




Análisis y diseño de algoritmos

Sesión 02

Logro de la sesión

Al finalizar la sesión, el estudiante realiza un análisis de la complejidad, de un algoritmo mediante el análisis de programas elaborados en un Lenguaje de Programación.

Eficiencia y Complejidad – Análisis de Algoritmos



"bit twiddling: 1.(peyorativo) Un ejercicio de afinación (tunning - ver [tune](#)) en el que se dedican cantidades increíbles de tiempo y esfuerzo para producir muy pocas mejoras notables, a menudo con el resultado que el código se vuelva incomprensible."

- The Hackers Dictionary, version 4.4.7

Tune (Afinación): Optimizar un programa o sistema para un entorno particular, por ejemplo, cambiando el # de líneas. Se puede afinar por tiempo (ejecución más rápida), afinar por espacio (menor uso de memoria) o afinar para configuración (uso más eficiente del hardware) .

Pregunta 1

- “Un programa encuentra todos los números primos entre 2 y 1.000.000.000 en 0,37 segundos .”
 - ¿Es ésta una solución rápida?

A. no

B. si

C. depende

Eficiencia

- No solo se escriben programas.
- También se ***analizan***.
- ¿Qué tan eficiente es un programa?
 - ¿Cuánto tiempo tarda el programa en completarse?
 - ¿Cuánta memoria usa un programa?
 - ¿Cómo cambian estos a medida que cambia la cantidad de datos de entrada?
 - ¿Cuál es la diferencia entre la eficiencia del mejor caso, el caso promedio y el del peor de los casos, si corresponde?

Técnica para “medir” la eficiencia

- El análisis teórico que hacemos aún es informal
 - Técnicas más formales, más adelante
- **¿Cuántos cálculos realizará este programa/método/algoritmo para obtener la respuesta?**
- Muchas simplificaciones
 - Ver los algoritmos como programas Java/python
 - Contar las operaciones/sentencias ejecutadas en un programa/método/función
 - Encontrar el número de operaciones/sentencias en función de la cantidad de datos de entrada
 - Centrarse en el *término dominante* en la función

Supuestos para contar las operaciones

- Una vez encontrada, acceder al valor de una primitiva es en un tiempo constante :

```
x = y; //una operación
```

- Las operaciones matemáticas y las comparaciones en expresiones booleanas son todas en tiempo constante.

```
x = y * 5 + z % 3; // 4 operaciones
```

- Una expresion “if” es de tiempo constante, si la prueba lógica y el tiempo para cada alternativa es constante

```
if( iMySuit == DIAMONDS || iMySuit == HEARTS )
```

```
    return RED;
```

```
else
```

```
    return BLACK;
```

```
// El if tiene 3 operaciones + 1 operacion return (peor caso)
```

Contar sentencias o declaraciones

```
int x; // 1 operación
```

```
x = 12; // 1 operación
```

```
int y = z * x + 3 % 5 * x / i; // 7 operaciones
```

```
x++; // 2 operaciones
```

```
boolean p = x < y && y % 2 == 0 || z >= y * x; // 9  
operaciones
```

```
int[] data = new int[100]; // 100 + 1 (declaración) + 1  
(asignación)
```

```
data[50] = x * x + y * y; // 5 operaciones
```


Pregunta 2

- ¿Cuál es la salida del siguiente código?

```
int total = 0;  
for(int i = 0; i < 2; i++)  
    total += 5;  
System.out.println( total );
```

A. 2 B. 5 C. 10 D. 15 E. 20

Pregunta 3

- Contar las operaciones/sentencias/declaraciones en bucles a menudo requiere un poco de *inducción matemática* informal
- ¿Cuál es la salida del siguiente código?

```
int total = 0;  
// asumir que limit es un int >= 0  
for(int i = 0; i < limit; i++)  
    total += 5;  
System.out.println( total );
```

- A. 0
- B. limit
- C. limit * 5
- D. limit * limit
- E. limit⁵

Pregunta 4

- ¿Cuál es el resultado del siguiente código?

```
int total = 0;
for (int i = 0; i < 13; i++)
    for (int j = 0; j < 11; j++)
        total += 2;
System.out.println(total);
```

- A. 24
- B. 120
- C. 143
- D. 286
- E. 338

Pregunta 5

- ¿Cuál es la salida del siguiente código?

```
int total = 0;  
// asumir que limit es un int >= 0  
for(int i = 0; i < limit; i++)  
    for(int j = 0; j < limit; j++)  
        total += 5;  
System.out.println( total );
```

- A. 5
- B. limit * limit
- C. limit * limit * 5
- D. 0
- E. limit⁵

Pregunta 6

- ¿Cuál es la salida cuando el método `sample` es llamado?

// pre: $n \geq 0, m \geq 0$

```
public static void sample(int n, int m) {  
    int total = 0;  
    for (int i = 0; i < n; i++)  
        for (int j = 0; j < m; j++)  
            total += 5;  
    System.out.println(total);  
}
```

A. 5

D. n^m

B. $n * m$

E. $(n * m)^5$

C. $n * m * 5$

Análisis Teórico de Algoritmos

- ¿De qué depende $T(n)$?

```
public static int search(int v[], int x) {  
    int index=-1;  
    int n=v.length;  
    for (int i=0; i<n && index==-1;i++) {  
        if (x==v[i]) index=i;  
    }  
    return index;  
}
```

Análisis Teórico de Algoritmos

- ¿De qué depende $T(n)$?

```
public static int search(int v[], int x) {  
    int index=-1;  
    int n=v.length;  
    for (int i=0; i<n && index==-1;i++) {  
        if (x==v[i]) index=i;  
    }  
    return index;  
}
```

-Tamaño de v

- Pero también del valor de x

Análisis Teórico de Algoritmos

- ¿De qué depende $T(n)$?

```
public static int search(int v[], int x) {  
    int index=-1;  
    int n=v.length;  
    for (int i=0; i<n && index==-1;i++) {  
        if (x==v[i]) index=i;  
    }  
    return index;  
}
```

- Mejor-caso: x es igual a $v[0]$
- Pero-caso: x no está en v o es igual a $v[n-1]$,

Análisis Teórico de Algoritmos

- ¿De qué depende $T(n)$?

```
public static int search(int v[], int x) {  
    int index=-1;  
    int n=v.length;  
    for (int i=0; i<n && index==-1;i++) {  
        if (x==v[i]) index=i;  
    }  
    return index;  
}
```

- Cuando el tiempo de ejecución depende de una entrada particular, definimos $T(n)$ como el peor-caso de tiempo de ejecución

	#
int index=-1;	2
int n=v.length;	3
int i=0;	2
i<n && index==-1	3(n+1)
i++	2n
if (x==v[i]) index=i;	2n
return index	1
T(n)	7n+11

Ejemplo

```
public int total(int[] values) {  
    int result = 0;  
    for (int i = 0; i < values.length; i++)  
        result += values[i];  
    return result;  
}
```

- ▶ ¿Cuántas operaciones se ejecutan por el método `total` como una función de `values.length`
- ▶ Sea $N = \text{values.length}$
 - ▶ N se usa comúnmente como una variable que denota la cantidad de datos de entrada

Contando sentencias o declaraciones

- `int result = 0;` 2
- `int i = 0;` 2
- `i < values.length;` $N + 1$
- `i++` $2N$
- `result += values[i];` $3 * N$
- `return result;` 1
- $T(N) = 6N + 6$

- $T(N)$ es el número de operaciones ejecutadas en el método `total` en función de `values.length`

```
public int total(int[] values) {  
    int result = 0;  
    for (int i = 0; i < values.length; i++)  
        result += values[i];  
    return result;  
}
```

Otra simplificación

- Al determinar la complejidad de un algoritmo, queremos simplificar las cosas
 - ocultar algunos detalles para facilitar las comparaciones
- Ejemplo: ¿Cómo asignar tu calificación para los cursos de secundaria?
 - Al final del curso, su expediente académico no incluirá todos los detalles de su desempeño en el curso
 - no enumerará las puntuaciones de todas las tareas, cuestionarios y pruebas
 - simplemente una calificación de letra, A, B, C o D
- Entonces nos enfocamos en el término dominante de la función e ignoramos el coeficiente

Big O

- El método y la notación más común para discutir sobre el tiempo de ejecución de los algoritmos es *Big O*, también llamado *Order*
- Big O es el *tiempo de ejecución asintótico* del algoritmo
 - En otras palabras, ¿cómo crece el tiempo de ejecución del algoritmo en función de la cantidad de datos de entrada?
- Big O es un límite superior
- Es una herramienta matemática
- Oculta muchos detalles sin importancia asignando un grado (función) simple a los algoritmos

Análisis Teórico

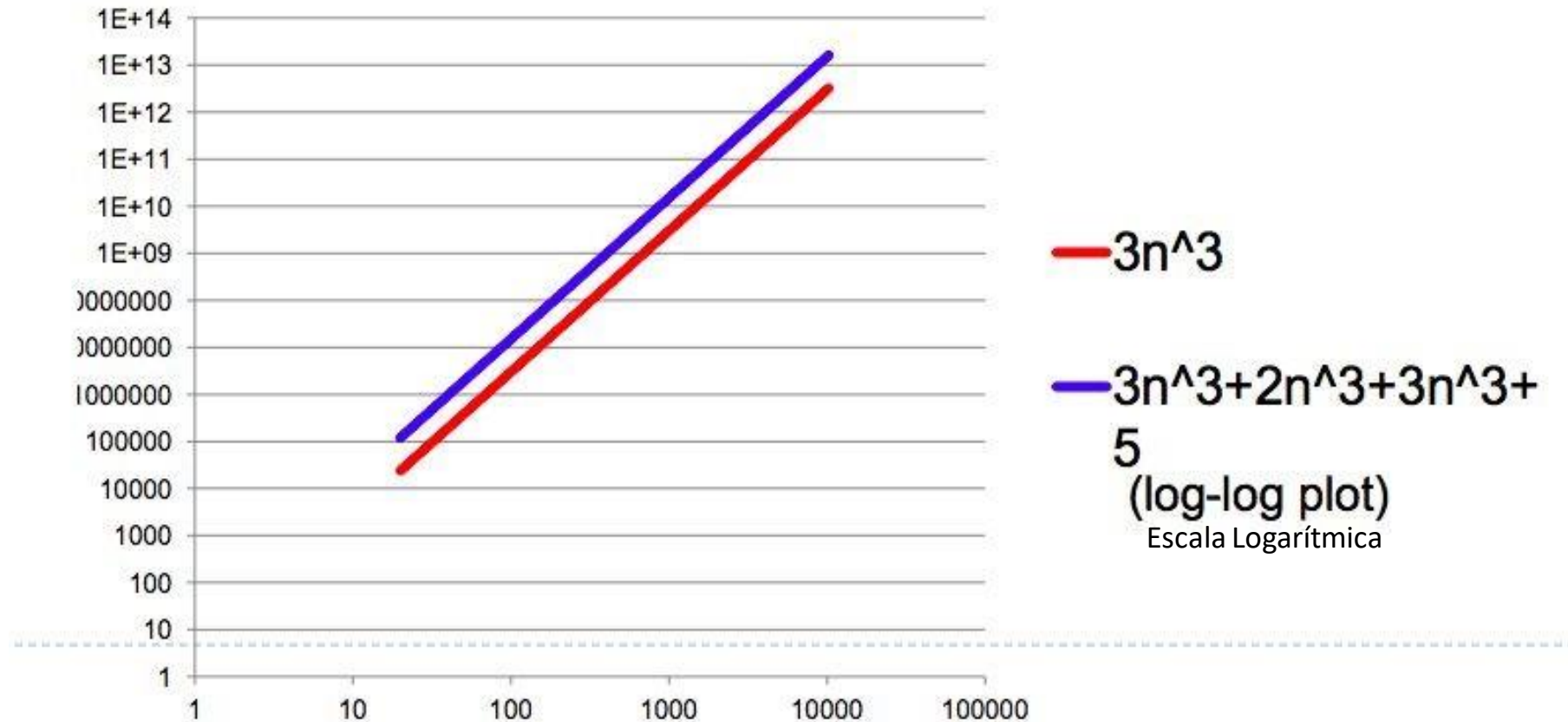
- Supón que tienes dos algoritmos, A y B, con las siguientes funciones de tiempo:
 - $T_A(n) = 3n^3$
 - $T_B(n) = 3n^3 + 2n^2 + 3n + 5$

¿Qué algoritmo es más eficiente?

