entrada(s) en la que la ejecución es más lenta
- El análisis del peor caso nos da una cota superior para la eficiencia del programa
 - Tener en cuenta que en algunos casos este análisis puede no reflejar el tiempo de ejecución real
 - Por ejemplo, si las entradas del peor caso rara vez ocurren en la práctica

Análisis del peor caso

- Para hacer el análisis del peor caso debemos:
 - Determinar el tamaño de las entradas del programa
 - Suele ser bastante directo: el tamaño de una lista/arreglo/ etc..., un entero, etc...
 - Determinar el peor caso del programa: Analizar el código para ver cuáles son las entradas que ejecutan más instrucciones
 - Determinar cuáles son las operaciones básicas del programa
 - Usualmente, asignaciones, operaciones aritméticas y booleanas, acceso a elementos de arreglos, etc...
 - Si bien estas instrucciones pueden tomar distinto tiempo en la práctica, nos abstraemos de estos detalles y las consideramos constantes

Análisis del caso promedio

- Otra posibilidad es considerar el tiempo de ejecución del algoritmo en el caso promedio
- En principio parece una medida más práctica, pero tiene sus problemas
 - Es mucho más difícil de calcular, el análisis se vuelve matemáticamente intratable rápidamente
 - Puede ser erróneo asumir que todas las entradas son igualmente probables
 - La noción de promedio no siempre tiene un significado obvio

