ANÁLISIS

DE

ALGORITMOS

Construcción de un algoritmo

- Podemos atacar el problema desde distintos puntos de vista,
- Aplicando distintas estrategias, y por tanto, llegando a soluciones algorítmicas distintas.

- Desde el **punto de vista computacional**, **es necesario** disponer de alguna forma de comparar una solución algorítmica con otra, para conocer cómo se comportarán cuando las implementemos, especialmente al atacar problemas "grandes".

La **complejidad algorítmica** es **una métrica teórica** que se aplica a los algoritmos en este sentido.

- Es un concepto fundamental para todos los programadores.
- Entender la complejidad es importante porque para resolver muchos problemas, utilizamos algoritmos ya diseñados. Conocer el valor de su complejidad **puede ayudarnos a escoger uno u otro**.

Características de los Algoritmos

- **Eficaces:** Cumplen con el requerimiento solicitado
- **Eficientes**: Lo hacen mejor que otros
- **Legibles,** claros y bien estructurados
- **Generales** (capaz de resolver una clase de problemas lo más amplia posible), de uso y mantenimiento fácil.
- **EFICIENTE:**
  - Rápido (eficiencia temporal)
  - Hace buen uso de recursos (eficiencia espacial).

- Al **tiempo** que consume un algoritmo para resolver un problema lo llamamos **complejidad temporal**

- A la **memoria** que utiliza el algoritmo la llamamos **complejidad espacial**

- La complejidad espacial, en general, tiene mucho menos interés. **El tiempo es un recurso mucho más valioso que el espacio**.

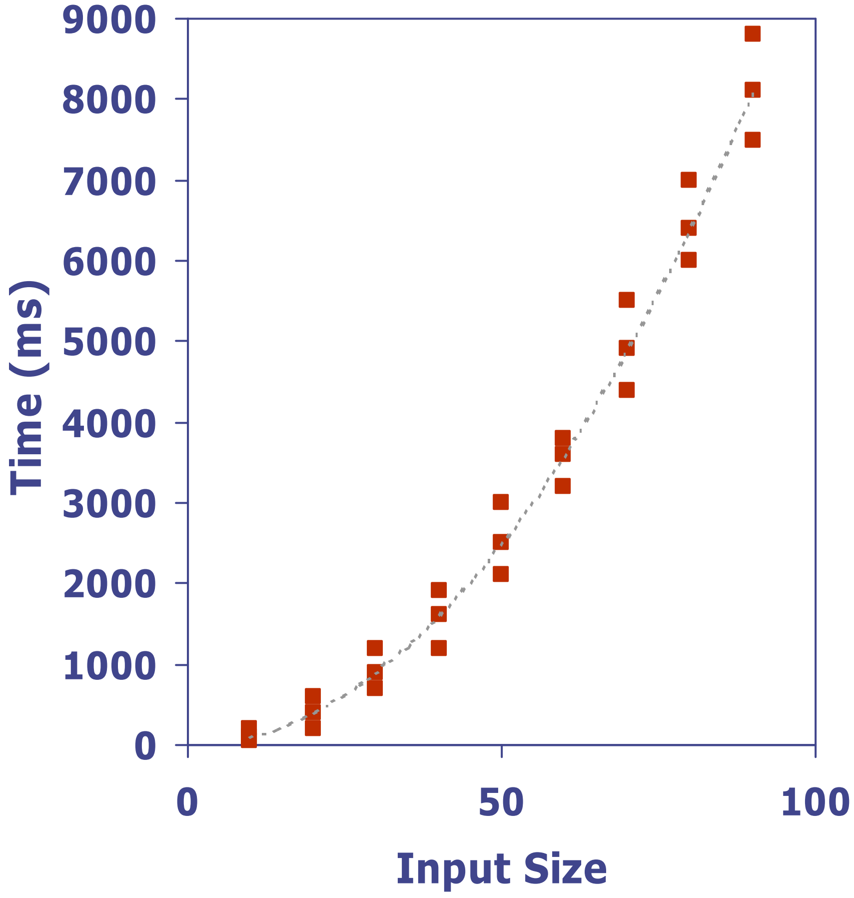
- Así que cuando hablamos de **complejidad** a secas, nos estamos refiriendo a **complejidad temporal.**

