

Estructura de Datos y Algoritmos

ITBA 2024-Q2



¿Cómo anduvo el testeo?

Contemplaron qué hacer si se hacen 2 stops?

Error?

Acumula?

¿Probaron en las 3 APIs con algo así?

```
int init= 0;
```

```
MyTimer myCrono = new MyTimer(init);
```

```
myCrono.stop(init + 86400000);
```

```
// (86400000 ms) 1 día 0 hs 0 min 0,000 s
```

```
int init= 0;
```

```
MyTimer myCrono = new MyTimer (init);
```

```
myCrono.stop(init + 86400000 * 10 );
```

Análisis de Algoritmos

Las principales métricas para medir la complejidad de algoritmos que ejecutan en máquinas secuenciales (un core) son:

1. El tiempo de ejecución (*runtime analysis/time complexity*)
2. El espacio que utilizan (*space complexity*)

1. Tiempo de ejecución

Pregunta:

¿ Y cómo mido ese tiempo?

1.A) Empíricamente



1.B) Teóricamente

1.A) Tiempo de ejecución **empírica**

A continuación tenemos 2 algoritmos: algoA y algoB.
Ambos calculan el máximo elemento de un vector.

Aclaración

Asumimos que ya tiene getters en la clase `MyTimer()`.

Para los próximos ejercicios asumimos que el método que devuelve la cantidad de milisegundos totales transcurridos en `MyTimer()` se llama así:

```
long getElapsedTime()
```

(o sea, lo que dentro del `toString` estaba entre paréntesis)

Bajar de campus

```
public static void main(String[] args) {
```

```
    MyTimer t2;
```

```
    try {
```

```
        t2= new MyTimer();
```

```
        t2.stop();
```

```
    }
```

```
    catch(Exception e) {
```

```
    }
```

```
    int[] myArray = new int[N];
```

```
    int rta;
```

```
    // generate array
```

```
    for (int rec = N; rec > 0; rec--)
```

```
        myArray[N - rec] = rec;
```


Bajar de campus

```
...  
    t2= new MyTimer();  
    rta = AlgoA.max(myArray);  
    t2.stop();  
    System.out.println(String.format("max Algo A %d. Delay %d (ms)", rta, t2.getElapsedTime()));  
  
    // generate array  
    for (int rec = N; rec > 0; rec--)  
        myArray[N - rec] = rec;  
  
    t2= new MyTimer();  
    rta = AlgoB.max(myArray);  
    t2.stop();  
    System.out.println(String.format("max Algo B %d. Delay %d (ms)", rta, t2.getElapsedTime()));  
  
    }  
  
}
```

Bajar de campus

```
public class AlgoA {  
  
    public static int max(int[] array) {  
        if (array == null || array.length == 0)  
            throw new RuntimeException("Empty array");  
  
        int candidate = array[0];  
        for (int rec = 1; rec < array.length; rec++)  
            if ( candidate < array[rec] )  
                candidate = array[rec];  
  
        return candidate;  
    }  
}
```

```
import java.util.Arrays;  
  
public class AlgoB {  
  
    public static int max (int[] array) {  
        if (array == null || array.length == 0)  
            throw new RuntimeException("Empty array");  
  
        Arrays.sort(array); // ordena ascendentemente  
  
        return array[array.length - 1];  
    }  
}
```

TP 1- Ejer 9

Generar un Nuevo Proyecto Maven con los seteos convenientes.

Debe usar la **biblioteca TimerFromScratch-1** (la API **from scratch**) para determinar cuál de los 2 algoritmos es mejor: ¿**algoA** o **algoB**?

Armar una tabla de comparación con los ms de cada uno para las siguientes ejecuciones.

Completar

El tiempo de ejecución empírico para los sig. valores:

n	Time(algoA) en ms	Time(algoB) en ms
1000		
100 000 000		
200 000 000		
400 000 000		
600 000 000		
800 000 000		
2 000 000 000		

Para poder compilar:

Tip: deberán incluir la dependencia de su biblioteca
Timer a usar



```
<dependencies>  
  <dependency>  
    <groupId>ar.edu.itba.eda</groupId>  
    <artifactId>TimerFromScratch</artifactId>  
    <version>1</version>  
  </dependency>  
</dependencies>
```

Estos son los que obtuve yo:

n	Time(algoA) en ms	Time(algoB) en ms
1000	2	1
100 000 000	100	364
200 000 000	160	850
400 000 000	320	1600
600 000 000	500	2349
800 000 000	1021	4201
2 000 000 000	Heap Overflow	



Cuidado: muchas veces inferimos que un algoritmo es bueno porque lo ejecutamos con “pocos datos” y cuando lo ponemos en producción no se comporta igual. El tamaño del input puede afectar la performance de un algoritmo.