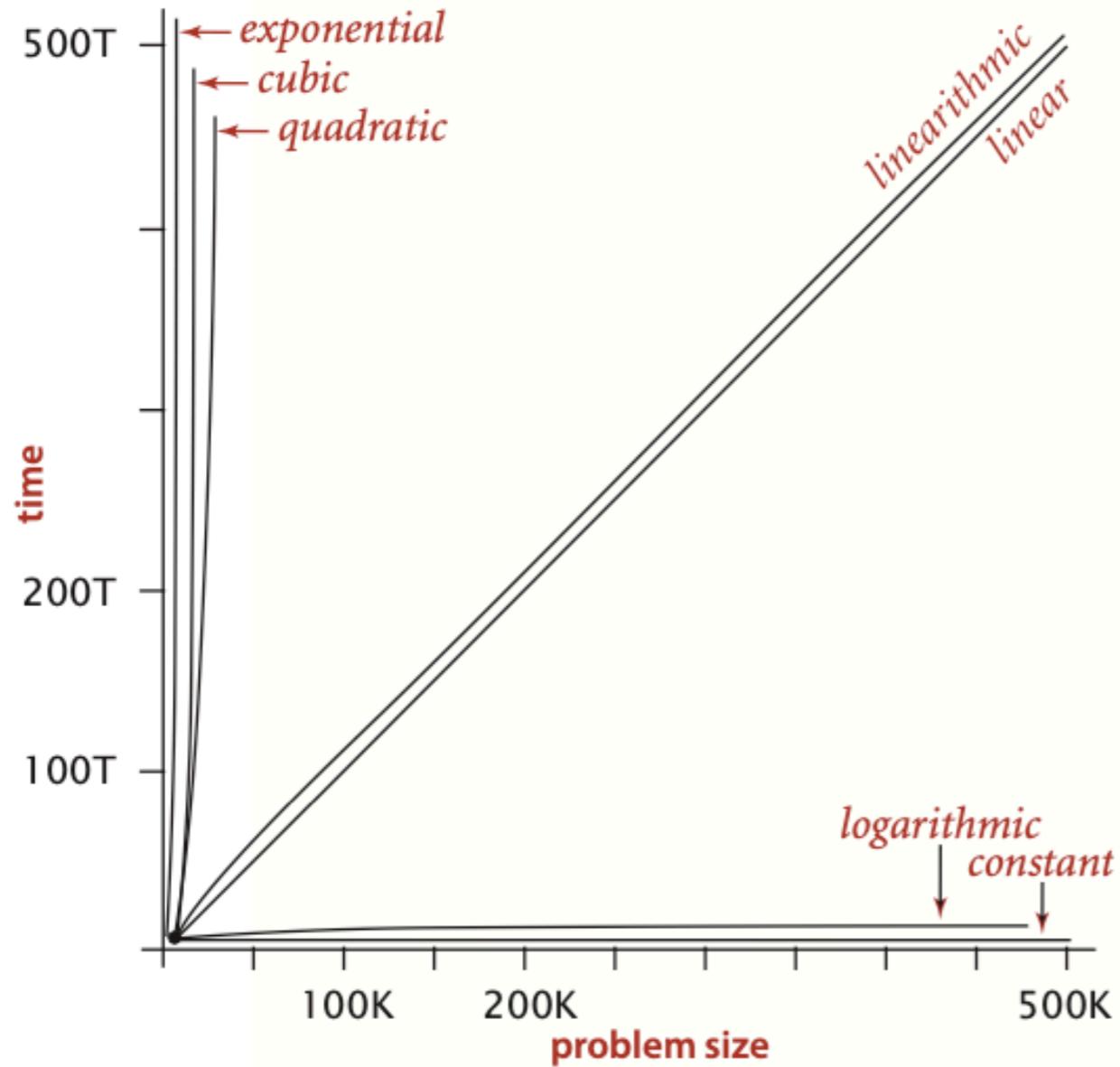
Función de crecimiento

- Dado un programa, definimos su función de crecimiento como:
 - $T(N) : \mathbb{N} \rightarrow \mathbb{R}^+$
- Esta función refleja el tiempo de ejecución del programa con respecto al tamaño de la entrada N
- Su definición depende del programa a analizar
- Suponemos que las funciones son monótonas
 - $n \leq m \implies T(n) \leq T(m)$