Complejidad

- ¿*Qué medimos?:* **Tiempo de ejecución**
- ¿*De qué depende?:*
  - Técnica utilizada para resolver el problema
  - Tamaño de la entrada
  - Otros factores (HW y SW)
    - **Características de la máquina (velocidad, tipo de procesador, RAM, SO…)**
    - **Software usado en la implementación del programa (lenguaje, compilador, bibliotecas)**
- ¿*Cómo medirlo?:*
  - Experimentalmente → *"a posteriori"*
  - Estimándolo matemáticamente → *"a priori"*

Estudios experimentales

- Escribir un programa implementando el algoritmo.
- Correr el programa con entradas de varios tamaños y composición.
- Usar un método tipo para tomar medidas precisas de cada corrida.
- Graficar los resultados.



Eficiencia de los algoritmos

Tiempo de ejecución:

- El tiempo de ejecución de un algoritmo típicamente crece con el tamaño de la entrada
- Pero no solo depende del tamaño, sino que influye el contenido de los datos
- Se distinguen 3 casos:
  - **Mejor Caso: mínimo tiempo posible.**
  - **Peor Caso: máximo tiempo posible. Es el caso más representativo.**
  - **Caso Medio: tiempo promedio.**

Entonces, la mejor opción es analizar el peor caso.

Análisis Teórico

- Utiliza una descripción de alto nivel del algoritmo, en lugar de una implementación
- Caracteriza los tiempos de ejecución como una función del número de elementos que deben ser procesados (tamaño de la entrada) → **T(N)**
- Toma en cuenta todas las posibles entradas y se emplea el caso peor (es más fácil de calcular)
- Permite evaluar el tiempo de ejecución de un algoritmo independientemente del entorno de hardware y software
- Interesa la velocidad de crecimiento del tiempo de ejecución en función del tamaño de la entrada. Comportamiento asintótico

COMPLEJIDAD

Cálculo de T(N):

- **Cálculo estimado de T(N)**: se mide el tiempo de ejecución empíricamente.

  Requiere trabajo planificado y sistemático, presentación de los resultados y validez.

- **Cálculo de la expresión de T(N):** se realiza un análisis matemático del algoritmo, ya sea para calcular una expresión exacta de T(N) o una cota superior de la misma (expresión asintótica, orden de magnitud).

Análisis Teórico

- Para la descripción del algoritmo se emplea un pseudocódigo.
  - Tiene menos nivel de detalle que un programa.
- Los pseudocódigos indican:
  - Control de flujo
  - Declaraciones de métodos
  - Llamadas a métodos
  - Retorno de valores
  - Expresiones

Los bucles son el término dominante para determinar la eficiencia

Análisis Teórico

- El tiempo se mide en función de cada operación o instrucción elemental **(primitiva)**:

  - Asignación
  - Llamada a método
  - Operación aritmética, lógica, relacional
  - Indexación en array
  - Seguir una referencia
  - Volver de un método

- Las operaciones primitivas son independientes de la máquina, del lenguaje de programación, del compilador, de cualquier otro elemento hardware o software.

- Cada una de ellas se contabiliza como 1 operación elemental.

- Se asume que tienen un tiempo de ejecución constante en un determinado modelo RAM (costo unitario).

Análisis Teórico

- Como ejemplo se muestra cierto código para la **búsqueda en un arreglo a con N elementos,** de manera que se pueda determinar si un valor específico destino se encuentra en el arreglo:

```
int i = 0;
bool encontrado = false;
while (( i < N) && !(encontrado))
    if (a[i] == destino)
        encontrado = true;
    else
        i++;
```

- **T(N) = 3 + 7N**

- 2 incializaciones, N repeticiones del ciclo en caso peor (que no esté en el arreglo), 3 operaciones para evaluar cada condición y 3 operaciones dentro del cuerpo en el peor caso

Ejemplo (en pseudocódigo)

incremento.c

```c
#include <stdio.h>
int main(void)
{
  int m, n, i, j;
  scanf("%d", &n);
  m = 0;
  for (i=0; i<n; i++)
    for (j=0; j<n; j++)
      m++;
  printf("%d\n", m);
  return 0;
}
```