1.A) Tiempo de ejecución **empírica**

La idea de usar la métrica “tiempo de ejecución calculada empíricamente” para rankear algoritmos tiene varias dificultades. ¿Cuáles?

Por ejemplo:

- Como los algoritmos tardan diferente dependiendo de los datos con los que operan (input), para probar realmente con datos grandes, habría que generar esos valores. Podría tardar días chequear los tiempos en grandes inputs....

¡Qué bueno sería si pudiera caracterizar algoritmos sin tener que generar esos datasets masivos !

- Si mi `algoA` lo ejecutó en mi compu y tarda X ms, y otro propone un `algoB` que ejecuta en su compu y tarda $X/2$ ms, ¿Cómo saber cuál realmente tarda menos si las computas son diferentes!!!. Ese ranking puede ser engañoso.

¡Qué bueno sería si pudiera caracterizar algoritmos sin depender de hardware y software donde ejecutan !

1.B) Tiempo de ejecución **teórica**

Consiste en usar una descripción de alto nivel del algoritmo para evaluar su eficiencia independientemente del hardware y software donde ejecute. Se lo describe con una “expresión (fórmula).”

Idea básica: “**contar la cantidad de operaciones primitivas**” que ejecuta el algoritmo. No importa cuanto tardan. Tardan “unidades de tiempo genéricas”.

Dichas operaciones son las más costosas en ejecutar en cualquier computadora: comparaciones, operaciones (aritméticas), transferencia de control desde una fn hacia otra. (las asignaciones llevan tiempo despreciable, se pueden ignorar).

Como el tamaño del input afecta la performance del algoritmo, entonces la “fórmula” se realiza contando la cantidad de operaciones primitivas que se realiza expresada en términos del tamaño de entrada.

TP 1- Ejer 10

Calcular $T(\text{algoA})$ como una fórmula expresada en términos de la cantidad de operaciones que realiza para un arreglo de tamaño N (sabemos que depende de eso...)

Luego, calcular O grande, es decir, cota.

1.B) Tiempo de ejecución **teórica**

```
public class AlgoA {  
  
    public static int max (int[] array) {  
        if (array == null || array.length == 0)  
            throw new RuntimeException("Empty array");  
  
        int candidate= array[0];  
        for (int rec= 1; rec < array.length;  rec++)  
            if ( candidate < array[rec] )  
                candidate= array[rec];  
  
        return candidate;  
    }  
}
```

3 operaciones fijas

Luego: 1 comparación+1 suma +
1 comparación. Esto se hace N-1
veces

$$T(\text{algoA}) = 3 + 3 * (N - 1) = 3 * N$$

1.B) Tiempo de ejecución **teórica**

```
public class AlgoB {  
  
    public static int max (int[] array) {  
        if (array == null || array.length == 0)  
            throw new RuntimeException("Empty array");  
  
        // ordena ascendentemente  
        Arrays.sort(array);  
  
        return array[array.length-1];  
    }  
}
```

3 operaciones fijas

Esa invocación, es 1 operación,
pero qué conlleva esa ejecución?
Buscar cómo dice que lo
implementó Java:

<https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html>

1 operaciones fijas


$$T(\text{algoB}) = 4 + N * \ln(N)$$

1.B) Tiempo de ejecución **teórica**

Resumiendo:


$$T(\text{algoA}) = 3 * N$$

$$T(\text{algoB}) = 4 + N * \ln(N)$$



1.B) Tiempo de ejecución **teórica**

Consiste en usar una descripción de alto nivel del algoritmo para evaluar su eficiencia independientemente del hardware y software donde ejecute. Se lo describe con una “expresión (fórmula).”