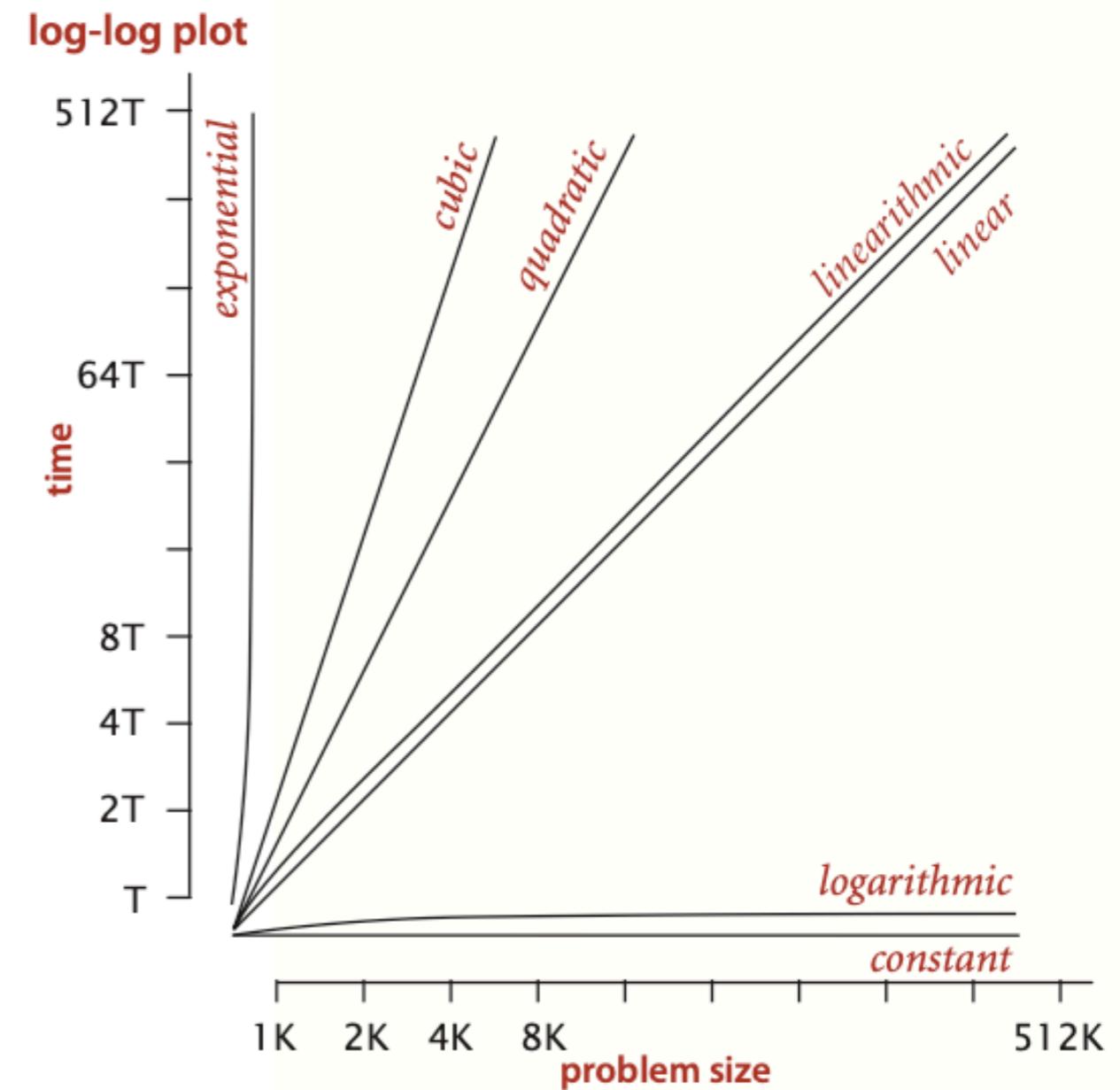
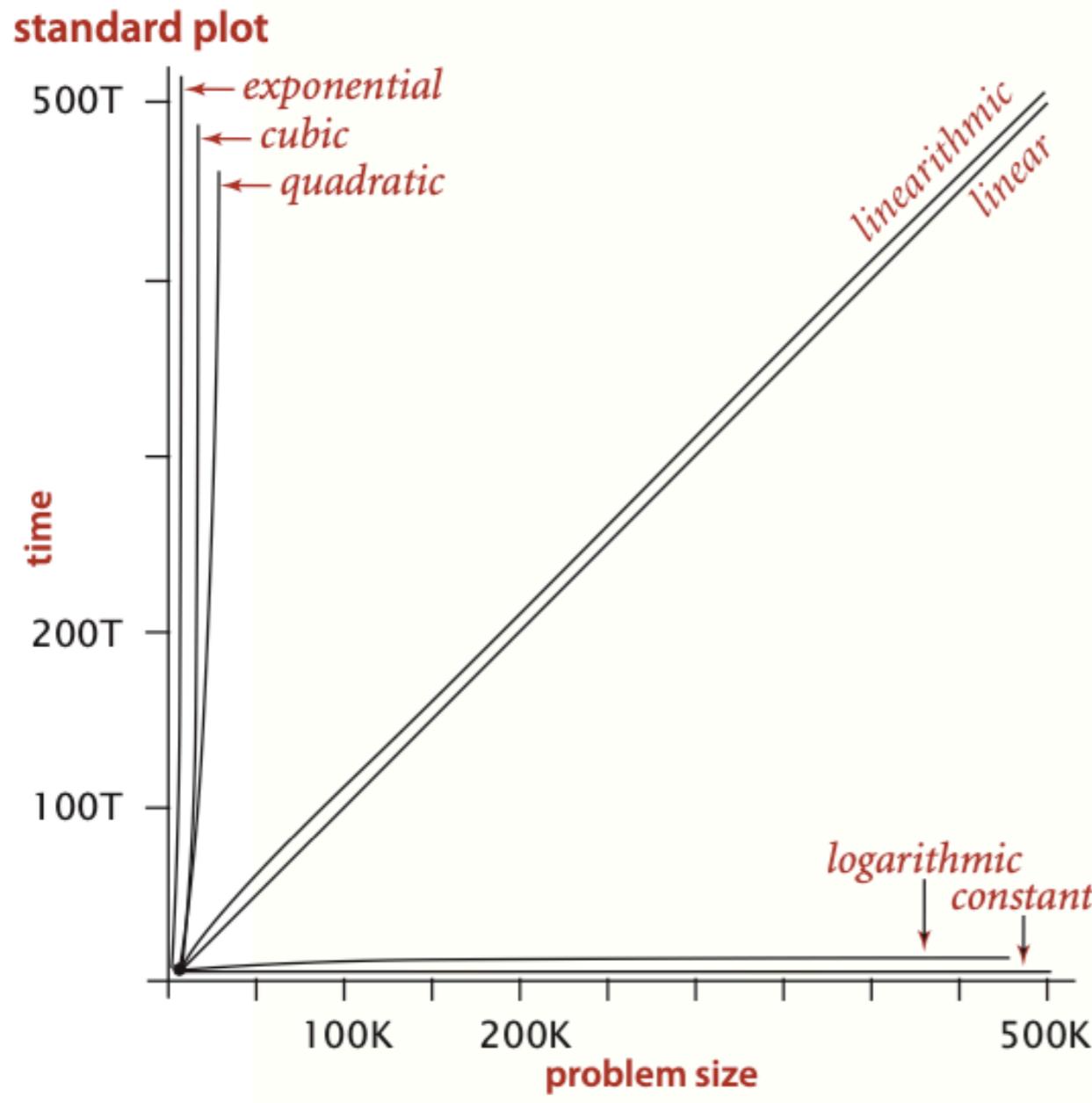
order of growth	description	function
constant		C
logarithmic		$\log N$
linear		N
linearithmic		$N \log N$
quadratic		N^2
cubic		N^3
exponential		2^N
Commonly encountered order-of-growth functions		

Funciones de crecimiento típicas

standard plot



Funciones de crecimiento típicas



Funciones de crecimiento típicas

Function	n					
	10	100	1,000	10,000	100,000	1,000,000
1	1	1	1	1	1	1
$\log_2 n$	3	6	9	13	16	19
n	10	10^2	10^3	10^4	10^5	10^6
$n * \log_2 n$	30	664	9,965	10^5	10^6	10^7
n^2	10^2	10^4	10^6	10^8	10^{10}	10^{12}
n^3	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	10^3	10^{30}	10^{301}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$

La tasa de crecimiento es más importante...