Análisis Teórico

$$T_A(n) = 3n^3$$

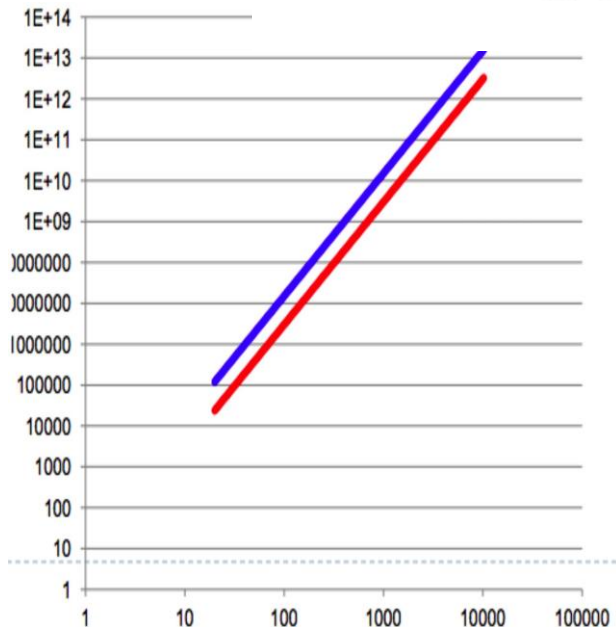
$$T_B(n) = 3n^3 + 2n^2 + 3n + 5$$



Análisis Teórico

- Dos funciones, f y g , son asintóticamente equivalentes cuando:

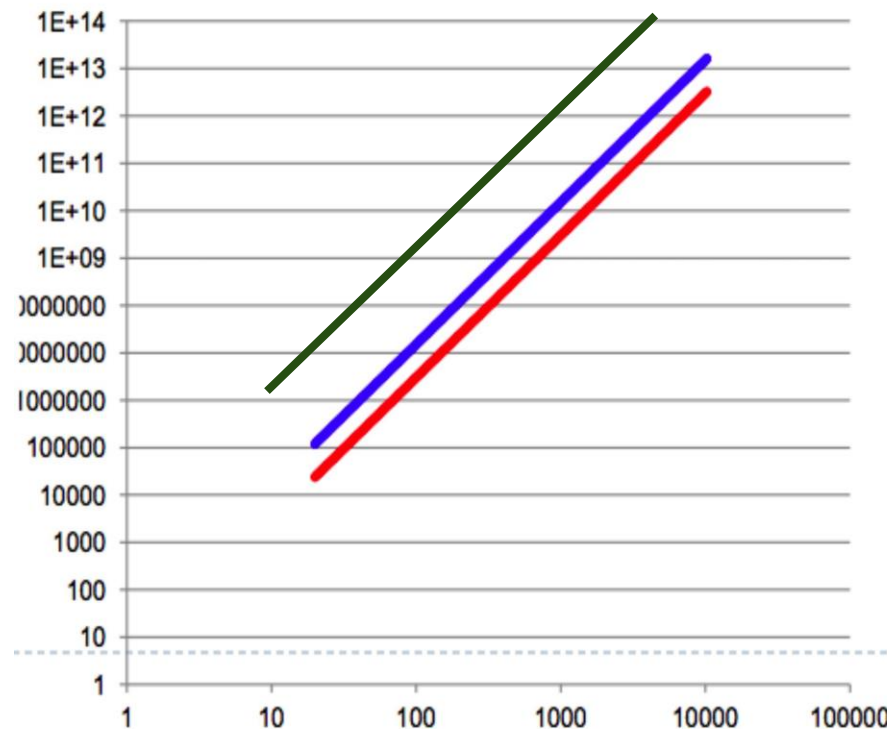
$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 1$$



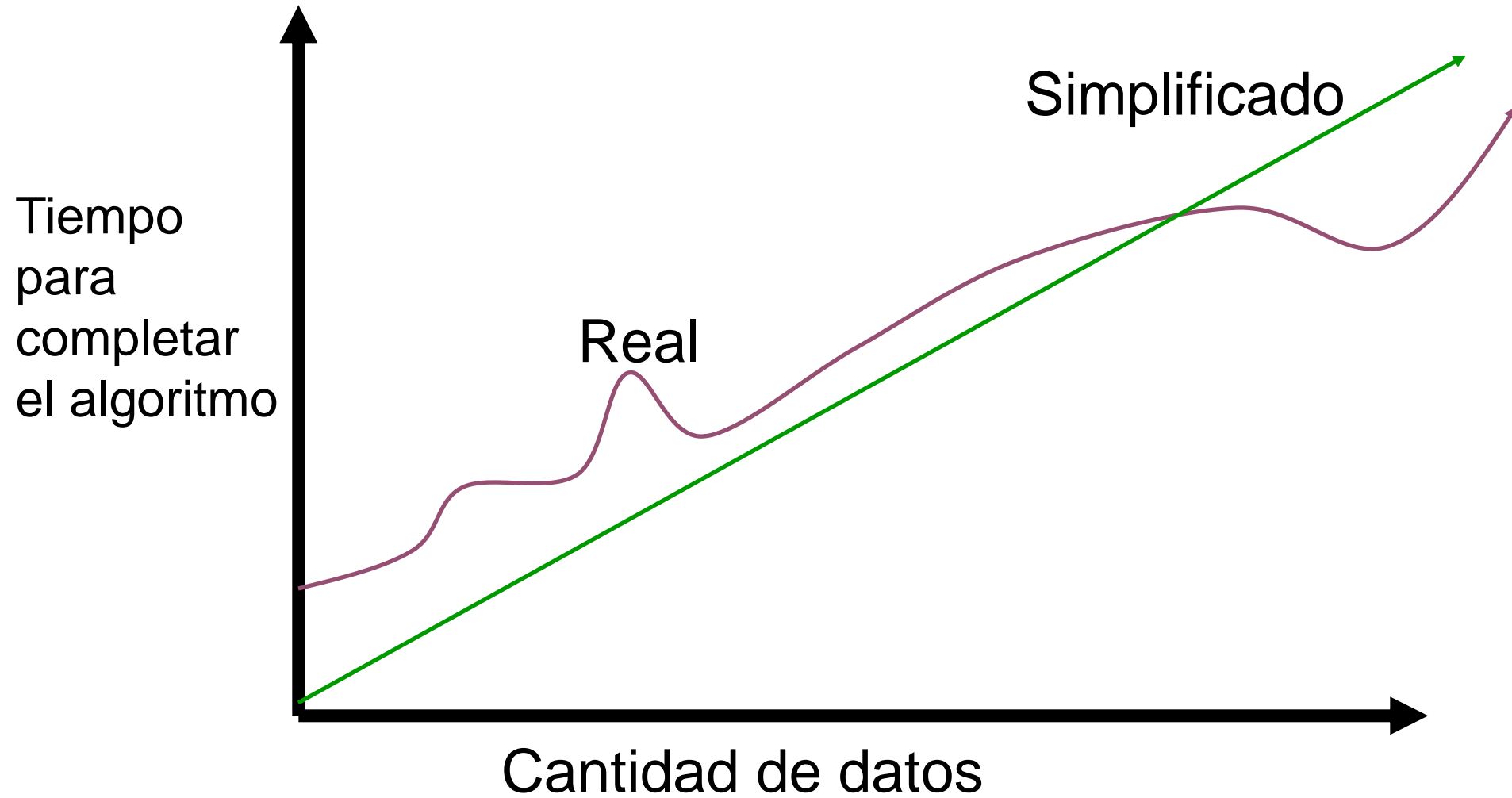
$$\lim_{n \rightarrow \infty} \frac{3n^3 + 2n^2 + 3n + 5}{3n^3} = 1$$

Análisis Teórico

- Tiempo de ejecución depende:
 - De la máquina en la que se ejecuta el programa.
 - Del compilador utilizado para generar el programa.
- Para facilitar la comparación de las funciones temporales, vamos a aproximar cada función temporal a una cota superior (análisis asintótico, Big-Oh)



Real vs. Big O



Análisis Teórico - cota superior asintótica

¿Cómo proponer una cota superior para una función $T(n)$

1. Buscar el término que crece más rápido (término de mayor grado).
2. Eliminar su coeficiente.

Análisis Teórico - cota superior asintótica

Buscar un límite superior a la función $T(n)$

- 1. Buscar el término que crece más rápido.**
2. Eliminar su coeficiente.

- $T_A(n)=3n^3 \rightarrow 3n^3$
- $T_B(n)=3n^3+2n^2+3n+5 \rightarrow 3n^3$

Análisis Teórico - cota superior asintótica

Buscar un límite superior a la función $T(n)$

1. Buscar el término que crece más rápido.
- 2. Eliminar su coeficiente.**

- $T_A(n)=3n^3 \rightarrow 3n^3 \rightarrow n^3$
- $T_B(n)=3n^3+2n^2+3n+5 \rightarrow 3n^3 \rightarrow n^3$

Cota superior también se llama función **Big O**

Análisis Teórico - cota superior asintótica

T(n)	BigO
4	O(?)
3n+4	O(?)
5n ² + 27n + 1005	O(?)
10n ³ + 2n ² + 7n + 1	O(?)
n!+ n ⁵	O(?)

Análisis Teórico - cota superior asintótica

$T(n)$	BigO
4	$O(1)$
$3n+4$	$O(n)$
$5n^2+ 27n + 1005$	$O(n^2)$
$10n^3+ 2n^2 + 7n + 1$	$O(n^3)$
$n!+ n^5$	$O(n!)$

Análisis Teórico - cota superior asintótica

$T(n)$	Big-O
$n + 2$	$O(?)$
$\frac{1}{2}(n+1)(n-1)$	$O(?)$
$7n^4+5n^2+1$	$O(?)$
$n(n-1)$	$O(?)$
$3n+\log(n)$	$O(?)$

Análisis Teórico - cota superior asintótica

$T(n)$	Big-O
$n + 2$	$O(n)$
$\frac{1}{2}(n+1)(n-1)$	$O(n^2)$
$7n^4+5n^2+1$	$O(n^4)$
$n(n-1)$	$O(n^2)$
$3n+\log(n)$	$O(n)$

Definición formal de Big O

- $T(N)$ es $O(F(N))$ si hay constantes positivas c y N_0 tales que $T(N) \leq cF(N)$ cuando $N \geq N_0$
 - N es el tamaño del conjunto de datos en el que trabaja el algoritmo
 - $T(N)$ es una función que caracteriza el tiempo de ejecución *real* del algoritmo
 - $F(N)$ es una función que caracteriza un límite superior en $T(N)$. Es un límite en el tiempo de ejecución del algoritmo. (La típica tabla de funciones Big)
 - c y N_0 son constantes

Notación asintótica (Big-Oh)

Sean $T(n)$ y $F(n) : \mathbb{N} \rightarrow \mathbb{R}$

$T(n)$ es $O(F(n))$ ssi existen c real con $c > 0$ y n_0 natural con $n_0 \geq 1$ tales que

$$T(n) \leq cF(n) \text{ para todo } n \geq n_0$$

“ $T(n)$ es $O(F(n))$ ” se lee “ $T(n)$ es big-oh de $F(n)$ ” o “ $T(n)$ es del orden de $F(n)$ ”

También se denota
como

$$T(n) = O(F(n))$$



Lo que significa

- $T(N)$ es la tasa de crecimiento real del algoritmo
 - Se puede igualar al número de sentencias ejecutables en un programa o fragmento de Código
- $F(N)$ es la función que señala los límites de la tasa de crecimiento
 - Puede ser el límite superior o inferior
- $T(N)$ puede no ser necesariamente igual a $F(N)$
 - Las constantes y términos menores son ignorados porque estamos hablando de una *función delimitadora*

Bucles que funcionan sobre un conjunto de datos

- ▶ El número de ejecuciones del ciclo depende de la longitud del arreglo, values.

```
public int total(int[] values)
{
    int result = 0;
    for(int i = 0; i < values.length; i++)
        result += values[i];
    return result;
}
```

- ▶ ¿Cuántas setencias se ejecutan con el método anterior?
- ▶ $N = \text{values.length}$. ¿Cuál es $T(N)$? $F(N)$? Big O?