`m = 0;`	1 paso	
`for (i=0; i<n; i++)`	$2n + 2$ pasos	
`for (j=0; j<n; j++)`	$2n + 2$ pasos,	n veces
`m++;`	1 paso,	n^2 veces
`return 0;`	1 paso	

T(n) =Total= $4n^2 + 5n + 4$

Gráfico comparativo

Tiempos de ejecución para tamaños de entrada pequeños

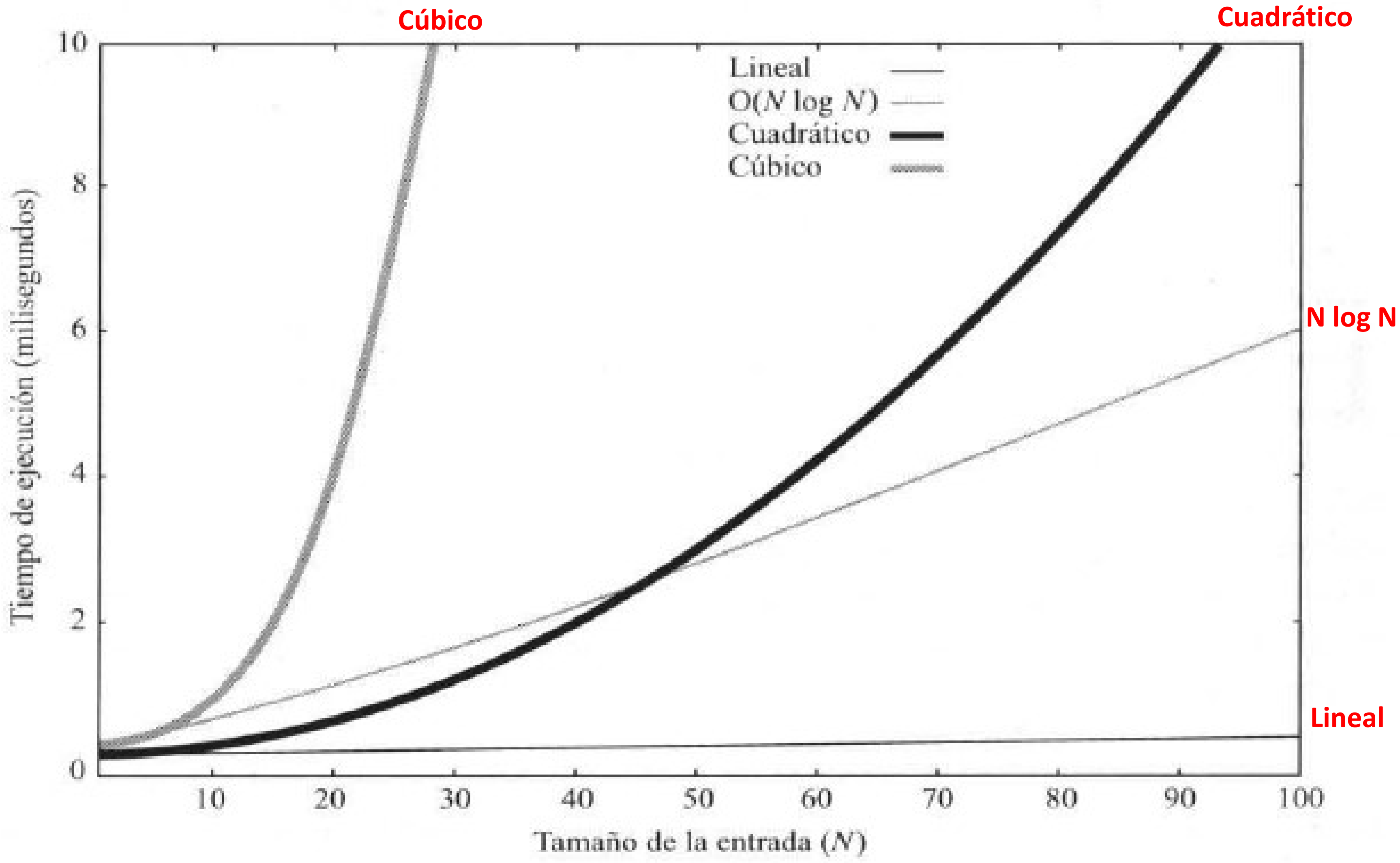