VS



ent.	Cray-1 Fortran Algoritmo Cubico	TRS-80 Basic Algoritmo Lineal
10	3 microseg.	200 miliseg.
100	3 miliseg.	2 seg.
1000	3 seg.	20 seg.
10000	50 seg.	50 seg.
100000	49 min.	3.2 min.
1000000	95 años	5.4 horas

La tasa de crecimiento es más importante...

- Supongamos que incrementamos la velocidad del CPU 10 veces

Running Time $T(n)$	Max. Problem Size for 10^3 sec.	Max. Problem Size for 10^4 sec.	Increase in Max. Problem Size
$100n$	10	100	10.0
$5n^2$	14	45	3.2
$n^3/2$	12	27	2.3
2^n	10	13	1.3

- El programa lineal puede resolver en el mismo tiempo un problema 10 veces más grande, el cuadrático un problema 3,2 veces más grande, el cúbico 2,3 veces más grande, y el exponencial 1,3 veces
- En otras palabras, un incremento de 1000% en la velocidad del CPU permite resolver problemas de tamaño 1000%, 320%, 230% y 130%, en el mismo tiempo, respectivamente

Tasa de Crecimiento

Podemos clasificar los programas según su tasa de crecimiento

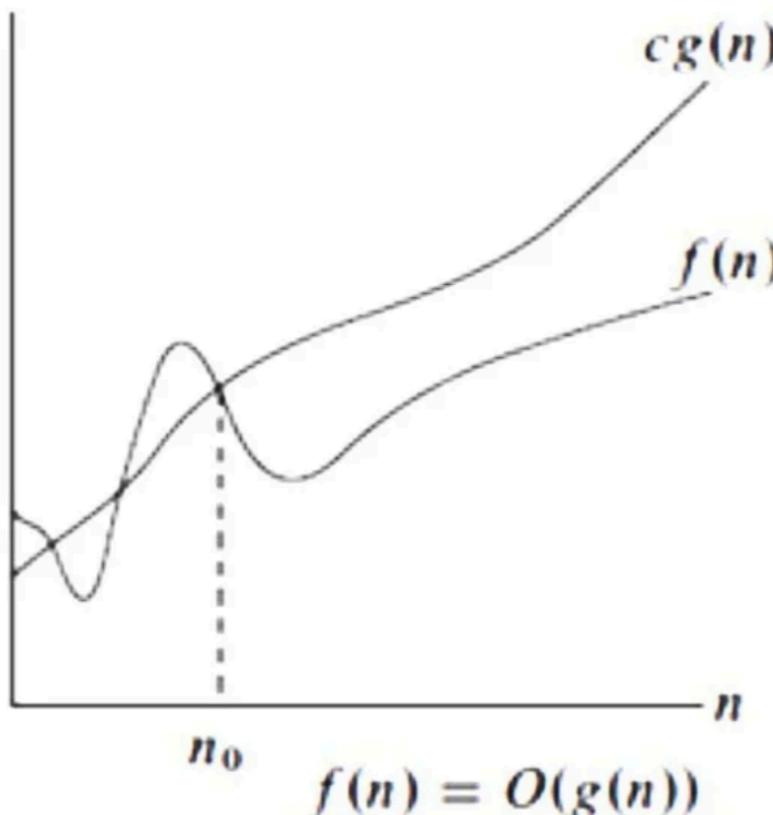
- $O(f(n))$: funciones que están acotadas por arriba por f
- $\Omega(f(n))$: funciones que están acotadas por abajo por f
- $\Theta(f(n))$: funciones que crecen exactamente como f

Notación O

$O(f(n))$ es la colección de funciones con una tasa de crecimiento menor o igual f

$t(n) \in O(g(n))$ ssi existen una constante positiva c y un entero no negativo n_0 tales que $t(n) \leq cg(n)$, para todo $n \geq n_0$.

Gráficamente:



Ejemplos:

