



an **NTT DATA** Company

Eficiencia de Algoritmos

Para el desarrollo backend

¿Por qué analizar los algoritmos?

Predecir el rendimiento.

Comparar algoritmos.

Proveer garantía.

En resumen: Evitar bugs de rendimiento.

Ejemplo: Ordenamiento



client gets poor performance because programmer
did not understand performance characteristics



Análisis de Eficiencia

¿De qué depende la eficiencia?

Hardware: CPU, RAM, cache, ...

Software: compilador, intérprete, ...

Sistema: S.O., redes, otras apps, ...

Algoritmo

Datos de entrada

Análisis de Eficiencia

¿De qué depende la eficiencia?

Hardware: CPU, RAM, cache, ...

Software: compilador, intérprete, ...

Sistema: S.O., redes, otras apps, ...

Algoritmo

Datos de entrada

*dependiente
del sistema*

Independiente

del sistema

Función de Tiempo (empírico)

Usando las librerías adecuadas podemos medir cuanto tiempo tardan nuestros programas en ejecutarse.

Y realizando el experimento repetidamente podemos incluso encontrar una ***función de tiempo*** que represente la relación entre los ***datos de entrada*** y el ***tiempo de ejecución***.

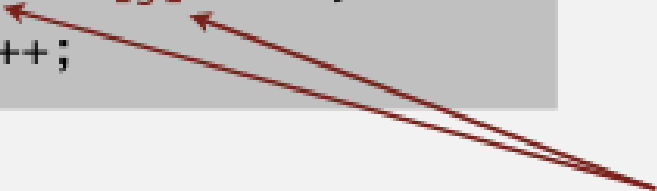
Ejemplo: Suma 2

Función de Tiempo (teórico)

Despreciando todos los factores dependientes del sistema, podemos definir una función de tiempo dependiente sólo del tamaño de la entrada de datos.

Veamos el siguiente código y calculemos la cantidad de operaciones

```
int count = 0;
for (int i = 0; i < N; i++)
    for (int j = i+1; j < N; j++)
        if (a[i] + a[j] == 0)
            count++;
```


$$0 + 1 + 2 + \dots + (N - 1) = \frac{1}{2} N (N - 1)$$

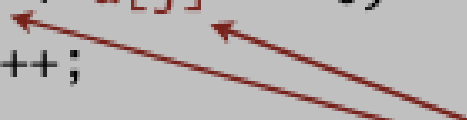
Función de Tiempo (teórico)

Cuando i es 0 , j va de 1 a $N - 1$, cuando i es 1 , j va de 2 a $N - 1$.

Es decir cuando i es 0 , se hacen $N - 1$ operaciones, cuando i es 1 se hacen $N - 2$ operaciones, así hasta cuando i es $N - 2$ y sólo hace 1 operación. Esto es:

$$(N - 1) + (N - 2) + (N - 3) + \dots + 3 + 2 + 1 = (N - 1) * N / 2$$

```
int count = 0;
for (int i = 0; i < N; i++)
    for (int j = i+1; j < N; j++)
        if (a[i] + a[j] == 0)
            count++;
```



Notación Asintótica: notación O

Debemos encontrar una función $F(N)$ que siempre sea mayor a nuestra función de tiempo $T(N)$.