Comportamiento asintótico cota superior u O grande

La descripción que buscamos para comparar algoritmos es una “asíntota” (cota) expresada en términos de N que nos permita caracterizar la “tasa de crecimiento u orden de crecimiento de la fórmula”

1.B) Tiempo de ejecución teórica

Definición de Comportamiento asintótico superior u O grande (*asymptotic upper bound running time u O-notation*) de un algoritmo.

Sean $T(N)$ y $g(N)$ funciones con $N > 0$.

Se dice que $T(n)$ es $O(g(N))$ si $\exists c > 0$ (constante no dependiente de N) y $\exists n_0 > 0$ tal que $\forall N \geq n_0$ se cumple que $0 \leq T(N) \leq c * g(N)$.

Ej: En nuestro caso, algoA tiene $T(N) = 3 * N$

Pero para hacer más complicada la discusión, supongamos que $T(N) = 1 + 3 * N$, o sea $g(N)$ puede ser N .

O sea, si $0 \leq 1 + 3 * N \leq c * N$ ¿cuánto debe valer c ?

$$0 \leq 1/N + 3 \leq c$$

Si N es 2 entonces $c \geq 3.5$

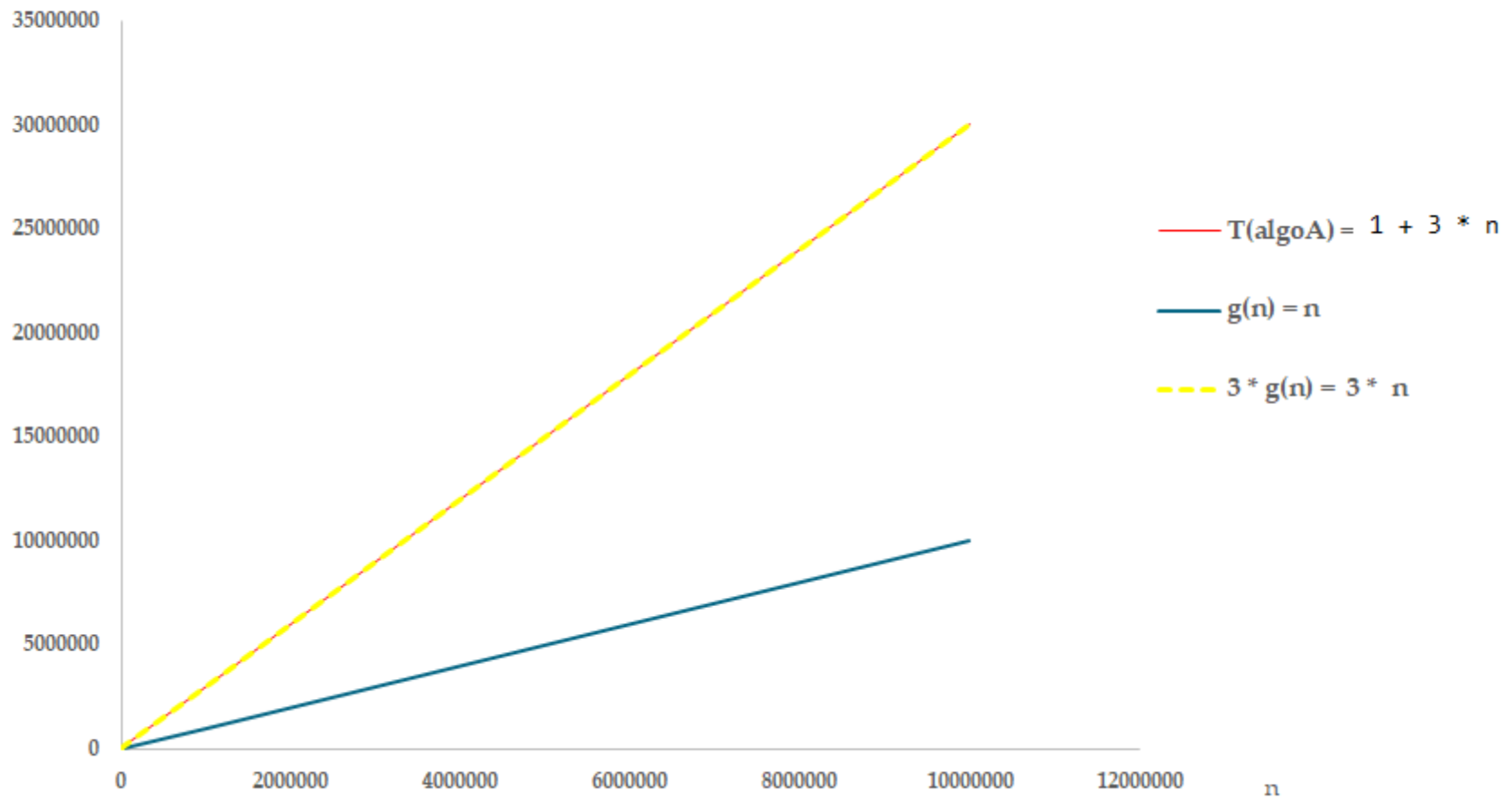
Si N es 5 entonces $c \geq 3.2$, mejor la cota.