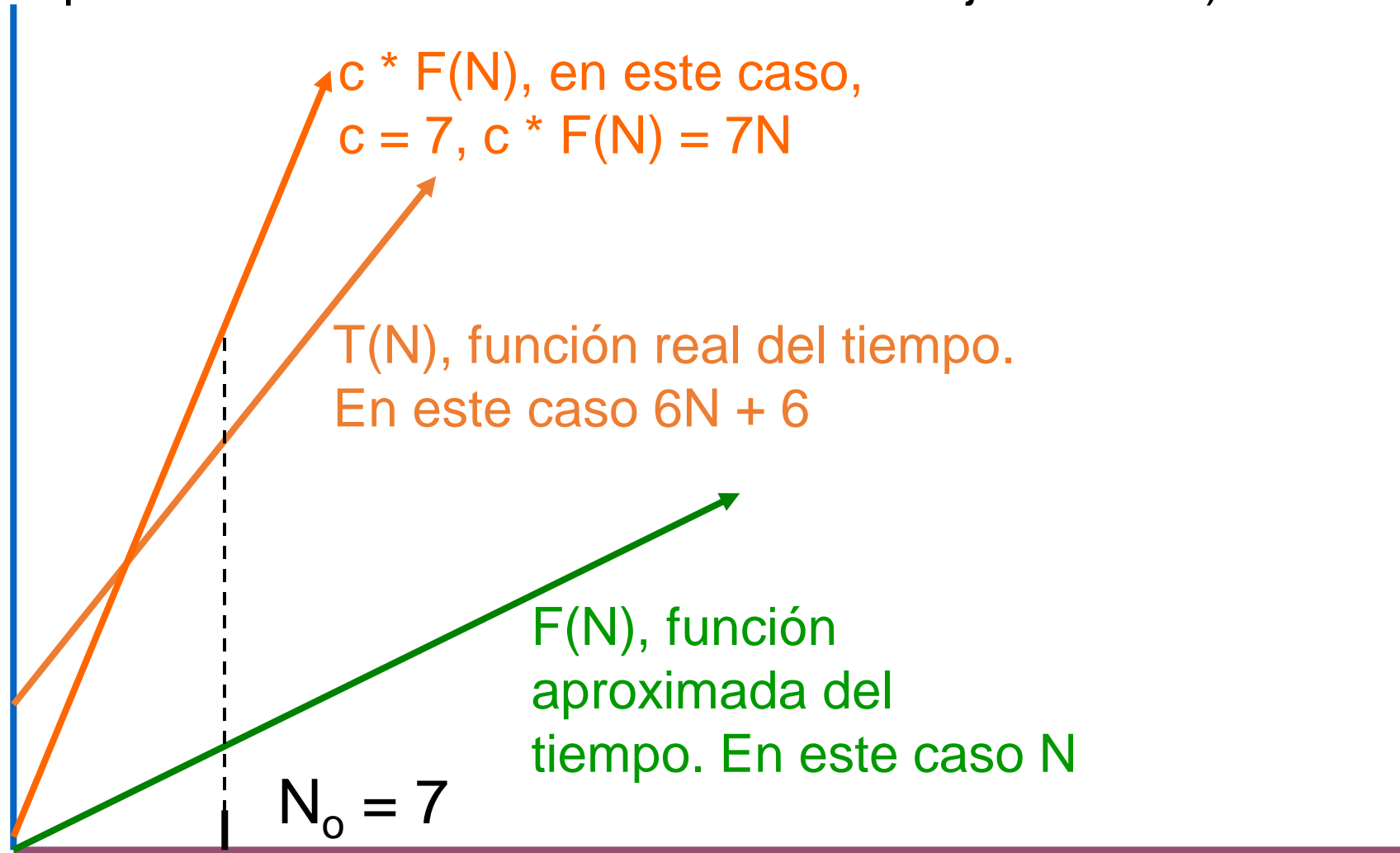
Contando sentencias o declaraciones

- `int result = 0;` 2 operaciones
- `int i = 0;` 2 operaciones
- `i < values.length;` $N + 1$ operaciones
- `i++` $2N$ operaciones
- `result += values[i];` $3N$ operaciones
- `return total;` 1 operación
- $T(N) = 6N + 6$
- $F(N) = N$
- Big O = $O(N)$

Demostrar que $O(N)$ es correcto

- Recuerde la definición formal de Big O
 - $T(N)$ es $O(F(N))$ si hay constantes positivas c y N_0 tales que $T(N) < cF(N)$ cuando $N > N_0$
- Analicemos el método `total`, $T(N) = 6N + 6$
 - Demuestre que el método `total` es $O(N)$.
 - $F(N)$ es N
- Necesitamos elegir las constantes c and N_0
- ¿Qué tal $c = 7$, $N_0 = 7$?

eje vertical: tiempo para que se complete el algoritmo.
(simplificado al número de declaraciones ejecutables)



eje horizontal: N , número de elementos en el conjunto de datos

Funciones Big O

- N es el tamaño del conjunto de datos.
- Las funciones no incluyen términos menos dominantes y no incluyen ningún coeficiente.
- $4N^2 + 10N - 100$ no es un $F(N)$ válido.
 - Simplemente sería $O(N^2)$
- Es posible tener dos variables independientes en la función Big O.
 - Ejemplo $O(M + \log N)$
 - M y N son los tamaños de dos conjuntos de datos diferentes pero que interactúan

Demostrar la notación Big O brevemente ...

- Demostrar que $10N^2 + 15N$ es $O(N^2)$
 - “Desmenuzar” la expression en varios términos.
 - $10N^2 \leq 10N^2$
 - $15N \leq 15N^2$ for $N \geq 1$ (ahora adicione)
 - $10N^2 + 15N \leq 10N^2 + 15N^2$ para $N \geq 1$
 - $10N^2 + 15N \leq 25N^2$ para $N \geq 1$
 - $c = 25, N_0 = 1$
 - Tenga en cuenta que las opciones para c y N_0 no son únicas.
- $T(N)$ es $O(F(N))$ si hay constantes positivas c y N_0 tales que $T(N) \leq cF(N)$ cuando $N \geq N_0$
 - N es el tamaño del conjunto de datos en el que trabaja el algoritmo
 - $T(N)$ es una función que caracteriza el tiempo de ejecución *real* del algoritmo
 - $F(N)$ es una función que caracteriza un límite superior en $T(N)$. Es un límite en el tiempo de ejecución del algoritmo. (La típica tabla de funciones Big)
 - c y N_0 son constantes

Ejemplo

Ejercicio: Mostrar que $3n^2+2n+5$ es $O(n^2)$

Proc: Hay que hallar c real y n_0 natural tal que
 $3n^2+2n+5 \leq cn^2$ para $n \geq n_0$.

$$3n^2 \leq 3n^2$$

$$2n \leq 2n^2$$

$$5 \leq 5n^2$$

$$\text{Luego } 3n^2+2n+5 \leq 3n^2 + 2n^2 + 5n^2 = (3+2+5)n^2 = 10n^2$$

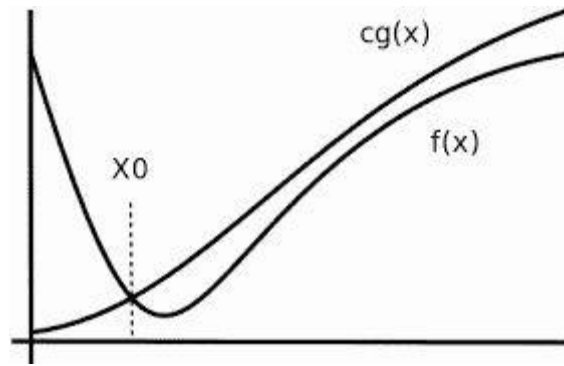
Por lo tanto: $c=10$

Para hallar n_0 resolver $10n^2 \geq 3n^2+2n+5$, lo que da $n \leq -5/7$ y $n \geq 1$.

Luego $n_0=1$.

Análisis Teórico - cota superior asintótica

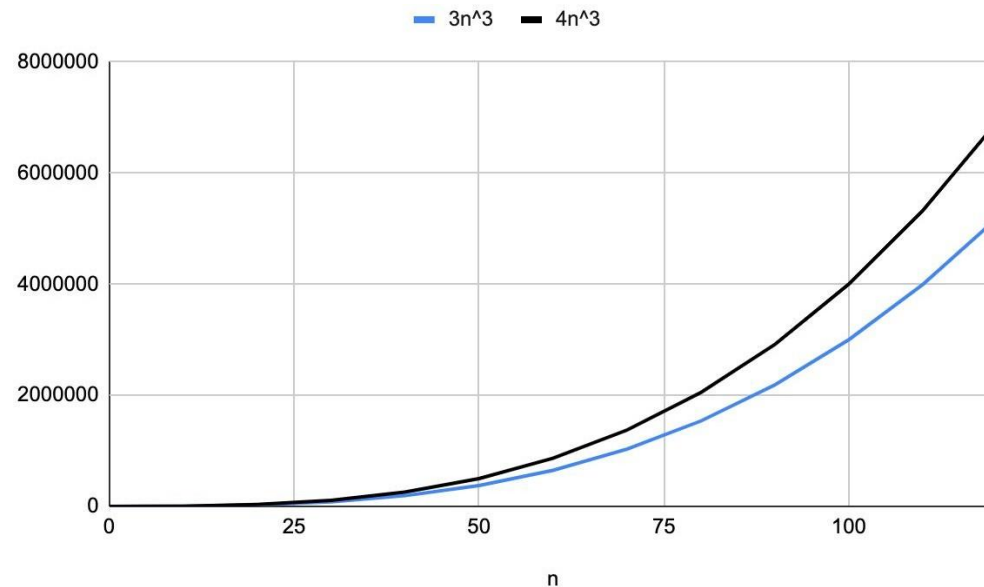
- Dadas dos funciones, $f(n)$ y $g(n)$, $f(n)$ es de **orden superior** $g(n)$, si existen $n_0 > 0$ y $c > 0$, se cumple:
 $f(n) \leq cg(n)$, para todo $n \geq n_0$



https://es.wikipedia.org/wiki/Cota_superior_asintótica

Análisis Teórico - cota superior asintótica

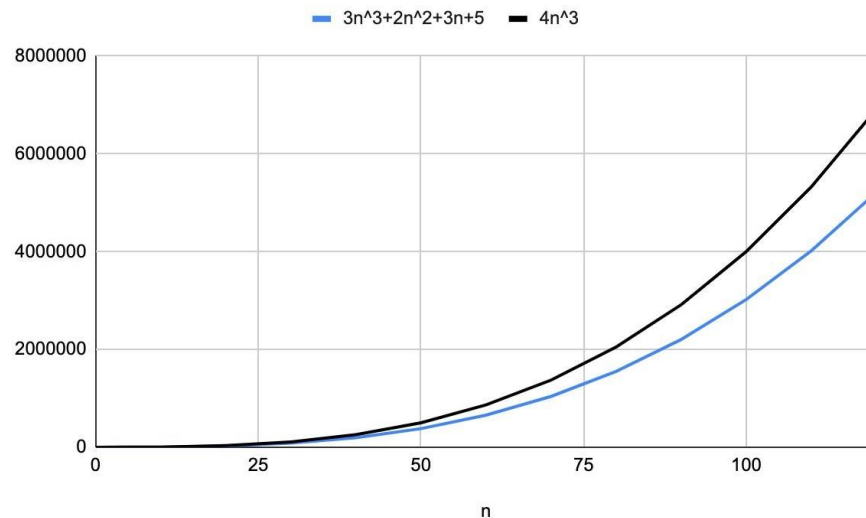
$T_A(n) = 3n^3$ es de orden superior n^3 porque **existen** $n_0=0, c=4$, tales que $T_A(n) \leq cn^3$, para todo $n \geq n_0$



n	3n^3	4n^3
0	0	0
5	375	500
10	3000	4000
20	24000	32000
30	81000	108000
40	192000	256000
50	375000	500000
60	648000	864000
70	1029000	1372000
80	1536000	2048000
90	2187000	2916000
100	3000000	4000000
110	3993000	5324000
120	5184000	6912000

Análisis Teórico - cota superior asintótica

$T_B(n) = 3n^3 + 2n^2 + 3n + 5$ es de orden superior n^3 porque existen $n_0 = 10$, $c = 4$, tales que $T_B(n) \leq cn^3$, para todo $n \geq n_0$



n	$3n^3 + 2n^2 + 3n + 5$	$4n^3$
0	5	0
5	445	500
10	3235	4000
20	24865	32000
30	82895	108000
40	195325	256000
50	380155	500000
60	655385	864000
70	1039015	1372000
80	1549045	2048000
90	2203475	2916000
100	3020305	4000000
110	4017535	5324000
120	5213165	6912000