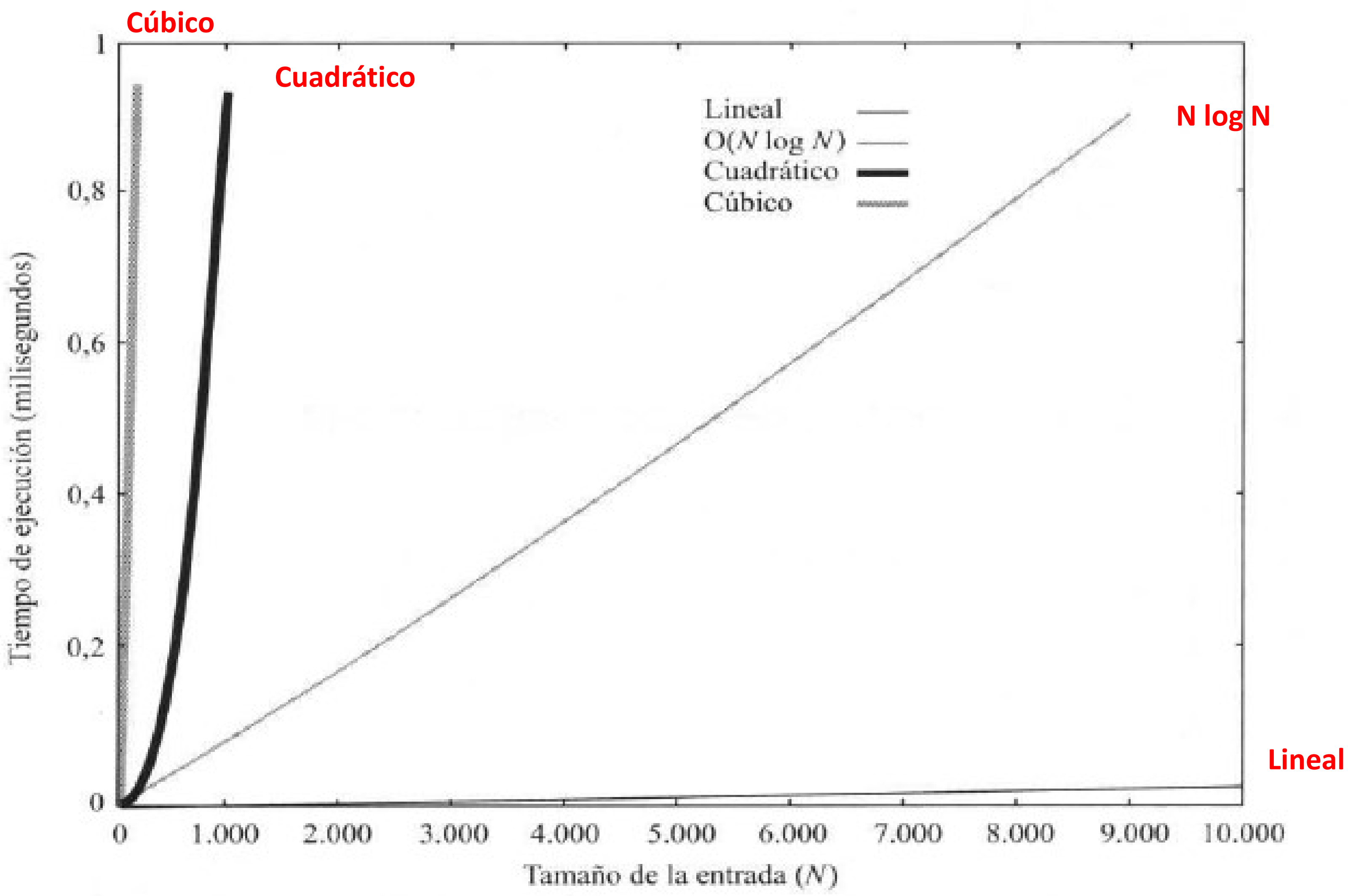


Gráfico comparativo

Tiempos de ejecución para tamaños de entrada mayores



COMPLEJIDAD- Función de Orden

Método de Cálculo de la expresión de T(N):

- **Expresar T(N) como el número de expresiones elementales (primitivas) ejecutadas por el algoritmo**