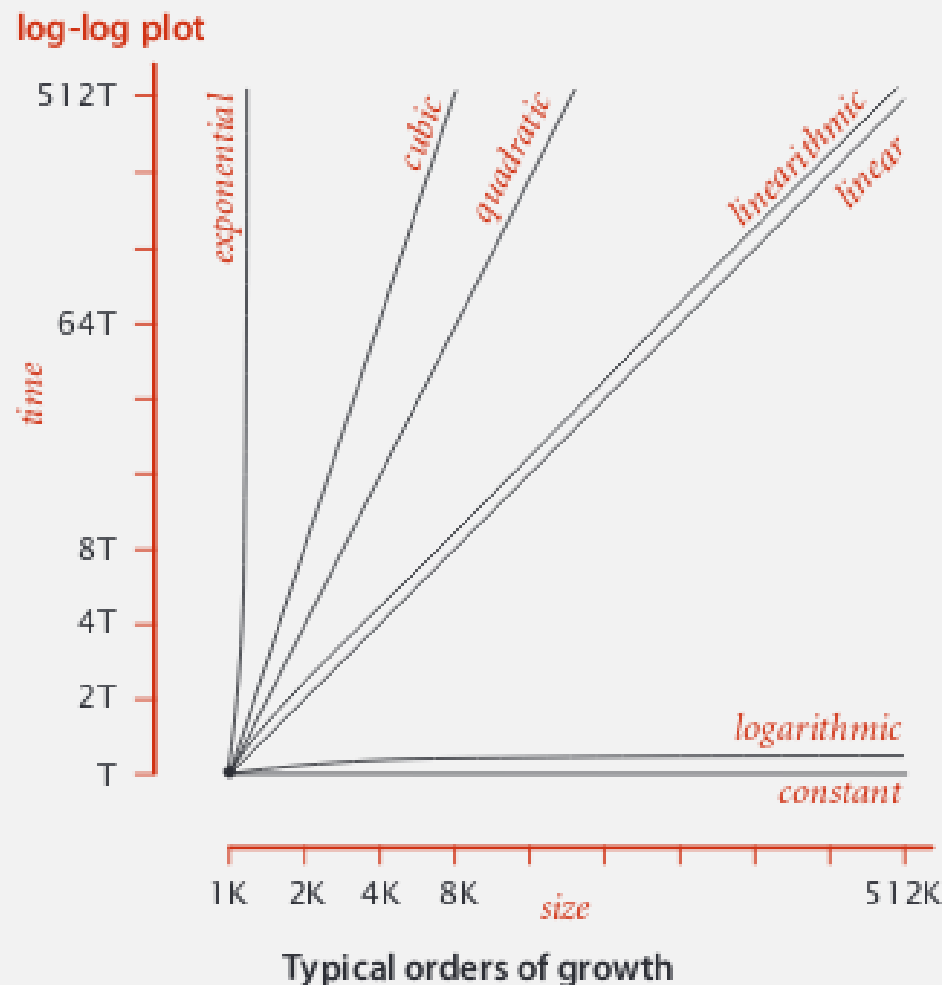
Teníamos:

$$T(N) = N * (N - 1) / 2 = N^2 / 2 - N / 2$$

Observamos que conforme crece N , cada vez es más despreciable el valor de $N / 2$ respecto a $N^2 / 2$. Por tanto hacemos $T(N) = N^2 / 2$. Entonces podemos definir una función $F(N) = k * N^2$, con $k > 1 / 2$ que siempre será mayor a $T(N)$. Se dice que la función $T(N)$ es de complejidad $O(F(N))$ esto es $O(N^2)$.

Orden de crecimiento

1, $\log N$, N , $N \log N$, N^2 , N^3 , and 2^N
suffices to describe the order of growth of most common algorithms.



Orden de crecimiento

order of growth	name	typical code framework	description	example	$T(2N) / T(N)$
1	constant	<code>a = b + c;</code>	statement	add two numbers	1
$\log N$	logarithmic	<code>while (N > 1) { N = N / 2; ... }</code>	divide in half	binary search	~ 1
N	linear	<code>for (int i = 0; i < N; i++) { ... }</code>	loop	find the maximum	2
$N \log N$	linearithmic	[see mergesort lecture]	divide and conquer	mergesort	~ 2
N^2	quadratic	<code>for (int i = 0; i < N; i++) for (int j = 0; j < N; j++) { ... }</code>	double loop	check all pairs	4
N^3	cubic	<code>for (int i = 0; i < N; i++) for (int j = 0; j < N; j++) for (int k = 0; k < N; k++) { ... }</code>	triple loop	check all triples	8
2^N	exponential	[see combinatorial search lecture]	exhaustive search	check all subsets	$T(N)$

Referencias

Algorithms 4th Edition – Robert Sedgewick and Kevin Wayne

Problemas y Algoritmos – Luis E. Vargas Azcona

Course Algorithms Part I – Coursera, Princeton University