Si N es 500, $c \geq 3.002$, mejor aún.

Yo quiero $N \rightarrow \infty$, entonces $c \geq 3$. Así, el orden de algoA es $O(N)$.

Encontré caracterizar una constante c , que es 3, para el cual se verifica la fórmula para $N \rightarrow \infty$ ($\forall N \geq n_0$). El algoritmo es $O(N)$.

Gráfico de tasa de crecimiento



1.B) Tiempo de ejecución **teórica**

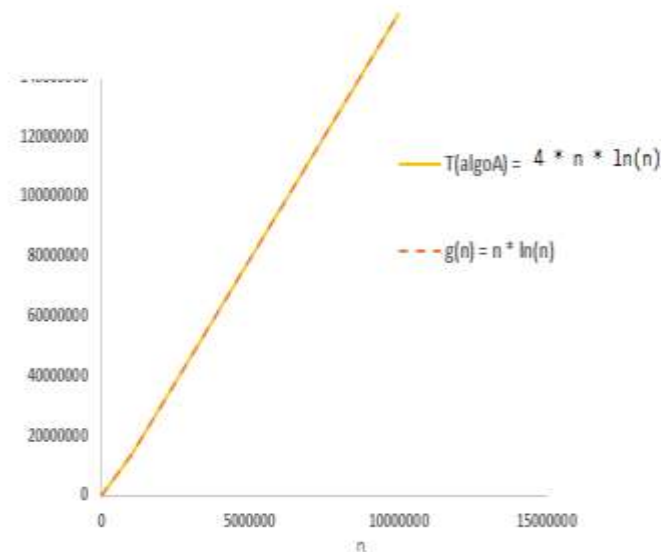
Ahora uds...

$$T(\text{algoB}) = 4 + N * \ln(N)$$

¿Cuál es O para algoB?