- **Expresar T(N) asintóticamente: interesa el orden de magnitud de la función (valores grandes de n). El objetivo es ver *cómo crece el tiempo de ejecución cuando crece el tamaño de la instancia*.**

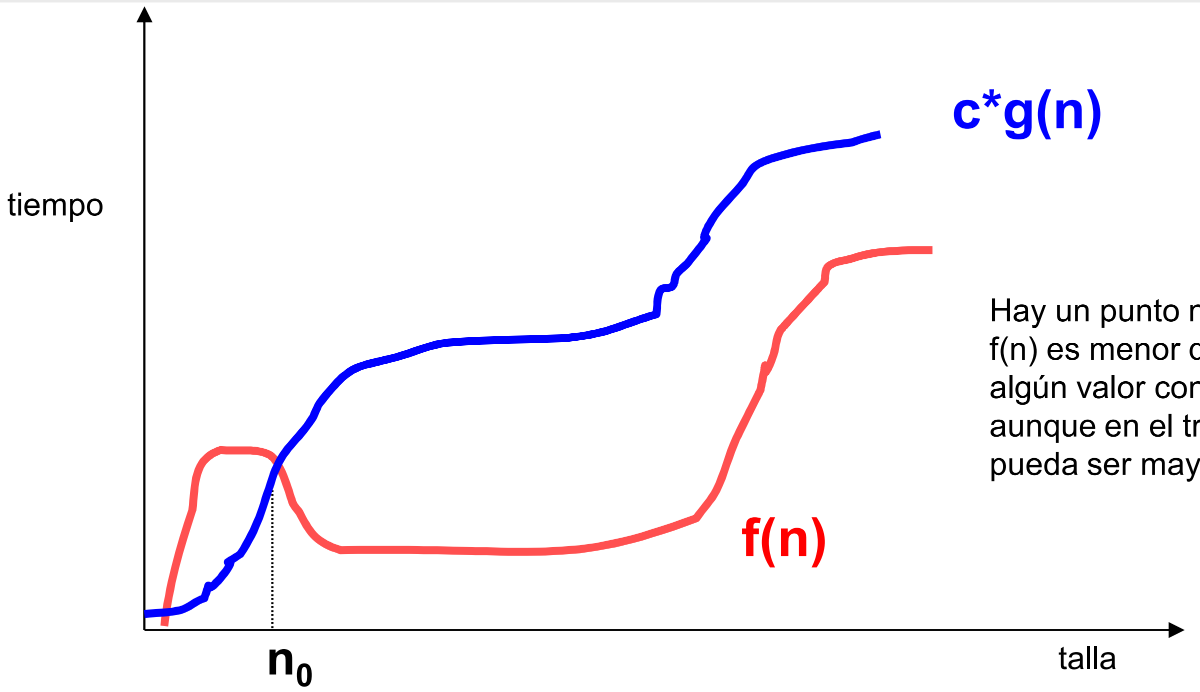
Se prescinde de las contantes multiplicativas y de los órdenes de magnitud menores:

$T(N) = 3n^2 + 4n + 4 \sim n^2$

Orden cuadrático: $O(n^2)$

Notación O

Interesa conocer el comportamiento del tiempo de ejecución cuando la cantidad de datos (N) crece.



Hay un punto n_0 a partir del cual f(n) es menor que cg(n) para algún valor constante de c, aunque en el tramo inicial f(n) pueda ser mayor que cg(n).

Dadas 2 funciones f(n) y g(n) se dice que **f(n) es O(g(n)) del "orden de complejidad de g(n)" si existen constantes positivas c y n_0 tal que:**

$$f(n) <= c*g(n) \quad \text{para } n >= n_0$$

- g(n) acota superiormente a f(n)

Notación O - Ejemplos

- $f(n) = 7n - 2$

  $f(n) = \mathbf{O(n)}$

- $f(n) = 3n^3 + 20n^2 + 5$

  $f(n) = \mathbf{O(n^3)}$

- $f(n) = 3 \log n + 5$

  $f(n) = \mathbf{O(\log n)}$

Notación O

La siguiente es una forma alterna y pragmática de pensar acerca de las estimaciones **Big O:**

Busque sólo en el término con el exponente más alto y no ponga atención a los múltiplos constantes.