$$n \in O(n^2)$$

$$100n + 5 \in O(n^2)$$

$$\frac{1}{2}n(n - 1) \in O(n^2)$$

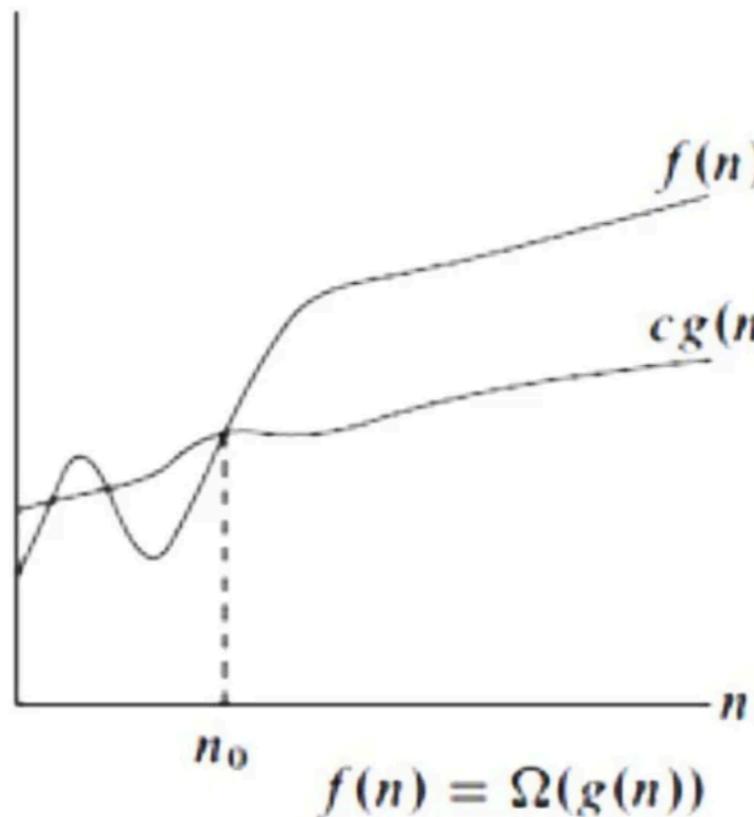
$$0.000001n^3 \notin O(n^2)$$

Notación Ω

$\Omega(g(n))$ es la colección de funciones con una tasa de crecimiento mayor o igual g.

$t(n) \in \Omega(g(n))$ ssi existen una constante positiva c y un entero no negativo n_0 tales que $t(n) \geq cg(n)$, para todo $n \geq n_0$

Gráficamente:



Ejemplos:

$$0.000001n^3 \in \Omega(n^3)$$

$$100n + 5 \notin \Omega(n^2)$$

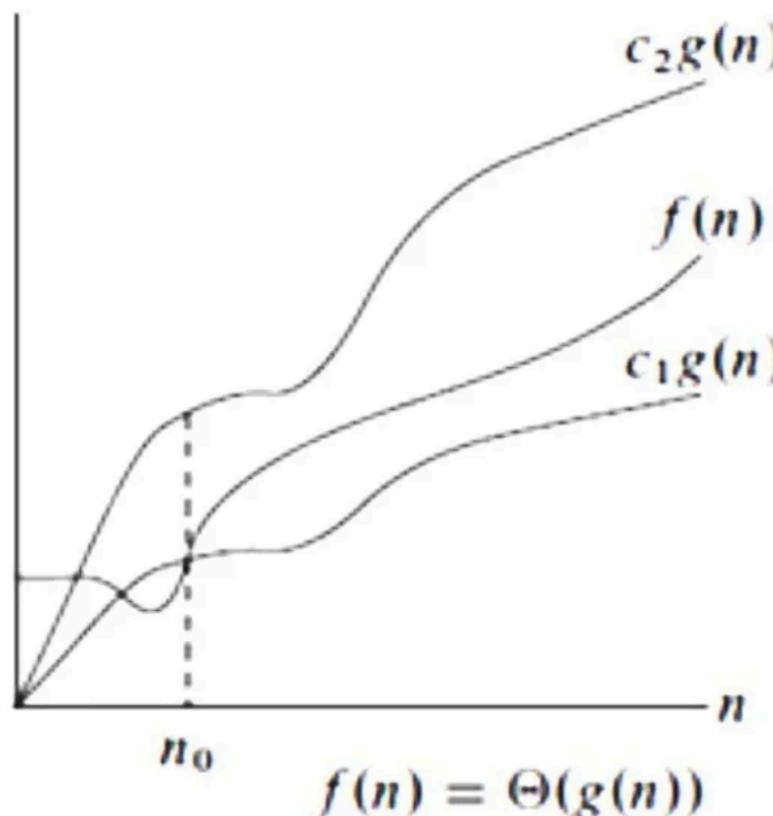
$$\frac{1}{2}n(n - 1) \in \Omega(n^2)$$

Notación Θ

$\Theta(f(n))$ Es la clase de funciones con un crecimiento exactamente igual a f

$t(n) \in \Theta(g(n))$ ssi existen constantes positivas c_1, c_2 y un entero no negativo n_0 tales que $c_1g(n) \leq t(n) \leq c_2g(n)$, para todo $n \geq n_0$

Gráficamente:



Ejemplos:

- $n \notin \Theta(n^2)$
- $100n + 5 \notin \Theta(n^2)$
- $\frac{1}{2}n(n - 1) \in \Theta(n^2)$
- $0.000001n^3 \notin \Theta(n^2)$

Propiedades

Propiedades de O (también valen para Θ):

- **Reflexividad:** $f(n) \in O(f(n))$
- **Transitividad:** $f(n) \in O(g(n))$ y $g(n) \in O(t(n)) \Rightarrow f(n) \in O(t(n))$
- **Sumas:** Si $f(n) \in O(g(n) + t(n))$, entonces $f(n) \in O(\text{Max}(g(n), t(n)))$
- **Mult. Constantes:** Si $f(n) \in O(c * g(n))$, entonces $f(n) \in O(g(n))$
- **Constantes:** $k \in O(1)$ para cualquier constante k
- **Polinomios:** Si $f(n)$ es un polinomio de grado k , $f(n) \in O(n^k)$