Análisis Teórico - cota superior asintótica

- Buenas noticias!!!: Un conjunto pequeño de instrucciones:

$$1 < \log n < n < n \log n < n^2 < n^3 < \dots < 2^n < n!$$

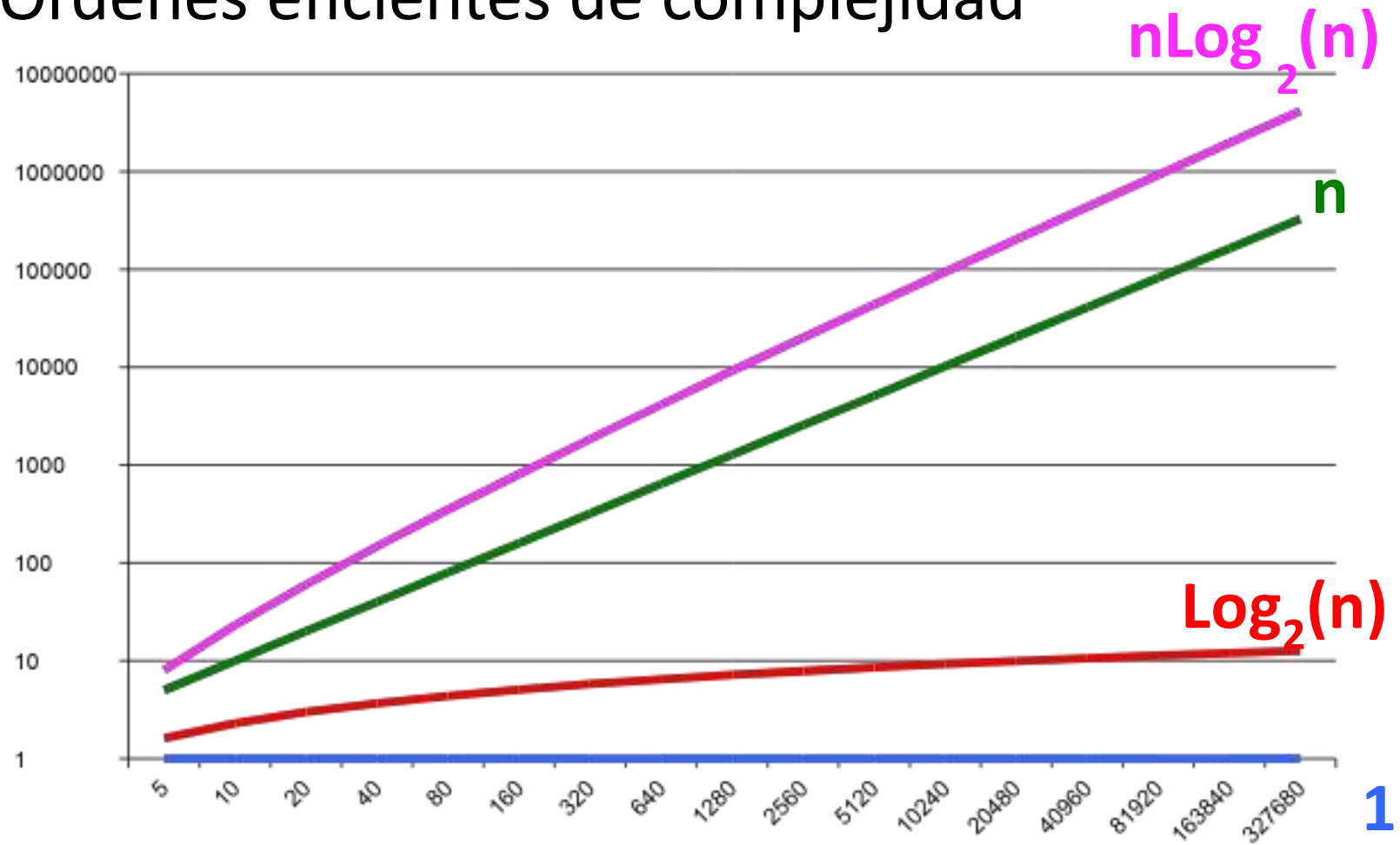
Análisis Teórico - cota superior asintótica

Big O funciones

notación	nombre
$O(1)$	constante
$O(\log n)$	logarítmico
$O(n)$	lineal
$O(n \log n)$	lineal-logarítmico
$O(n^2)$	cuadrática
$O(n^c)$	polinómico
$O(c^n)$	exponencial
$O(n!)$	factorial

Análisis Teórico - Big O eficientes

Órdenes eficientes de complejidad

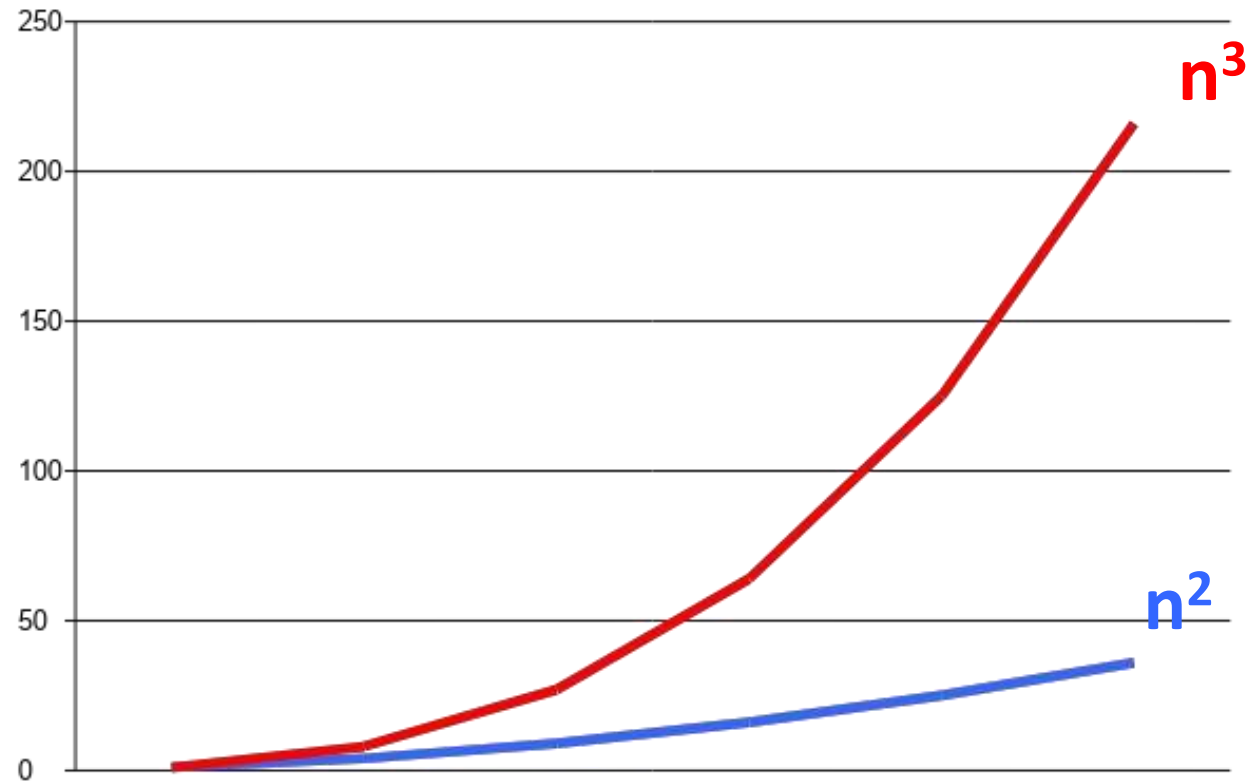


Análisis Teórico - Big O eficientes

Orden	Nombre	Descripción	Ejemplo
1	Constante	Independiente del tamaño	Borrar el primer elemento de una cola
$\text{Log}_2(n)$	Logarítmico	Dividir por la mitad	Búsqueda binaria
n	Lineal	Bucle	Suma de los elementos de una lista
$n\text{Log}_2(n)$	Lineal-logarítmico	Divide y Vencerás	Mergesort, quicksort

Análisis Teórico - Big O tratables

Órdenes tratables de complejidad

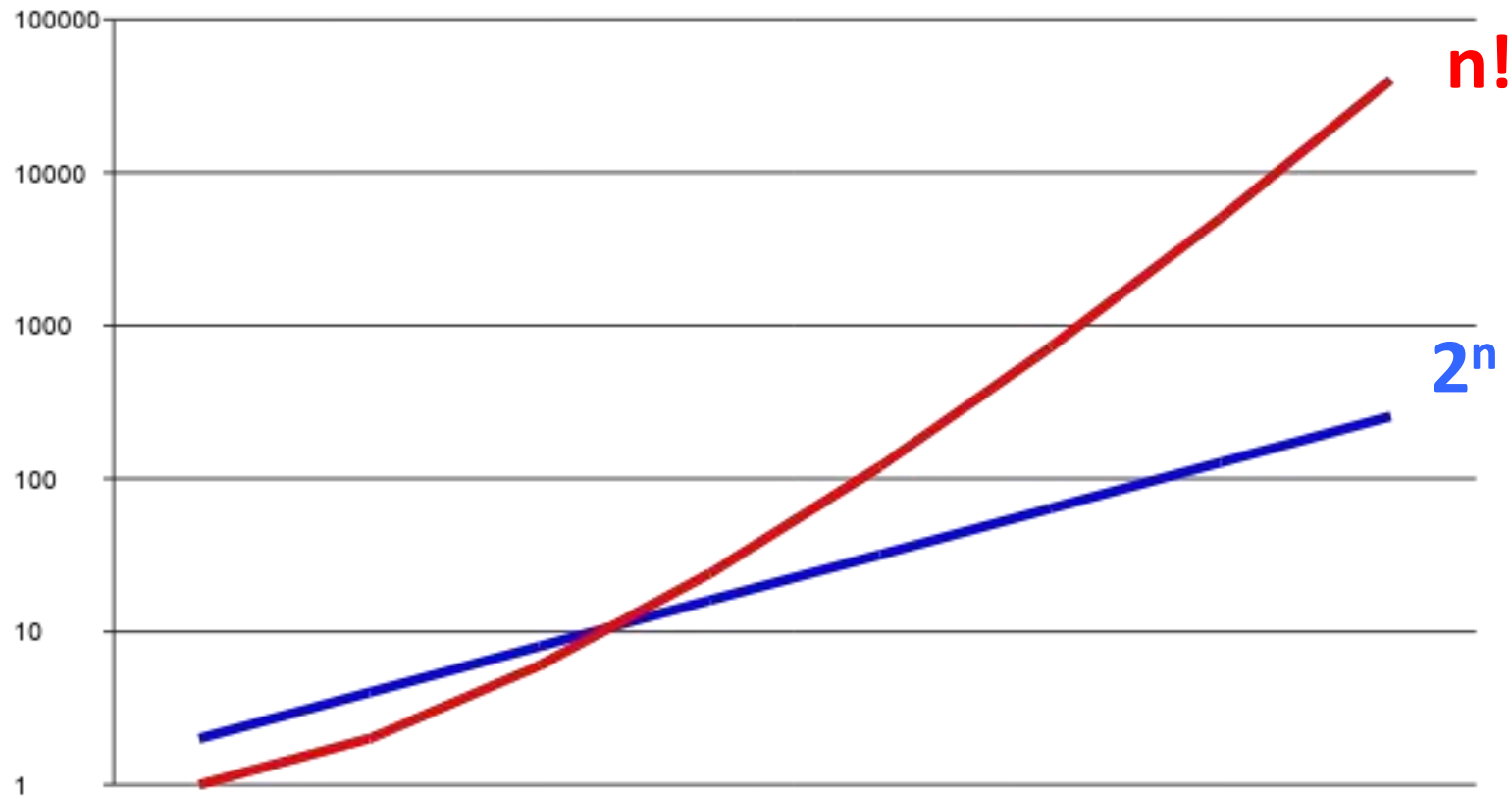


Análisis Teórico - Big O tratables

Orden	Nombre	Descripción	Ejemplo
n^2	Cuadrático	Bucles dobles	Sumar dos matrices; método burbuja para ordenar
n^3	Cúbico	Bucles triples	Multiplicar dos matrices

Análisis Teórico - Big O intratables

Órdenes intratables de complejidad:



Análisis Teórico - Big O intratables

Orden	Nombre	Descripción	Ejemplo
k^n	Exponencial	Búsqueda exhaustiva	Adivinar una password
$n!$	Factorial	Búsqueda fuerza bruta	Enumerar todas las posibles particiones de un conjunto

Típicas funciones Big O – "Grados"

Función	Nombre Común
$N!$	Factorial
2^N	Exponencial
$N^d, d > 3$	Polinomial
N^3	Cubica
N^2	Cuadrática
$N\sqrt{N}$	N raíz cuadrada N
$N \log N$	$N \log N$
N	Lineal
\sqrt{N}	Raíz – n
$\log N$	Logaritmica
1	Constante



El tiempo de ejecución crece 'rápidamente' con una entrada más grande

El tiempo de ejecución crece 'lentamente' con una entrada más grande

Pregunta 7

■ Marque la opción verdadera

```
public int total(int[] values)
{
    int result = 0;
    for(int i = 0; i < values.length; i++)
        result += values[i];
    return result;
}
```

- A. El método `total` es $O(N^{1/2})$
- B. El método `total` es $O(N)$
- C. El método `total` es $O(N^2)$
- D. Dos de las opciones anteriores son correctas
- E. A, B, C son correctas

Más sobre la definición formal

- Hay un punto N_0 tal que para todos los valores de N que superan este punto, $T(N)$ está acotado por algún múltiplo de $F(N)$
- Por lo tanto, si $T(N)$ del algoritmo es $O(N^2)$, ignorando las constantes, en algún punto podemos *limitar* el tiempo de ejecución mediante una función cuadrática
- dado un algoritmo *lineal*, es *técnicamente correcto* decir que el tiempo de ejecución es $O(N^2)$. $O(N)$ es una respuesta más precisa en cuanto al Big O del algoritmo lineal
 - Por ello, la advertencia de "elegir la función más restrictiva" en las preguntas de tipo Big O..