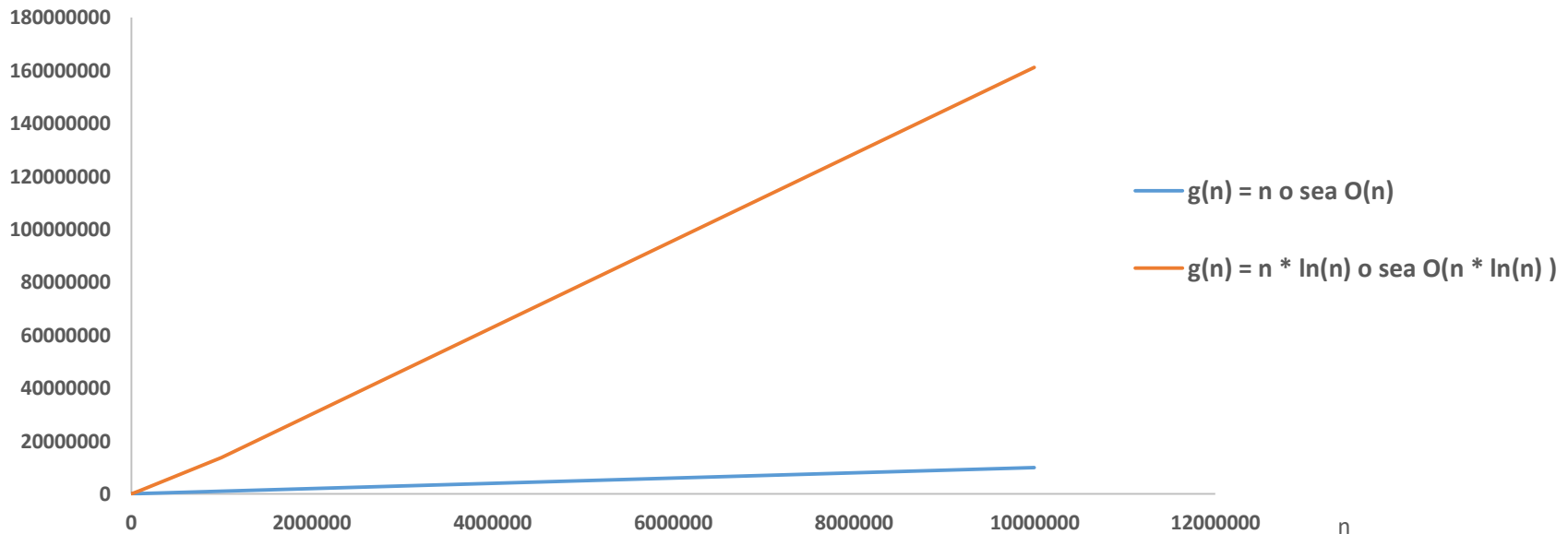
Rta: algoB tiene $O(N * \ln(N))$

Gráfico de tasa de crecimiento



y finalmente, ¿Cuál de los 2 algoritmos es mejor, para N creciente?

Gráfico de tasa de crecimiento



Rta: $O(N)$. O sea, algoA es mejor que algoB. Coincide con el cálculo empírico. Bien!