Algunas Clases Importantes

Las notaciones $O \Theta \Omega$ permiten clasificar los programas según su orden, hay infinitas clases de funciones de crecimiento, pero algunas importantes:

- 1 : Algoritmos constantes, no dependen de la entrada
- n : Algoritmos lineales, recorren la entrada una cantidad constante de veces.
- $n * \log n$: Varios algoritmos de sorting caen en esta clase

Algunas Clases Importantes (cont)

- n^2 : Algoritmos cuadráticos, por cada elemento recorren una vez la entrada
- n^3 : Algoritmos cúbicos, tres ciclos anidados.
- 2^n : Exponencial resuelven un problema utilizando varias instancias menores del mismo
- $n!$: Factorial tratan todas las combinaciones posibles

Reglas para el análisis en el peor caso

Analizar el código fuente aplicando las reglas a continuación:

- Las operaciones elementales toman tiempo constante c
 - Asignaciones, operaciones lógicas básicas, operaciones aritméticas básicas, acceso a arreglos, etc...
- El tiempo de $S_1; S_2$ es: $T_{S_1} + T_{S_2}$
 - En particular, si $T_{S_1} = c_1 \wedge T_{S_2} = c_2 \implies T_{S_1;S_2} = c_3$, donde $c_3 = \max(c_1, c_2)$
- El tiempo de *If B then S₁ else S₂* es: $T_B + \max(T_{S_1}, T_{S_2})$
- Si hay una invocación a un procedimiento $P(x)$, tenemos que computar el tiempo de P por separado

Análisis de Ciclos

```
for (int i=0; i<n; i++) {
    for (int j=0; j<m; j++) {
        A[i][j] = 0;
    }
}
```

El tiempo que tarda un ciclo es:

- el tiempo que tarda la inicialización (se omite si es constante) más:
 - multiplicar la cantidad de iteraciones por:
 - más el tiempo que tarda la evaluación de la condición (se omite si es constante),
 - el tiempo que tarda el código del cuerpo,
 - más el tiempo que tarda el incremento (en un for) y el tiempo que tarda saltar al inicio del ciclo (se omite si es constante)

Análisis de Ciclos

```
for (int i=0; i<n; i++) {
    for (int j=0; j<m; j++) {
        A[i][j] = 0;
    }
}
```

Luego, ignorando constantes, el tiempo en el peor caso viene dado por:

$$T(n, m) = \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} c = c * m * n$$

El algoritmo es: $\Theta(n * m)$

Propiedades de las sumatorias

$$\sum_{i=k}^n c = ((n - k) + 1) * c$$

$$\sum_{i=0}^n c^i = \frac{c^{n+1} - 1}{c - 1}, \quad c \neq 1$$

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

$$\sum_{i=k}^n (t + t') = \sum_{i=k}^n t + \sum_{i=k}^n t'$$

$$\sum_{i=1}^n i^2 = \frac{n(n + 1)(2n + 1)}{6}$$

$$\sum_{i=k}^n c * t = c * \sum_{i=k}^n t$$

$$\sum_{i=1}^n i^3 = \frac{n^2(n + 1)^2}{4}$$

Análisis en el peor caso: Ejemplo

```
public static boolean containsDuplicate(int[] a){  
    for (int i=0; i< a.length; i++){  
        for (int j=0; j < a.length; j++){  
            if (a[i] == a[j] && i!=j)  
                return true;  
        }  
    }  
    return false;  
}
```

El peor caso es cuando el arreglo no tiene elementos duplicados

Sea $n = a.length$, el tiempo viene dado por:

$$T(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} c = c * n * n = c * n^2$$

El algoritmo es: $\Theta(n^2)$

Análisis del Selection Sort

- Tamaño de entrada: La longitud del arreglo a ordenar.
- Peor caso: El arreglo está ordenado en forma creciente.
- Operaciones Elementales: Asignaciones, comparaciones, operaciones aritméticas, acceso a arreglos.

Debemos dar su función de crecimiento...