Por ejemplo, todas las siguientes son $O(N2)$:

$N^2 + 2N + 1$, $3N^2 + 7$, $100N^2 + N$

Todas las siguientes son $O(N3)$:

$N^3 + 5N^2 2 + N + 1$, $8N^3 + 7$, $100N^3 + 4N + 1$

Sin duda, las estimaciones de tiempo de ejecución Big O son burdas, pero contienen cierta información.

No diferencian entre un tiempo de ejecución de $5N + 5$ y un tiempo de ejecución de $100N$, pero nos permiten distinguir entre algunos tiempos de ejecución y por consecuencia determinan que algunos algoritmos son más rápidos que otros.

Notación O – Algunas propiedades

1. Siendo c una constante, $c*O(f(n)) = O(f(n))$

   Por ejemplo si $f(n) = 3n^4$, entonces $f(n) = 3*O(n^4) = O(n^4)$

2. $O(f(n)) + O(g(n)) = O(f(n)+g(n))$.

   Por ejemplo, si $f(n) = 2e^n$ y $g(n) = 2n^3$:

   $$O(f(n)) + O(g(n)) = O(f(n)+g(n)) = O(2e^n + 2\ n^3) = O(e^n)$$

3. $Maximo(O(f(n)),O(g(n))) = O(Maximo(f(n),g(n))$.

   Por ejemplo,

   $$Maximo(O(log(n)),O(n)) = O(Maximo(log(n),n)) = O(n).$$

4. $O(f(n)) * O(g(n)) = O(f(n)*g(n))$.

   Por ejemplo, si $f(n) = 2n^3$ y $g(n) = n$:

   $$O(f(n))*\ O(g(x)) = O(f(x)*g(x)) = O(2n^3 * n) = O(n^4)$$

Principales órdenes de complejidad

Notación O	Orden
$O(1)$	Constante
$O(\log n)$	Logarítmico
$O(n)$	Lineal
$O(n^2)$	Cuadrático
$O(n^a)$	Polinomial ($a > 2$)
$O(a^n)$	Exponencial ($a > 1$)
$O(n!)$	Factorial

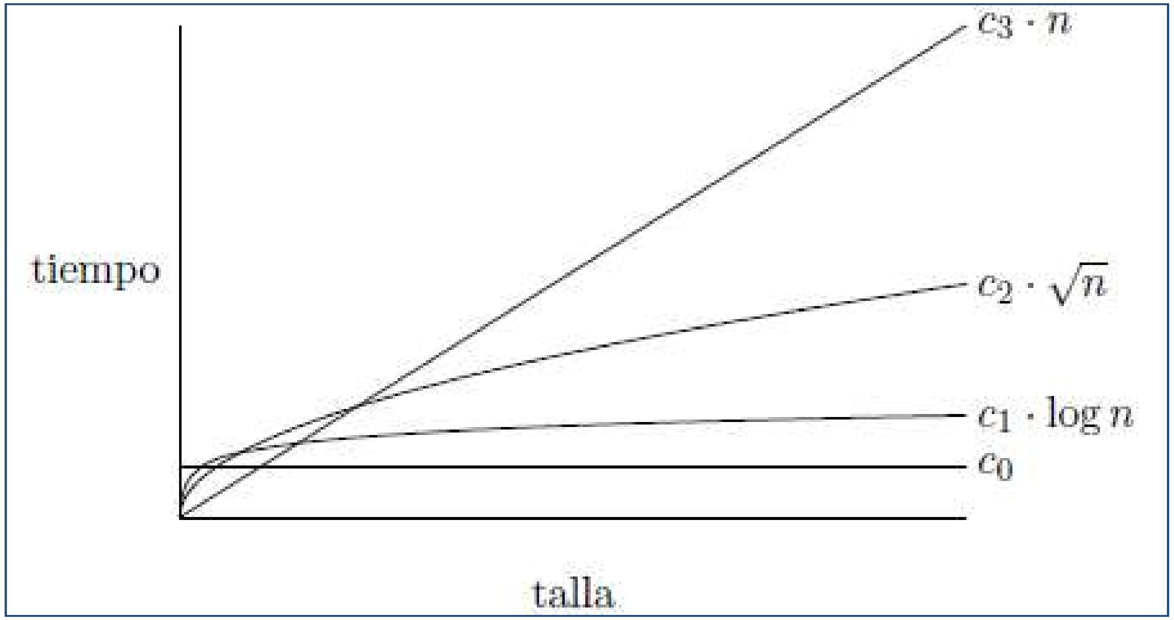
Jerarquía de cotas

Gráfica de comparación de crecimiento de funciones

- Un algoritmo de coste **constante** ejecuta un n° constante de instrucciones. Un algoritmo que soluciona un problema en tiempo constante es lo ideal.