TP 1- Ejer 12

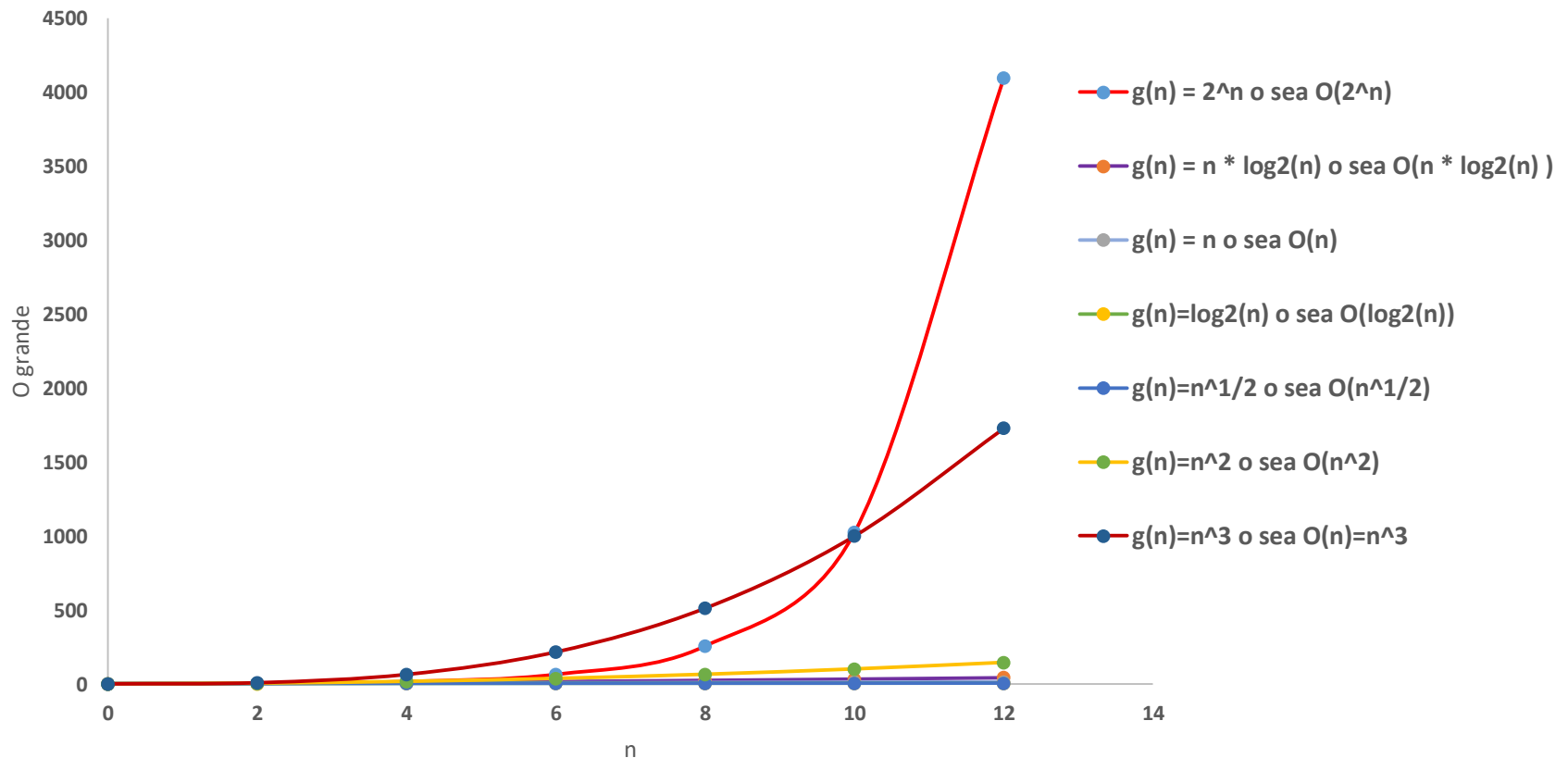


Caracterizar las siguientes complejidades O grande, para ver cuáles algoritmos serían mejores que otros.

Representar en una planilla de cálculo el gráfico del comportamiento para $n= 0, 2, 4, 6, 8, 10$ y 12 para las siguientes complejidades: $O(2^N)$, $O(N \cdot \log_2(N))$, $O(N)$, $O(\log_2(N))$, $O(\sqrt{N})$, $O(N^3)$, $O(N^2)$

Rta: el peor es $O(2^N)$ a partir de $n \geq 10$ (no muy grande...)

Gráfico de tasa de crecimiento



Rta: por ejemplo $O(N!)$

Gráfico de tasa de crecimiento

El gráfico muestra la tasa de crecimiento de dos funciones, $g(n) = 2^n$ (línea azul) y $g(n) = n!$ (línea naranja), en función de n . El eje horizontal (n) va de 0 a 10, y el eje vertical (O grande) va de 0 a 45,000. La función $g(n) = 2^n$ crece exponencialmente, mientras que $g(n) = n!$ crece factorialmente, superando a la primera a partir de $n=6$.

n	$g(n) = 2^n$	$g(n) = n!$
0	1	1
1	2	1
2	4	2
3	8	6
4	16	24
5	32	120
6	64	720
7	128	5,040
8	256	40,320

Dado que 2^N y N^3 crecen tanto, para poder ver gráficamente cómo crecen las otras, saquémoslo del gráfico. ¿Cómo queda?



n	$g(n) = \log_2(n)$ o sea $O(\log_2(n))$	$g(n) = n^{1/2}$ o sea $O(n^{1/2})$
0	#NUM!	0
10	3,321928095	3,16227766
20	4,321928095	4,472135955
30	4,906890596	5,477225575
40	5,321928095	6,32455532
50	5,64385619	7,071067812
60	5,906890596	7,745966692
70	6,129283017	8,366600265
80	6,321928095	8,94427191
90	6,491853096	9,486832981
100	6,64385619	10
128	7	11,3137085
1024	10	32

Es decir que para $n \rightarrow \infty$ tenemos que

$$\dots < O(\log_2(N)) < O(\sqrt{N}) < O(N) < O(N * \log_2(N)) < O(N^2) < O(N^3) < O(2^N) < O(N!) < \dots$$

¿ Cómo se considera que es la O grande cuando el algoritmo no depende del tamaño de entrada de los datos?

$O(1)$. Obviamente, la menor de todas!

1. Tiempo de ejecución

Pregunta:

¿ Y cómo mido ese tiempo?