Ejemplo: Selection Sort

```
for i := 1 to n - 1 do begin
{ select the lowest among A[i], ..., A[n]
and swap it with A[i] }
    lowindex := i;
    lowkey := A[i].key;
    for j := i + 1 to n do begin
        { compare each key with current
        lowkey }
        if A[j].key < lowkey then begin
            lowkey := A[j].key;
            lowindex := j
        end;
    end;
    swap(A[i], A[lowindex])
end
```

$$\begin{aligned} T(n) &= \sum_{i=1}^{n-1} (c_2 + \sum_{j=i+1}^n c_3) \\ &= \sum_{i=1}^{n-1} (c_2 + c_3 * (n - i)) \\ &= \sum_{i=1}^{n-1} c_2 + \sum_{i=1}^{n-1} c_3 * (n - i) \\ &= c_2 * (n - 1) + c_3 * (\sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i) \\ &= c_2 n - c_2 + c_3 * (n^2 - n - 1/2n^2 + 1/2n) \\ &= c_2 n - c_2 + c_3/2 * n^2 - c_3/2 * n \end{aligned}$$

Ejemplo: Selection Sort

```
for i := 1 to n - 1 do begin
{ select the lowest among A[i], ..., A[n]
and swap it with A[i] }
    lowindex := i;
    lowkey := A[i].key;
    for j := i + 1 to n do begin
        { compare each key with current
        lowkey }
        if A[j].key < lowkey then begin
            lowkey := A[j].key;
            lowindex := j
        end;
    end;
    swap(A[i], A[lowindex])
end
```

$$\begin{aligned} T(n) &= \sum_{i=1}^{n-1} (c_2 + \sum_{j=i+1}^n c_3) \\ &= \sum_{i=1}^{n-1} (c_2 + c_3 * (n - i)) \\ &= \sum_{i=1}^{n-1} c_2 + \sum_{i=1}^{n-1} c_3 * (n - i) \\ &= c_2 * (n - 1) + c_3 * (\sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i) \\ &= c_2 n - c_2 + c_3 * (n^2 - n - 1/2n^2 + 1/2n) \\ &= c_2 n - c_2 + c_3/2 * n^2 - c_3/2 * n \end{aligned}$$

El algoritmo es: $\theta(n^2)$

Algoritmos Recursivos

La técnica principal es expansión de recurrencias:

- La función de tiempo viene expresada por una ecuación recursiva
- Para resolverlas tenemos que utilizar sustituciones
- Los casos bases nos permiten terminar este proceso de sustitución

Algoritmos recursivos: Ejemplo

Consideremos de nuevo el método:

```
/**  
 * Version ineficiente de f(n)=2^n  
 */  
public static long exp1(long n){  
    if (n == 0)  
        return 1;  
    else  
        return exp1(n-1) + exp1(n-1);  
}
```

Hace dos llamadas recursivas
por cada predecesor de n

$$T(0) = c_1$$

$$T(n) = 2 * T(n - 1) + c_2$$

Ecuación de
recurrencia

Resolvamos la ecuación

$$T(n) = 2 * T(n - 1) + c_2 \text{ si } n \geq 1$$

$$\{ T(n - 1) = 2 * T(n - 2) + c_2 \}$$

$$= 2 * (2 * T(n - 2) + c_2) + c_2 \text{ si } n \geq 2$$

$$= 4 * T(n - 2) + 3c_2 \text{ si } n \geq 2$$

$$\{ T(n - 2) = 2 * T(n - 3) + c_2 \}$$

$$= 4 * (2 * T(n - 3) + c_2) + 3c_2 \text{ si } n \geq 3$$

$$= 8 * T(n - 3) + 7c_2 \text{ si } n \geq 3$$

:

:

$$= 2^i * T(n - i) + (2^i - 1)c_2 \text{ si } n \geq i$$

Podemos reemplazar $T(n - i)$ por c_1 cuando $n - i = 0$ (equiv. $i = n$)

Reemplazando i por n :

$$= 2^n * T(n - n) + (2^n - 1)c_2$$

$$= 2^n c_1 + (2^n - 1)c_2 = (c_1 + c_2)2^n - c_2$$

Ecuación de recurrencia:

$$T(0) = c_1$$

$$T(n) = 2 * T(n - 1) + c_2$$

$$T(n) \in \theta(2^n)$$

Otra versión

Otra versión más eficiente del mismo algoritmo:

```
/**  
 * Version mas eficiente de f(n) = 2^n  
 */  
public static long exp2(long n){  
    if (n == 0)  
        return 1;  
    else  
        return 2*exp2(n-1);  
}
```

Hace solo una llamada recursiva por cada predecesor de n

$$T(0) = c_1$$

$$T(n) = T(n - 1) + c_2$$

Ecuación de recurrencia

Resolvamos la ecuación

$$\begin{aligned}T(n) &= T(n - 1) + c_2 \quad \text{si } n \geq 1 \\&\quad \{T(n - 1) = T(n - 2) + c_2\} \\&= (T(n - 2) + c_2) + c_2 \quad \text{si } n \geq 2 \\&= T(n - 2) + 2c_2 \quad \text{si } n \geq 2 \\&\quad \{T(n - 2) = T(n - 3) + c_2\} \\&= (T(n - 3) + c_2) + 2c_2 \quad \text{si } n \geq 3 \\&= T(n - 3) + 3c_2 \quad \text{si } n \geq 3 \\&\quad \vdots \\&= T(n - i) + ic_2 \quad \text{si } n \geq i\end{aligned}$$

Ecuación de recurrencia:

$$T(0) = c_1$$

$$T(n) = T(n - 1) + c_2$$

Podemos reemplazar $T(n - i)$ por c_1 cuando $n - i = 0$ (equiv. $i = n$)

Reemplazando i por n :

$$\begin{aligned}&= T(n - n) + nc_2 \\&= c_1 + nc_2\end{aligned}$$

$T(n)$ es $\theta(n)$

Algoritmo lineal, mucho más eficiente!