Tratar con otros métodos

- ¿Qué hago con las llamadas a los métodos?

```
double sum = 0.0;  
for (int i = 0; i < n; i++)  
    sum += Math.sqrt(i);
```

- Camino largo

- ir a ese método o constructor y contar las sentencias o declaraciones

- Camino corto

- Sustituya la función simplificada Big O por ese método.
- **Si** `Math.sqrt` es de tiempo constante, $O(1)$, simplemente cuente `sum += Math.sqrt(i);` como una sentencia o declaración.

Tratar con otros métodos

```
public int foo(int[] data) {  
    int total = 0;  
    for (int i = 0; i < data.length; i++)  
        total += countDups(data[i], data);  
    return total;  
}  
// el método countDups es O(N) donde N es la  
// longitud del arreglo que se pasa
```

¿Cuál es la notación Big O de foo?

- A. $O(1)$ B. $O(N)$ C. $O(N \log N)$
D. $O(N^2)$ E. $O(N!)$

Bucles independientes

// de la clase Matrix

```
public void scale(int factor) {  
    for (int r = 0; r < numRows(); r++)  
        for (int c = 0; c < numCols(); c++)  
            iCells[r][c] *= factor;  
}
```

Asuma que `numRows() = numCols() = N`.

En otras palabras una matriz cuadrada.

`numRows` y `numCols` son $O(1)$

¿Cuál es el $T(N)$?, ¿Cuál es la notación Big O?

- A. $O(1)$
- B. $O(N)$
- C. $O(N \log N)$
- D. $O(N^2)$
- E. $O(N!)$

Pregunta extra. ¿Qué pasa si `numRows` es $O(N)$?

Sólo contar bucles, verdad?

```
// asumimos que mat es una matriz 2d de valores booleanos
// asumimos que mat es cuadrada con N Filas,
// y N columnas
public static void count(boolean[][] mat,
                          int row, int col) {
    int numThings = 0;
    for (int r = row - 1; r <= row + 1; r++)
        for (int c = col - 1; c <= col + 1; c++)
            if (mat[r][c])
                numThings++;
}
```

¿Cuál es el orden (Notación Big O) del método anterior?

- A. $O(1)$ B. $O(N^{0.5})$ C. $O(N)$ D. $O(N^2)$ E. $O(N^3)$

No se trata solo de contar bucles

// El ejemplo de la diapositiva anterior podría ser
// reescrito de la siguiente manera:

```
int numThings = 0;
if (mat[r-1][c-1]) numThings++;
if (mat[r-1][c]) numThings++;
if (mat[r-1][c+1]) numThings++;
if (mat[r][c-1]) numThings++;
if (mat[r][c]) numThings++;
if (mat[r][c+1]) numThings++;
if (mat[r+1][c-1]) numThings++;
if (mat[r+1][c]) numThings++;
if (mat[r+1][c+1]) numThings++;
```

Un punto aparte, el logaritmo

- $3^2 = 9$
- Igualmente $\log_3 9 = 2$
 - “El log en base 3 de 9 es 2.”
- La forma de pensar acerca del logaritmo es:
 - “El log en base x de y es el número al que puede elevar x para obtener y.”
 - Dígase a si mismo “El log es el exponente.” (y repítelo una y otra vez hasta que lo crea)
 - Trabajaremos más con logaritmos en base 2
- $\log_2 32 = ?$ $\log_2 8 = ?$ $\log_2 1024 = ?$ $\log_{10} 1000 = ?$

¿Cuándo ocurren los Logaritmos?

- Los algoritmos tienden a tener un término logarítmico cuando usan la técnica divide y vencerás (la cual estudiaremos más adelante)
- El tamaño del conjunto de datos se obtiene dividiendo entre 2

```
public int foo(int n) {  
    // pre n > 0  
    int total = 0;  
    while (n > 0) {  
        n = n / 2;  
        total++;  
    }  
    return total;  
}
```

- ¿Cuál es el orden (notación Big O) del código anterior?

- A. $O(1)$ B. $O(\log N)$ C. $O(N)$
D. $O(N \log N)$ E. $O(N^2)$

Ejercicio

Demostrar que la búsqueda binaria en un arreglo de n componentes ordenados en forma ascendente tiene un orden logarítmico de base 2 de la cantidad de elementos del arreglo $= O(\log_2 n)$.

```
public static int bsearch( int [] a, int n, int x )
{ int ini = 0, fin = n-1;
  while( ini <= fin ) {
    int medio = (ini + fin) / 2;
    if( a[medio] == x ) return medio;
    else if( a[medio] > x ) fin = medio-1;
    else ini = medio + 1;
  }
  return -1;
}
```

Ejercicio

Tamaño de la entrada: n = cantidad de componentes de a
Peor caso: x no está en a

```
public static int bsearch( int [] a, int n, int x ) {  
    int ini = 0, fin = n-1;            $c_1$   
    while( ini <= fin ) {             Tiempo de la condición:  $c_2$   
        int medio = (ini + fin) / 2;   Tiempo del cuerpo:  $c_3$   
        if( a[medio] == x ) return medio;  
        else if( a[medio] > x ) fin = medio-1;  
        else ini = medio + 1;  
    }  
    return -1;                         $c_4$   
}
```

Sea k = cantidad de iteraciones del while, entonces

$$T(n) = c_1 + k(c_2 + c_3) + c_2 + c_4.$$

La pregunta es cómo definir k en función de n .

Ejercicio

¿Cómo estimar k en función de n?

El peor caso es cuando “x” no está en “a”.

Veamos cómo vamos descartando componentes del arreglo en función del número de iteración del while: Si tenemos n componentes, en cada iteración se descarta la componente del medio del arreglo, entonces de las n-1 componentes que falta revisar sólo se va considerar la mitad, entonces quedan $(n-1)/2$ componentes para la siguiente iteración, y así sucesivamente.

Calculemos cuál es caso para la iteración genérica k.

Número de iteración	Componentes por revisar
1	n
2	$(n-1)/2$
3	$(n-3)/4$
4	$(n-7)/8$
5	$(n-15)/16$
...	...
k	$\frac{n - (2^{k-1} - 1)}{2^{k-1}}$

Ejercicio

Entonces vimos que en la iteración k , la cantidad de componentes que quedan por revisar es:

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

Como x no está en el arreglo a , en la última iteración completa que realiza el while queda una componente del arreglo por revisar.

Entonces:

$$\frac{n - (2^{k-1} - 1)}{2^{k-1}} = 1;$$

$$n - (2^{k-1} - 1) = 2^{k-1};$$

$$n - 2^{k-1} + 1 = 2^{k-1};$$

$$n + 1 = 2^{k-1} + 2^{k-1};$$

$$n + 1 = 2 \times 2^{k-1};$$

$$n + 1 = 2^k;$$

$$\log_2(n + 1) = k.$$

Con lo que $T(n) = c_1 + \log_2(n+1)(c_2+c_3) + c_2 + c_4$. Luego, el algoritmo es $O(\log_2(n))$, lo que se quería demostrar

Mejora significativa – Algoritmo con una función Big O más pequeña

- Problema: Dada un arreglo de enteros, reemplace cualquier elemento igual a 0 con el valor positivo máximo a la derecha de ese elemento. (si no hay un valor positivo a la derecha, déjelo sin cambios)

Dado:

[0, 9, 0, 13, 0, 0, 7, 1, -1, 0, 1, 0]

Se convierte en:

[13, 9, 13, 13, 7, 7, 7, 1, -1, 1, 1, 0]

Reemplazar ceros: solución típica

```
public void replace0s(int[] data) {  
    for(int i = 0; i < data.length; i++) {  
        if (data[i] == 0) {  
            int max = 0;  
            for(int j = i+1; j < data.length; j++)  
                max = Math.max(max, data[j]);  
            data[i] = max;  
        }  
    }  
}
```

Asuma que todos los valores son ceros

Ejemplo de **bucles dependientes**.

Big O de este enfoque?

- | | | |
|-------------|------------|------------------|
| A. $O(1)$ | B. $O(N)$ | C. $O(N \log N)$ |
| D. $O(N^2)$ | E. $O(N!)$ | |

Reemplazar ceros: solución alternativa

```
public void replace0s(int[] data) {  
    int max =  
        Math.max(0, data[data.length - 1]);  
    int start = data.length - 2;  
    for (int i = start; i >= 0; i--) {  
        if (data[i] == 0)  
            data[i] = max;  
        else  
            max = Math.max(max, data[i]);  
    }  
}
```

Big O de este enfoque?

A. $O(1)$

B. $O(N)$

C. $O(N \log N)$

D. $O(N^2)$

E. $O(N!)$

Pregunta 8

- Es $O(N)$ realmente mucho más rápido que $O(N^2)$?

A. Nunca

B. Siempre

C. Típicamente

- Depende de las funciones reales y del valor de N .
- $1000N + 250$ comparado con $N^2 + 10$
- ¿Cuándo utilizamos el cálculo automático?
- $N = 100,000$
- $100,000,250 < 10,000,000,010$ ($10^8 < 10^{10}$)

Una proporción útil

- Dado que $F(N)$ caracteriza el tiempo de ejecución de un algoritmo, la siguiente proporción debería ser cierta:

$$F(N_0) / F(N_1) \sim \text{tiempo}_0 / \text{tiempo}_1$$

- Un algoritmo que es $O(N^2)$ toma 3 segundos en ejecutarse dado 10,000 datos.
 - ¿Cuánto tiempo esperas que tome su ejecución cuando hay 30,000 datos?
 - Error común
 - ¿logaritmos?

¿Porqué utilizar Big O?

- A medida que construimos estructuras de datos, Big O es la herramienta que usaremos para decidir bajo qué condiciones una estructura de datos es mejor que otra
- Piense en el rendimiento cuando hay muchos datos.
 - "Funcionó tan bien con pequeños conjuntos de datos ..."
 - [Joel Spolsky, Schlemiel the painter's Algorithm](#)
- Muchas compensaciones (trade offs)
 - algunas estructuras de datos son buenas para ciertos tipos de problemas
 - A menudo compensamos ESPACIO por TIEMPO.
 - Solución más rápida que ocupa más espacio
 - Solución más lenta que ocupa menos espacio

Espacio Big O

- Big O podría usarse para especificar cuánto espacio se necesita para un algoritmo en particular
 - en otras palabras, cuántas variables se necesitan
- A menudo hay una *compensación entre el tiempo y el espacio*
 - a menudo puede tomar menos tiempo si está dispuesto a usar más memoria
 - a menudo puede usar menos memoria si está dispuesto a tomar más tiempo
 - las soluciones verdaderamente hermosas toman menos tiempo y espacio

La mayor diferencia entre el tiempo y el espacio es que no se puede reutilizar el tiempo. - Merrick Furst

Cuantificadores Big O

- A menudo es útil discutir sobre diferentes casos para un algoritmo
- Mejor caso: ¿qué es lo mejor que podemos esperar?
 - Menos interesante
- Caso promedio (también conocido como tiempo de ejecución esperado): ¿qué sucede normalmente con el algoritmo?
- Peor caso: ¿qué es lo peor que podemos esperar del algoritmo?
 - muy interesante comparar esto con el caso promedio

Mejor caso, caso promedio, peor caso

- Para determinar el mejor Big O , el Big O promedio y el big O para el peor de los casos, debemos hacer suposiciones sobre el conjunto de datos
- Mejor caso -> ¿Cuáles son las propiedades del conjunto de datos que conducirán a la menor cantidad de sentencias ejecutables (pasos en el algoritmo)
- Peor caso -> ¿Cuáles son las propiedades del conjunto de datos que conducirán a la mayor cantidad de sentencias ejecutables?
- Caso promedio -> Por lo general, esto significa asumir que los datos se distribuyen aleatoriamente
 - o si ejecuté el algoritmo una gran cantidad de veces con diferentes conjuntos de datos, ¿cuál sería la cantidad promedio de trabajo para esas ejecuciones?