1.A) Empíricamente



1.B) Teóricamente

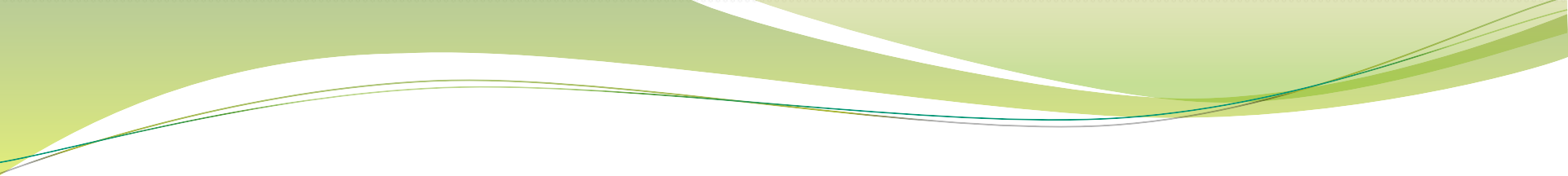


Importante

Para realizar el cálculo de la complejidad temporal de un algoritmo, no basta con analizar el “tamaño de los datos de entrada”. La performance del algoritmo también puede depender de cómo vienen los datos.

Ej: si ordeno un vector de componentes, puede que mi algoritmo sea mejor si los datos vienen “casi ordenados” que si vienen “totalmente desordenados”.

Se puede hacer un análisis del “mejor caso”, “caso promedio” o “peor caso” de input. Nosotros, salvo que digamos lo contrario, vamos a realizar siempre el análisis del peor caso.

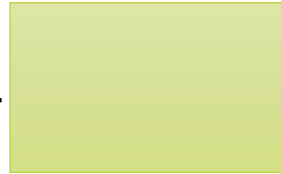


Resumiendo: el cálculo de la complejidad temporal de un algoritmo depende del tamaño de los datos de entrada y de cómo vienen esos datos.

Ejercicios. Calcular O grande en cada caso, conociendo $T(N)$

$$T(N) = 6 * N + 2$$

=>



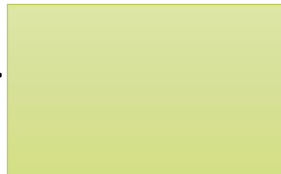
$$T(N) = 2 * N^3 + 100 * N^2$$

=>



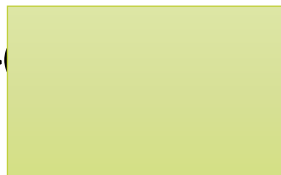
$$T(N) = 2^N + N^3$$

=>




$$T(N) = N + 6 * \log_{10} N$$

=>



Análisis de Algoritmos

Las principales métricas para medir la complejidad de algoritmos que ejecutan en máquinas secuenciales (un core) son:

1. El tiempo de ejecución (*runtime analysis/time complexity*) 
2. El espacio que utilizan (*space complexity*)

2. Espacio de RAM

Pregunta:

¿ Y cómo mido ese espacio?

Teóricamente

Para que un algoritmo ejecute algo en el procesador, los datos deben estar en RAM.

O sea, puede ser que los datos residan en disco, pero el procesador no va directo a disco. Va a disco, lo carga en RAM y ejecuta. O sea, que ese es el espacio que tendremos en cuenta.

2) Espacio de RAM

Consiste en usar una descripción de alto nivel del algoritmo para evaluar cuánta **espacio extra precisa** para sus variables (parámetros formales, invocaciones a otras funciones, variables locales). Se lo describe con una “expresión (fórmula) en términos del tamaño de entrada del problema.

La idea es la misma, buscan una cota (O grande) para el espacio RAM (stack y heap). Busca independizarse de software y hardware, es decir no va tener en cuenta si una computadora es de 32 bits o 64 bits, etc. Se expresa a través de “ N ”.