- El coste de un algoritmo **logarítmico** crece muy lentamente conforme crece n.

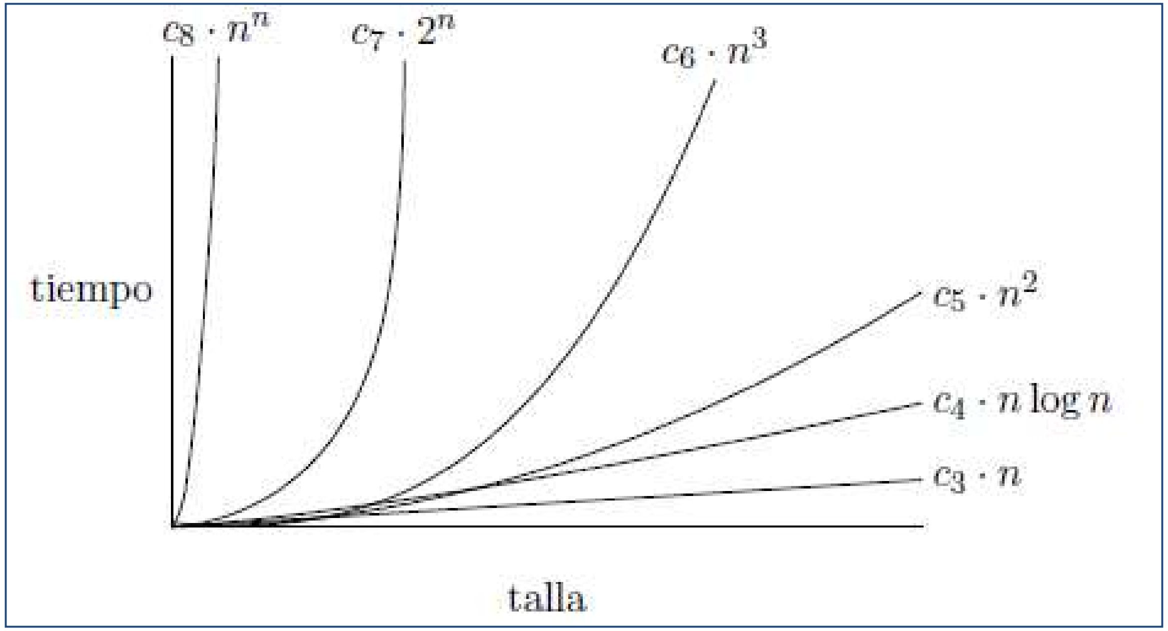
- Un algoritmo cuyo coste es √**n** crece a un ritmo superior que otro que es logarítmico, pero no llega a presentar un crecimiento lineal. Cuando la talla se multiplica por 4, el coste se multiplica por 2.



Jerarquía de cotas

Gráfica de comparación de crecimiento de funciones

- Un algoritmo cuyo coste temporal es **(n log n)** presenta un crecimiento del coste ligeramente superior al de un algoritmo lineal.
- Un algoritmo de coste **cuadrático** empieza a dejar de ser útil para tallas medias o grandes, pues duplicar el tamaño del problema requiere 4 veces más tiempo.
- Un algoritmo de coste **cúbico** sólo es útil para problemas pequeños: duplicar el tamaño del problema hace que se tarde 8 veces más tiempo.
- Un algoritmo de coste **exponencial** raramente es útil. Duplicar el tamaño del problema requiere ¡unas 1000 veces más tiempo!



Crecimiento de funciones

En la siguiente tabla se comparan los tiempos para los diferentes ordenes de complejidad.