Análisis Teórico - Mejor y peor caso

- **Mejor caso:** el caso que requiere el menor número de operaciones.
- **Peor caso:** el caso que requiere el mayor número de operaciones.

Análisis Teórico - Caso medio

- **Caso medio:** representa el número medio de operaciones para ser ejecutadas.
- Para conocer este número medio, debemos tomar todas las posibles entradas y calcular sus número de operaciones.
- El análisis del caso medio no es fácil de estimar en la mayoría de los casos.

Análisis Teórico - siempre el peor caso

- Para **analizar un algoritmos**, siempre lo haremos en función de su **peor caso**.
- De esta forma, garantizamos un límite superior para todas las funciones temporales del algoritmo.

Análisis Teórico - Mejor y peor caso

```
Algorithm contains(data: list, x: int) -> int:  
    for c in data:  
        if c==x:  
            return True  
  
    return False
```

- Mejor caso?
- Peor caso?

Análisis Teórico - Mejor y peor caso

```
Algorithm contains(data: list, x: int) -> int:  
    for c in data:  
        if c==x:  
            return True  
  
    return False
```

- Como mejor caso podemos considerar:
 - data es None o está vacío.
 - el primer elemento de data es x

Análisis Teórico - Mejor y peor caso

```
Algorithm contains(data: list, x: int) -> int:  
    for c in data:  
        if c==x:  
            return True  
  
    return False
```

- Como peor caso:
 - x no está en la lista o es el último elemento (tendremos que recorrer la lista completa)

Análisis Teórico - Mejor y peor caso

- En algunos algoritmos, todos los casos son computacionalmente igual de costosos.
- En el siguiente algoritmo, **no existe peor ni mejor caso** (suponiendo que data nunca está vacío), porque en todos los casos siempre es necesario recorrer toda la lista.

```
Algorithm sum_list(data: list) -> int:  
    total=0  
    for c in data:  
        total = total + c  
    return total
```

Análisis Teórico - Ejercicio

- Calcula su función temporal ($T(n)$) y su orden de complejidad (BigO). Discute sobre su mejor y peor caso (suponiendo que data no está vacía).

Algorithm *findMax*(data: int) -> int:

max=-999999

for c **in** data:

if c>max **then**

 max=c

return max

Análisis Teórico - Ejercicio

```
Algorithm findMax(data: int) -> int:
    max=-999999                      # 1
    for c in data:
        if c>max then                # n * (1+1)
            max=c
    return max                       # 1
```

- La función temporal $T(n) = 2n + 2$
- Su cota superior es $O(n) = n$
- No hay mejor ni peor caso porque siempre es necesario recorrer toda la lista para encontrar el máximo.

Análisis Teórico - Ejercicio

```
Algorithm first_even(data: list) -> int
  for c in data:
    if c%2==0 then
      return c
  return None
```

- La función temporal $T(n) = n + 1$
- Su cota superior es $O(n) = n$
- Mejor caso: el primer elemento es par (o lista vacía)
- Peor caso: el primer par es el último elemento o no existe ningún par.

Otro ejemplo

```
public double minimum(double[] values) {  
    int n = values.length;  
    double minValue = values[0];  
    for (int i = 1; i < n; i++)  
        if (values[i] < minValue)  
            minValue = values[i];  
    return minValue;  
}
```

- T(N)? F(N)? Big O? ¿Mejor caso? ¿Peor caso? ¿Caso promedio?
- Si no hay otra información, suponga que se pregunta por el caso promedio

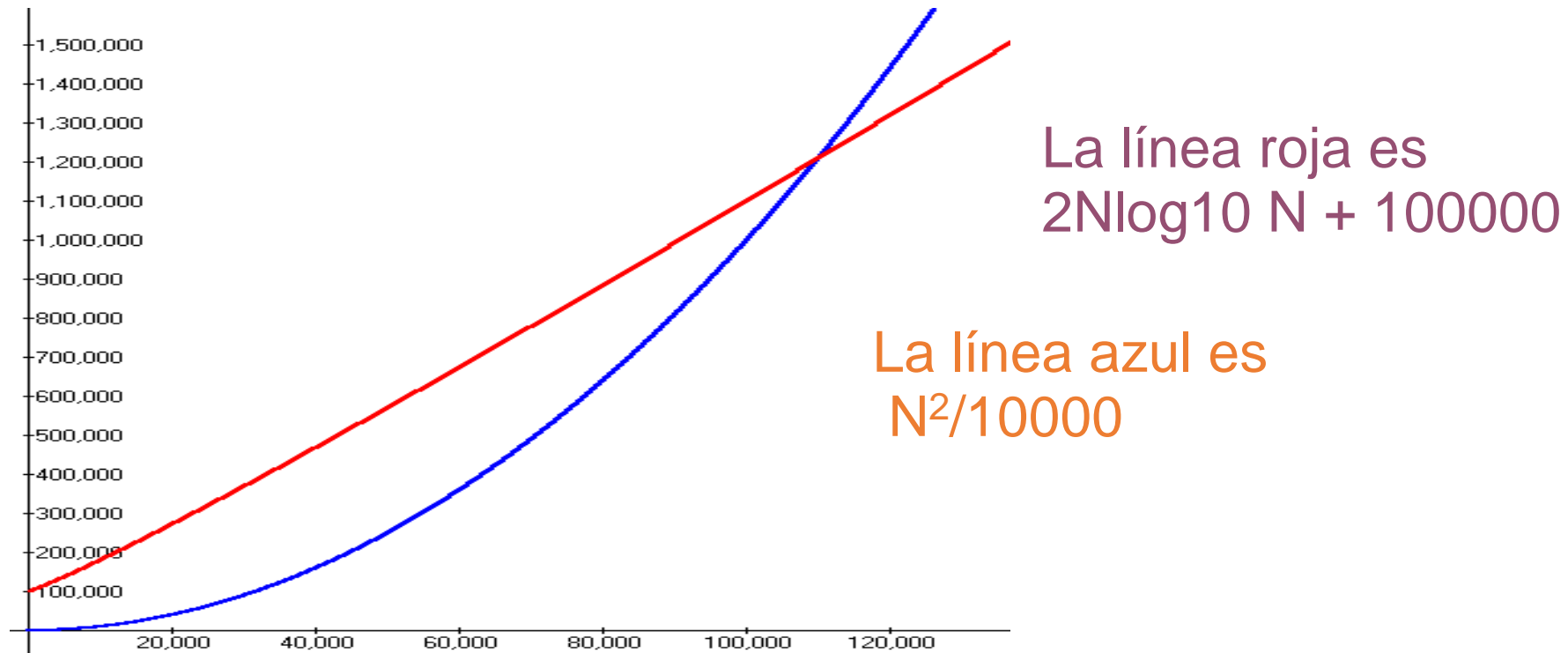
Ejemplo de dominancia

- Mira un ejemplo extremo. Suponga que el número real en función de la cantidad de datos es:

$$N^2/10000 + 2N\log_{10} N + 100000$$

- ¿Es plausible decir que el término N^2 domina a pesar de que está dividido por 10000 y que el algoritmo es $O(N^2)$?
- ¿Qué pasa si separamos la ecuación en $(N^2/10000)$ y $(2N\log_{10} N + 100000)$ y graficamos los resultados ?

Sumar tiempos de ejecución



- Para grandes valores de N el término N^2 predomina así que el algoritmo es $O(N^2)$
- ¿Cuándo tiene sentido usar una computadora?

Comparación de grados

- Asuma que tenemos un problema
- El algoritmo A resuelve el problema correctamente y es $O(N^2)$
- El algoritmo B resuelve el mismo problema correctamente y es $O(N \log_2 N)$
- ¿Cuál algoritmo es más rápido?
- Una de las suposiciones de Big O es que el conjunto de datos es grande.
- Los "grados" deben ser herramientas precisas si esto es cierto

Tiempos de ejecución

- Suponga que $N = 100.000$ y la velocidad del procesador es 1.000.000.000 de operaciones por segundo

Función	Tiempo de ejecución
2^N	$3.2 \times 10^{30,086}$ años
N^4	3171 años
N^3	11.6 días
N^2	10 segundos
$N\sqrt{N}$	0.032 segundos
$N \log N$	0.0017 segundos
\sqrt{N}	0.0001 segundos
N	3.2×10^{-7} segundos
$\log N$	1.2×10^{-8} segundos

Tiempos de ejecución

	1000	2000	4000	8000	16000	32000	64000	128K
$O(N)$	2.2×10^{-5}	2.7×10^{-5}	5.4×10^{-5}	4.2×10^{-5}	6.8×10^{-5}	1.2×10^{-4}	2.3×10^{-4}	5.1×10^{-4}
$O(N \log N)$	8.5×10^{-5}	1.9×10^{-4}	3.7×10^{-4}	4.7×10^{-4}	1.0×10^{-3}	2.1×10^{-3}	4.6×10^{-3}	1.2×10^{-2}
$O(N^{3/2})$	3.5×10^{-5}	6.9×10^{-4}	1.7×10^{-3}	5.0×10^{-3}	1.4×10^{-2}	3.8×10^{-2}	0.11	0.30
$O(N^2)$ ind.	3.4×10^{-3}	1.4×10^{-3}	4.4×10^{-3}	0.22	0.86	3.45	13.79	(55)
$O(N^2)$ dep.	1.8×10^{-3}	7.1×10^{-3}	2.7×10^{-2}	0.11	0.43	1.73	6.90	(27.6)
$O(N^3)$	3.40	27.26	(218)	(1745) 29 min.	(13,957) 233 min	(112k) 31 hrs	(896k) 10 days	(7.2m) 80 days