Calcular complejidad espacial

```
public class algoA {  
  
    public static int max (int[] array)  
    {  
        if (array == null || array.length == 0)  
            throw new RuntimeException("Empty array");  
  
        int candidate= array[0];  
        for (int rec= 1; rec < array.length; rec++)  
            if ( candidate < array[rec] )  
                candidate= array[rec];  
  
        return candidate;  
    }  
}
```

```
public class algoB {  
  
    public static int max (int[] array)  
    {  
        if (array == null || array.length == 0)  
            throw new RuntimeException("Empty array");  
  
        Arrays.sort(array); // ordena ascendentemente  
  
        return array[array.length-1];  
    }  
}
```

Calcular complejidad espacial

```
public class algoA {
```

```
public static int max (int[] array)  
{
```

```
    if (array == null || array.length == 0)  
        throw new RuntimeException("Empty array");
```

```
    int candidate= array[0];
```

```
    for (int rec= 1; rec < array.length; rec++)
```

```
        if ( candidate < array[rec] )  
            candidate= array[rec];
```

```
    return candidate;
```

```
}
```

```
}
```

array es un puntero a un arreglo pre
alocado: 1 unidad

1 unidad para candidate.
1 unidad para rec

$S(\text{algoA}) = 3$ o sea, el espacio usado es de $O(1)$

Calcular complejidad espacial

```
public class algoB {
```

```
    public static int max (int[] array)
    {
```

```
        if (array == null || array.length == 0)
            throw new RuntimeException("Empty array");
```

```
        Arrays.sort(array); // ordena ascendentemente
```

```
        return array[array.length-1];
```

```
    }
```

```
}
```

array es un puntero a un arreglo pre
alocado: 1 unidad

????

Miren la implementación:

<https://github.com/frohoff/jdk8u-dev-jdk/blob/master/src/share/classes/java/util/DualPivotQuicksort.java>

S(algoB) usa una unidad (puntero), e invoca a un algoritmo que tiene dos invocaciones recursivas que se acumulan.

El espacio usado es $O(N)$? Es $O(\log N)$?

Calcular complejidad espacial

En términos de espacio, también **algoA** es mejor que **algoB**.

algoA es superador que **algoB**



Sin embargo, no siempre un algoritmo es mejor que otro tanto en la parte temporal como en la espacial.

Muchas veces sucede que es mejor en lo temporal y peor en lo espacial o viceversa.

Es decir, evaluar si un algoritmo es mejor que otro puede ser un tradeoff entre espacio vs tiempo...



Ejemplo de trafeoff

algoC tiene

complejidad temporal $O(2^n)$

complejidad espacial $O(\log_2(n))$

algoD tiene

complejidad temporal $O(n)$

complejidad espacial $O(n)$

¿Cuál eligen como mejor?

2. Espacio de RAM

Para caracterizar el espacio en nuestros algoritmos que escribimos en Java, tenemos que tener en cuenta el espacio que se aloca para:

- a) **Heap** => cada vez que hacemos “new” reservamos lugar en esta zona. El GC es el proceso que libera esa zona cuando detecta que una zona ya no es más referenciada por ninguna variable.
- b) **Stack** => cada vez que se invoca un método se genera un *stack frame* para el mismo, conteniendo: los parámetros formales con sus valores, variables auxiliares declaradas dentro del mismo y el lugar de la próxima sentencia que falta a ejecutar (así, cuando se retorne, continúa la ejecución). O sea, no resulta gratis “invocar funciones”, se generan *stack frames*...

. Ejemplo:

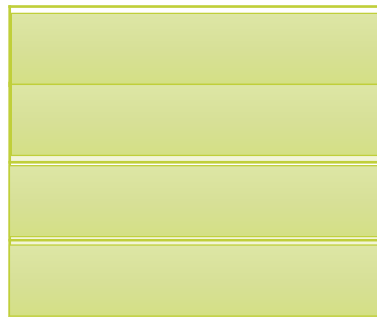
```
x = 3;
```

```
matriz= new int [x];
```

```
other = matriz;
```

```
newone= new int[x];
```

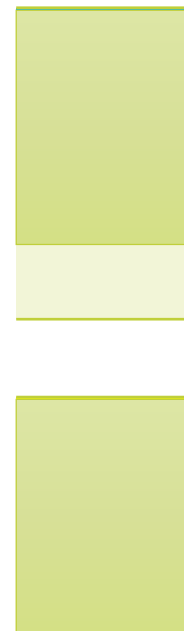
Stack



Heap

\$2A

\$10



Ejemplo:

`x = 3;`

`matriz= new int [x];`

`other = matriz;`

`newone= new int[x];`

`other = newone;`

Stack

\$2A (newone)
\$ 2A (other)
\$10 (matriz)
3 (x)

Heap