Un Ejemplo

Consideremos:

```
public void ordenar( int A[], int i, int j ) {  
    if ( i<=j ) {  
        int ind = i; boolean ordenado = false;  
        ordenar(A, i+1, j);  
        while ( (ind<j) & !ordenado ) {  
            if ( A[ind]>A[ind+1] ) {  
                int aux = A[ind+1];  
                A[ind+1] = A[ind];  
                A[ind] = aux;  
            }  
            else {  
                ordenado = true;  
            }  
            ind++;  
        }  
    }  
}
```

Método de ordenamiento recursivo

Peor caso: ordenado
decrecientemente

$$\begin{aligned} T(0) &= c_1 \\ T(n) &= c_2 + T(n - 1) + \sum_{i=1}^{n-1} c_3 \\ &= c_2 + T(n - 1) + c_3 n - c_3 \\ &\quad \{c_4 = c_2 - c_3\} \\ &= T(n - 1) + c_3 n + c_4 \end{aligned}$$

Resolvamos la ecuación

$$T(n) = T(n - 1) + c_3n + c_4 \text{ si } n \geq 1$$

$$\{ T(n - 1) = T(n - 2) + c_3(n - 1) + c_4 \}$$

$$= T(n - 2) + c_3n + c_3(n - 1) + 2c_4 \text{ si } n \geq 2$$

$$\{ T(n - 2) = T(n - 3) + c_3(n - 2) + c_4 \}$$

$$= T(n - 3) + c_3n + c_3(n - 1) + c_3(n - 2) + 3c_4 \text{ si } n \geq 3$$

⋮

$$= T(n - i) + \sum_{j=0}^{i-1} (c_3(n - j)) + ic_4 \text{ si } n \geq i$$

$$= T(n - i) + c_3 * (\sum_{j=0}^{i-1} n - \sum_{j=0}^{i-1} j) + ic_4 \text{ si } n \geq i$$

Ecuación de recurrencia:

$$T(0) = c_1$$

$$T(n) = T(n - 1) + c_3n + c_4$$

Resolvamos la ecuación

$$= T(n - i) + c_3 * \left(\sum_{j=0}^{i-1} n - \sum_{j=0}^{i-1} j \right) + ic_4 \text{ si } n \geq i$$

$$= T(n - i) + c_3 * (in - i^2/2 - i/2) + ic_4 \text{ si } n \geq i$$

Ecuación de recurrencia:

$$T(0) = c_1$$

$$T(n) = T(n - 1) + c_3n + c_4$$

Podemos reemplazar $T(n - i)$ por c_1 cuando $n - i = 0$ (equiv. $i = n$)

Reemplazando i por n y simplificando

$$= c_1 + c_3/2 * n^2 - c_3/2 * n + nc_4$$

Resolvamos la ecuación

$$= T(n - i) + c_3 * \left(\sum_{j=0}^{i-1} n - \sum_{j=0}^{i-1} j \right) + ic_4 \text{ si } n \geq i$$

$$= T(n - i) + c_3 * (in - i^2/2 - i/2) + ic_4 \text{ si } n \geq i$$

Ecuación de recurrencia:

$$T(0) = c_1$$

$$T(n) = T(n - 1) + c_3n + c_4$$

Podemos reemplazar $T(n - i)$ por c_1 cuando $n - i = 0$ (equiv. $i = n$)

Reemplazando i por n y simplificando

$$= c_1 + c_3/2 * n^2 - c_3/2 * n + nc_4$$

El algoritmo es: $\theta(n^2)$

Actividades

- Leer el capítulo 1 del libro "Data Structures and Algorithms". A. Aho, J. Hopcroft, J. Ullman. Addison-Wesley. 1983
- Leer los capítulos 2 y 3 del libro "Introduction to Algorithms (3rd edition)". T. Cormen, C. Leiserson, R. Rivest, C. Stein. 2010.

Bibliografía

- "Data Structures and Algorithms". A. Aho, J. Hopcroft, J. Ullman. Addison-Wesley. 1983
- "Introduction to Algorithms (3rd edition)". T. Cormen, C. Leiserson, R. Rivest, C. Stein. 2010
- "Algorithms (4th edition)". R. Sedgewick, K. Wayne. Addison-Wesley. 2016