	Tiempo						
n	Log n	$\sqrt{n}$	n	n log n	n^2	n^3	2^n
2	1	1.4	2	2	4	8	4
4	2	2,0	4	8	16	64	16
8	3	2,8	8	24	64	512	256
16	4	4,0	16	64	256	4.096	65.536
32	5	5,7	32	160	1.024	32.768	4.294.967.296
64	6	8,0	64	384	4.096	262.144	$1,8 \cdot 10^{19}$
128	7	11,0	128	896	16.536	2.097.152	$3,4 \cdot 10^{38}$

COMPLEJIDAD

```cpp
#include <iostream>
#include <stdio.h>

using namespace std;
const int Max = 5;

int main (void)
{     int n,f,min;
      cout << "ingrese datos:   " ;
      cin >> n;
      min =n;
      for (f=1; f< Max; f++){
          cin >> n;
          if (n<min) min= n;
      };
      cout << endl << min;
      return 0;
}
```

Cantidad de operaciones primitivas:

5n - 1

```
C:\Program Files\ZinjaI\runner.exe
ingrese datos:   4
1
0
7
2

0

<< El programa ha finaliz
```

COMPLEJIDAD

- Problema: Sumar los elementos del triángulo superior de una matriz de NxN.

```
Algoritmo B1 (A,N)
        S=0
        for i=0 to N-1 do
            for j=0 to N-1
                if j>=i
                  then S=S +A[i , j ]
        return S
```

```
Algoritmo B2 (A,N)
        S=0
        for i=0 to N-1 do
            for j=i to N-1
                S=S +A[i , j ]
        return S
```

$T(N)= 6N^2 +4N + 4$
$=1 + (2+2N) + N(2+2N) + N^2 + 3N^2 + 1$

$T(N)= 5N^2+ 4N + 4 =$
$1 + (2+2N) + (2N^2+2N) + 3N^2 +1$

Análisis de COMPLEJIDAD

Complejidad de Búsqueda Lineal: $O(N)$

Complejidad de Búsqueda Binaria: $O(\log_2 N)$

Complejidad de Mezcla de dos arreglos de N1 y N2 elementos: $O(N1 + N2)$

Complejidad de ordenamiento por Selección: $O(N^2)$

Hace un total de N-1 iteraciones
y en cada una (N-i-1) comparaciones y
3 asignaciones del intercambio:

$$3N + (N-1) + (N-2) + \ldots + 1$$

Complejidad de ordenamiento por Inserción: $O(N^2)$

Hace un total de N-1 iteraciones y
en cada una i comparaciones y
(i+1) asignaciones (por el desplazamiento):

$$((N-1) + (N-2) + \ldots + 1) + (N + (N-1) + \ldots + 2)$$

COMPLEJIDAD

Complejidad de ordenamiento por Burbuja: $O(N^2)$

Hace un total de N-1 iteraciones o pasadas y
en cada una (N-i-1) comparaciones +
3 (N-i-1) asignaciones:
((N-1) + (N-2) + …..+ 1) + 3((N-1) + …. + 1)

Complejidad de ordenamiento por Burbuja Mejorado : $O(N^2)$

Solo se modifica el Mejor Caso – tamaño N

Complejidad de ordenamiento por Merge (Merge Sort): $O(N \log N)$

COMPLEJIDAD

La complejidad es del algoritmo, no del programa ni de los datos

- No perder tiempo intentando optimizar el código, sino:

Tratar de optimizar el algoritmo!!

Ejercicios

- Obtener el menor de un arreglo de números “ordenado ascendentemente” **$O(1)$**
- Buscar un número en un arreglo desordenado **$O(N)$**
- Determinar el mayor de 3 números **$O(1)$**
- Buscar un elemento en una matriz **$O(N^2)$**
- Imprimir los valores de la diagonal principal de una matriz de N filas y N columnas **$O(N)$**
- Sumar los elementos de las columnas impares de una matriz **$O(N^2)$**
- En una matriz cuadrada con sus diagonales principal y secundaria ordenadas ascendentemente, calcular el producto de los elementos máximos de cada diagonal **$O(1)$**
- Mezclar 2 arreglos **$O(N)$**

LEER

Capítulo 14

Apunte de Complejidad (Campus)
Marzal - Gracia

Capítulo 6

Libro: Estructura de datos en C++
Joyanes