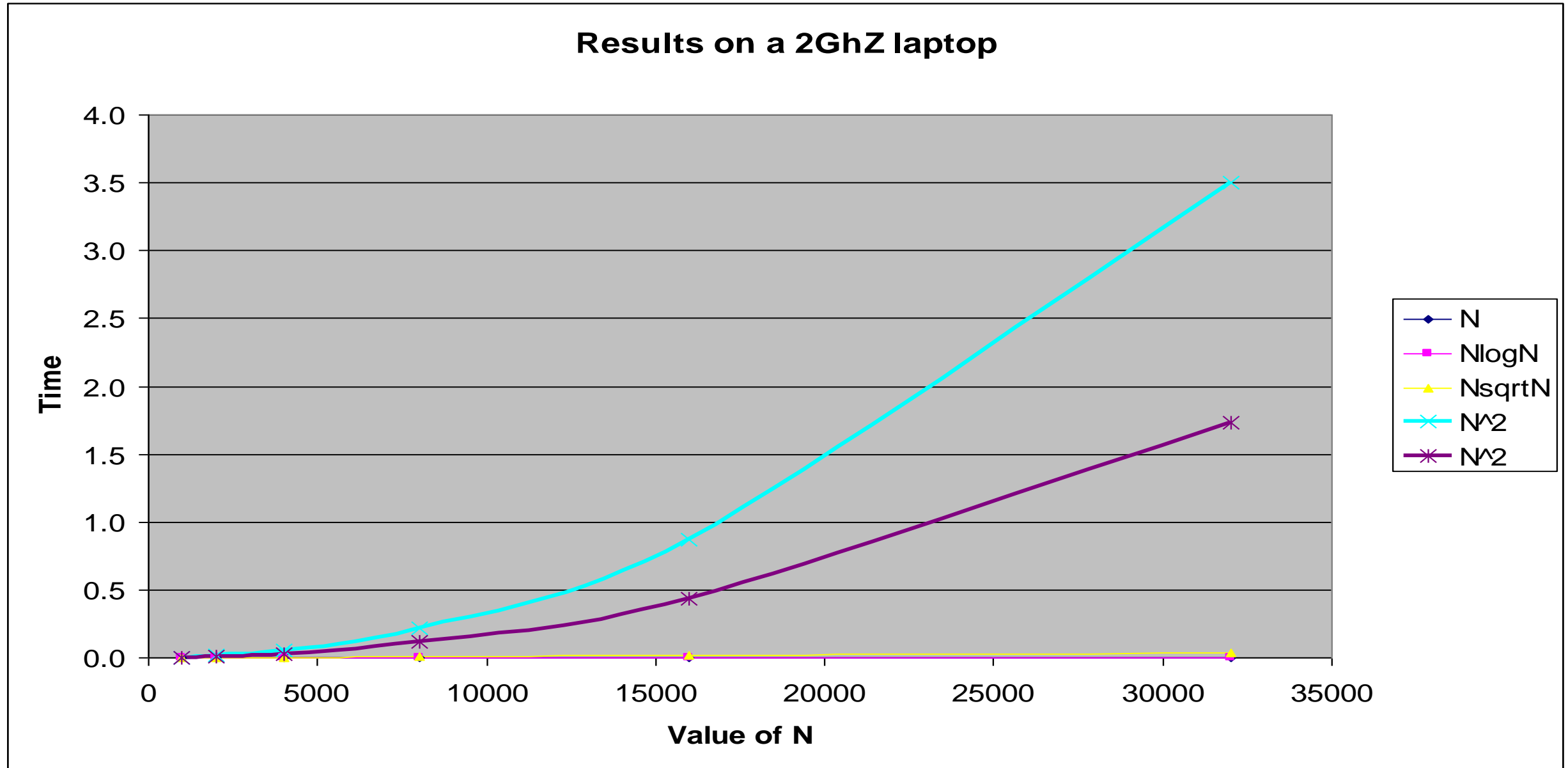
Tiempos en segundos. Rojo indica el valor previsto

Cambios entre diferentes datos de entrada

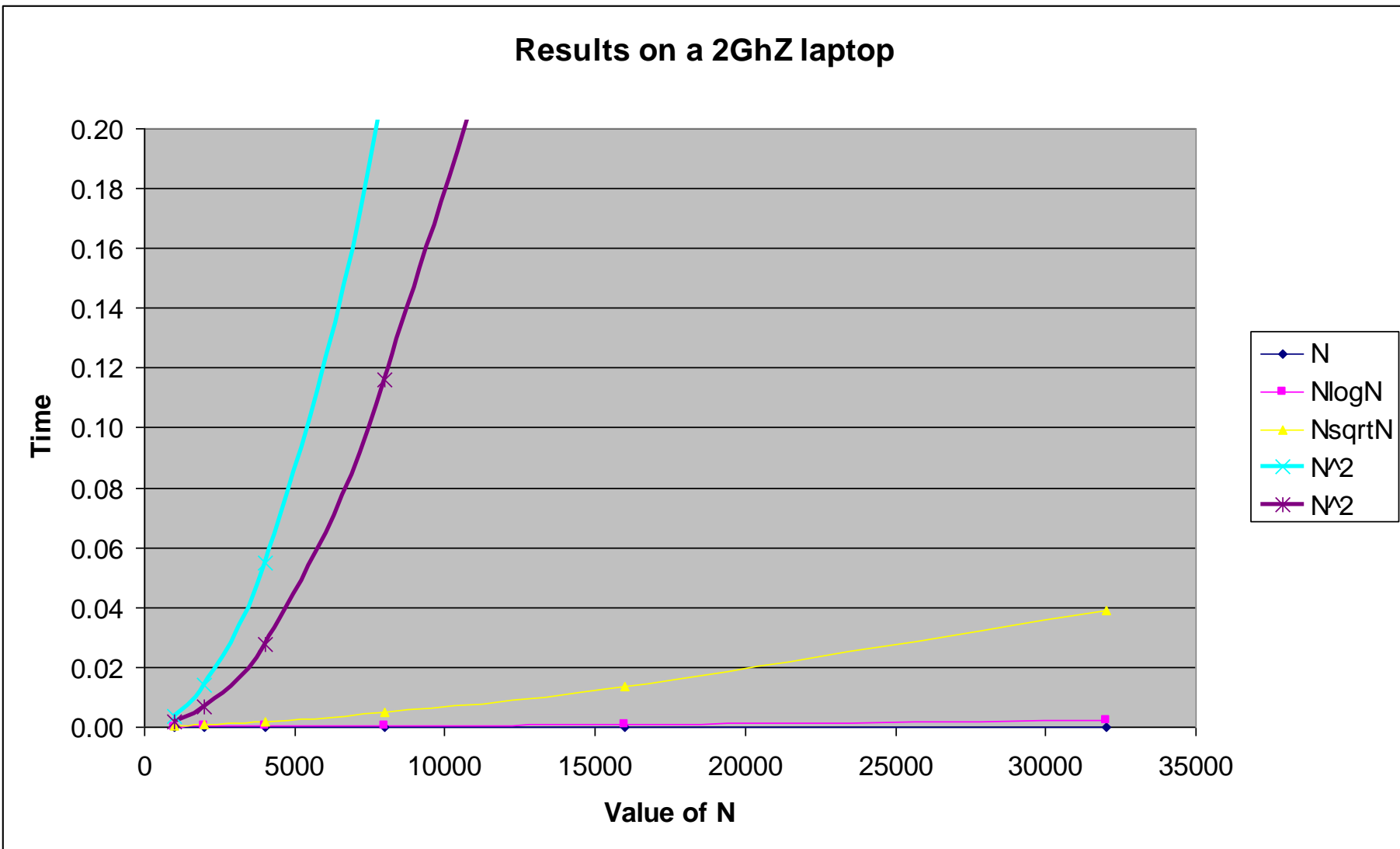
	1000	2000	4000	8000	16000	32000	64000	128K	256k	512k
$O(N)$	-	1.21	2.02	0.78	1.62	1.76	1.89	2.24	2.11	1.62
$O(N \log N)$	-	2.18	1.99	1.27	2.13	2.15	2.15	2.71	1.64	2.40
$O(N^{3/2})$	-	1.98	2.48	2.87	2.79	2.76	2.85	2.79	2.82	2.81
$O(N^2)$ ind	-	4.06	3.98	3.94	3.99	4.00	3.99	-	-	-
$O(N^2)$ dep	-	4.00	3.82	3.97	4.00	4.01	3.98	-	-	-
$O(N^3)$	-	8.03	-	-	-	-	-	-	-	-

Valor obtenido por $\text{Tiempo}_x / \text{Tiempo}_{x-1}$

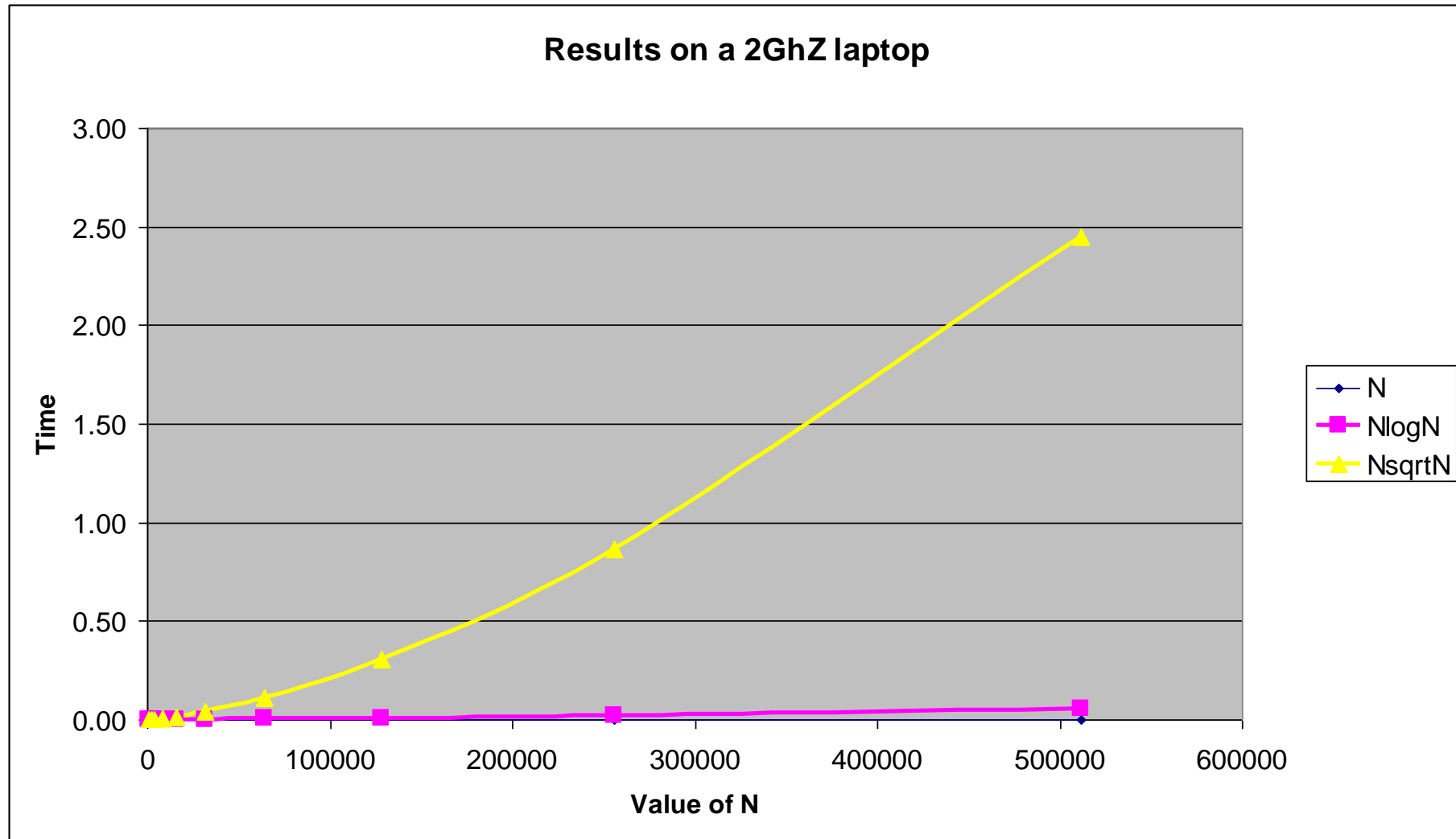
Diagramas



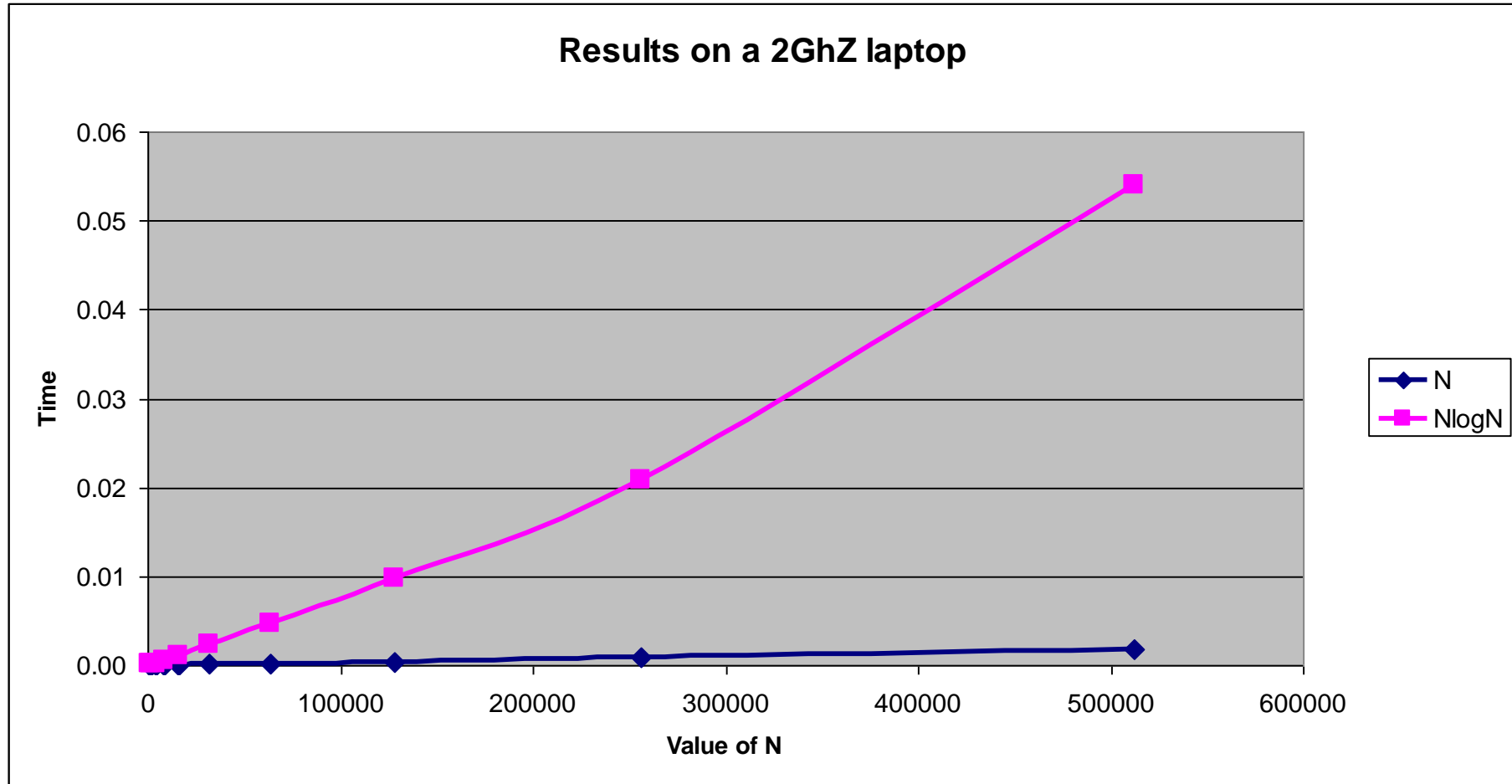
Ponga un límite al tiempo



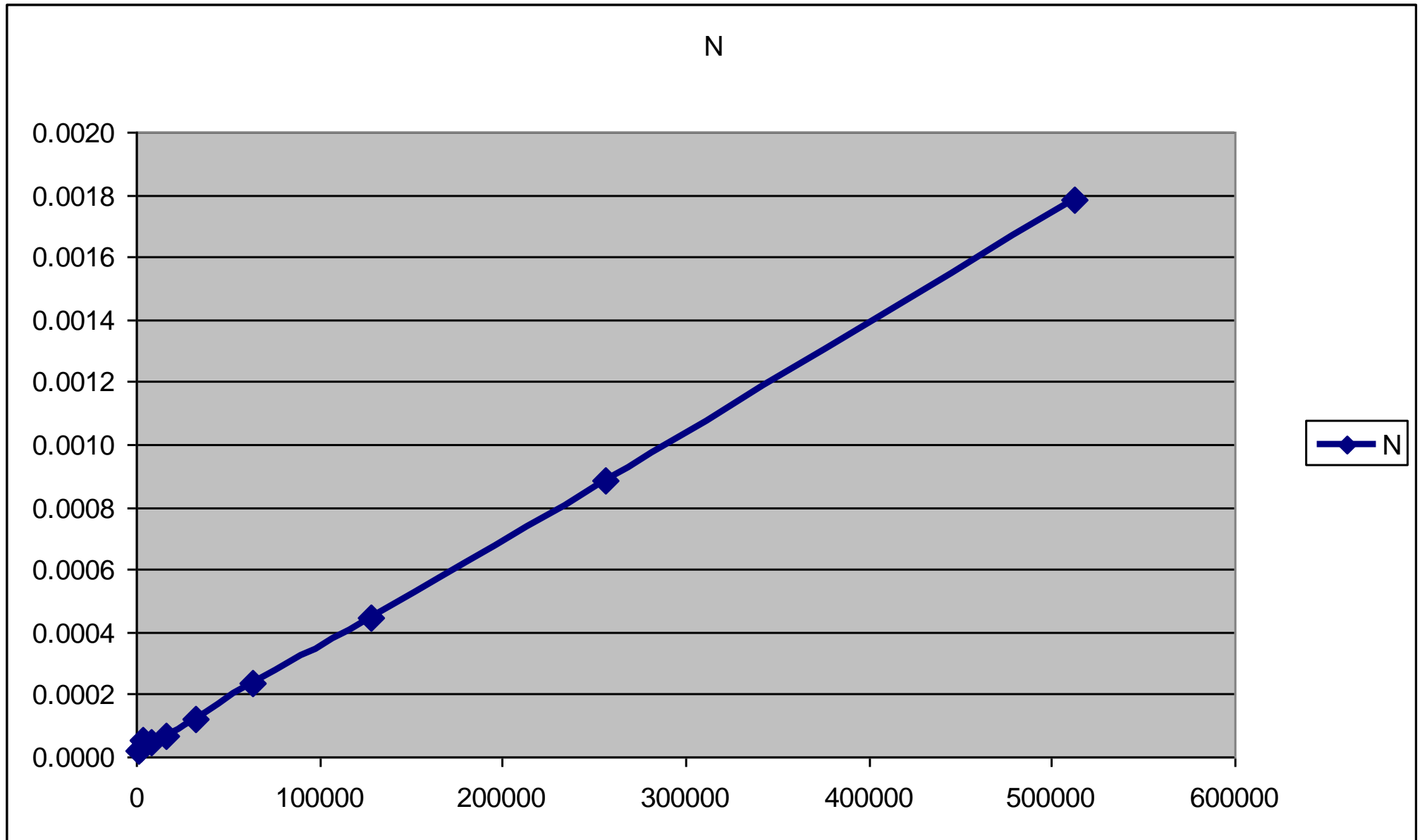
Sin datos $O(N^2)$



Sólo $O(N)$ y $O(N\log N)$



Sólo $O(N)$



10⁹ instrucciones/seg, tiempos de ejecución

N	$O(\log N)$	$O(N)$	$O(N \log N)$	$O(N^2)$
10	0.000000003	0.000000001	0.000000033	0.00000001
100	0.000000007	0.000000010	0.000000664	0.0001000
1,000	0.000000010	0.000000100	0.000010000	0.001
10,000	0.000000013	0.000001000	0.000132900	0.1 min
100,000	0.000000017	0.00010000	0.001661000	10 segundos
1,000,000	0.000000020	0.001	0.0199	16.7 minutos
1,000,000,000	0.000000030	1.0 segundo	30 segundos	31.7 años

Definición formal de Big O (repetido)

- $T(N)$ es $O(F(N))$ si hay constantes positivas c y N_0 tales que $T(N) \leq cF(N)$ cuando $N \geq N_0$
 - N es el tamaño del conjunto de datos en el que trabaja el algoritmo
 - $T(N)$ es una función que caracteriza el tiempo de ejecución *real* del algoritmo
 - $F(N)$ es una función que caracteriza un límite superior en $T(N)$. Es un límite en el tiempo de ejecución del algoritmo. (La típica table de funciones Big)
 - c y N_0 son constantes

¿Qué significa esto?

- $T(N)$ es la tasa de crecimiento real del algoritmo
 - se puede equiparar al número de declaraciones ejecutables en un programa o fragmento de código
- $F(N)$ es la función que limita la tasa de crecimiento
 - puede ser límite superior o inferior
- $T(N)$ puede no ser necesariamente igual a $F(N)$
 - constantes y términos menores ignorados porque es una *función delimitadora*

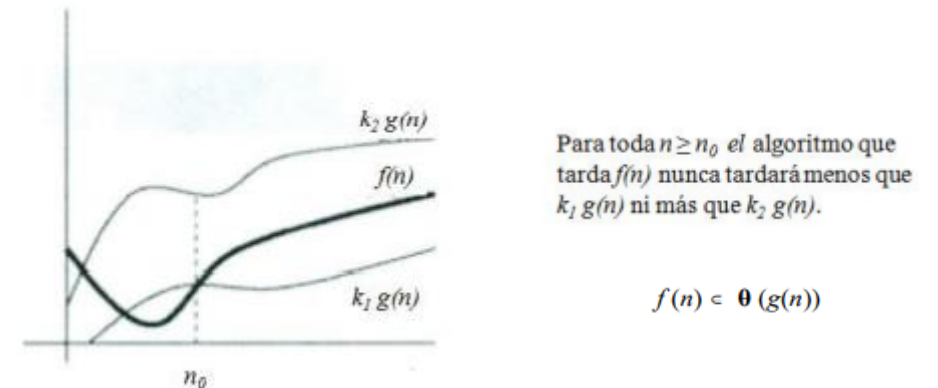
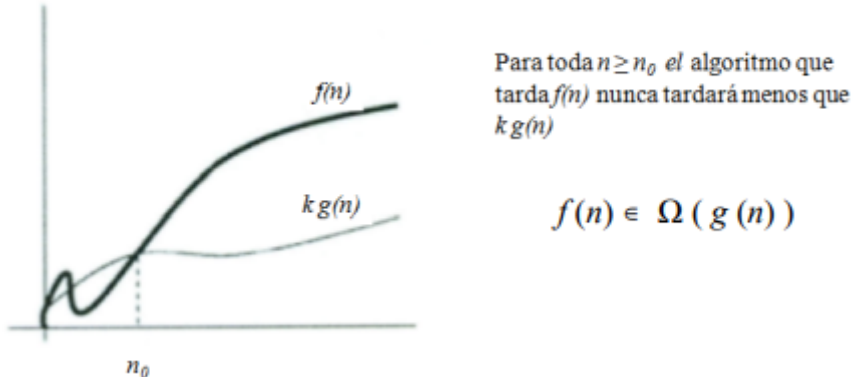
Otras herramientas de análisis algorítmico

- *Big Omega* $T(N)$ es $\Omega(F(N))$ si hay constantes positivas c y N_0 tales que
 $T(N) \geq cF(N)$ cuando $N \geq N_0$
 - Big O es similar a menor o igual que, un límite superior
 - Big Omega es similar a mayor o igual que, un límite inferior
- *Big Theta* $T(N)$ es $\theta(F(N))$ si y solo si $T(N)$ es $O(F(N))$ y $T(N)$ es $\Omega(F(N))$.
 - Big Theta es similar a iguales

Big-Omega y Big-Theta

- Big-Omega: Sean $T(n)$ y $F(n)$ funciones de los naturales en los reales. $T(n)$ es $\Omega(F(n))$ ssi existen c real positivo y n_0 natural tales que $T(n) \geq cF(n)$ para todo $n \geq n_0$.
- Big-Theta: Sean $T(n)$ y $F(n)$ funciones de los naturales en los reales. $T(n)$ es $\Theta(F(n))$ ssi $T(n)$ es $O(F(n))$ y $T(n)$ es $\Omega(F(n))$.
- Nota: Big-Theta quiere decir $c_1F(n) \leq T(n) \leq c_2F(n)$. Por lo tanto, tienen un crecimiento asintótico equivalente.

Tipo de análisis	Expresión matemática	Tasas relativas de crecimiento
Big O	$T(N) = O(F(N))$	$T(N) \leq F(N)$
Big Ω	$T(N) = \Omega(F(N))$	$T(N) \geq F(N)$
Big θ	$T(N) = \theta(F(N))$	$T(N) = F(N)$



Tasas relativas de crecimiento

Tipo de análisis	Expresión matemática	Tasas relativas de crecimiento
Big O	$T(N) = O(F(N))$	$T(N) \leq F(N)$
Big Ω	$T(N) = \Omega(F(N))$	$T(N) \geq F(N)$
Big θ	$T(N) = \theta(F(N))$	$T(N) = F(N)$

"A pesar de la precisión adicional que ofrece Big Theta, Big O se usa con más frecuencia, excepto por investigadores en el campo del análisis de algoritmos" - Mark Weiss

Comparación de varios Algoritmos de ordenamiento

Num Items	Seleccion	Insercion	Shellsort	Quicksort
1000	16	5	0	0
2000	59	49	0	6
4000	271	175	6	5
8000	1056	686	11	0
16000	4203	2754	32	11
32000	16852	11039	37	45
64000	Espera demasiado?	Espera demasiado?	100	68
128000	Espera demasiado?	Espera demasiado?	257	158
256000	Espera demasiado?	Espera demasiado?	543	335
512000	Espera demasiado?	Espera demasiado?	1210	722
1024000	Espera demasiado?	Espera demasiado?	2522	1550

Tiempo en milisegundos

¿Preguntas?



FIN