uc3m | Universidad Carlos III de Madrid

Tema 3.
Análisis de algoritmos

Estructura de Datos y Algoritmos (EDA)

Objetivos

- Al final de la clase, los estudiantes deben ser capaces de:

1) Determinar empíricamente la complejidad temporal de algoritmos simples.
2) Determinar el orden de complejidad de un algoritmo.
3) Comparar y clasificar los algoritmos de acuerdo a su complejidad.
4) Diferenciar los conceptos mejor caso y peor caso en el rendimiento de un algoritmo.

Contenidos

- Análisis de Algoritmos
  - Análisis Empírico de Algoritmos
  - Análisis Teórico de Algoritmos

Análisis de Algoritmos

- Un problema puede tener varias soluciones diferentes (**algoritmos**)

- Objetivo: **elegir el algoritmo más eficiente**

Análisis de Algoritmos

- Un **algoritmo** es un conjunto de pasos (instrucciones) para resolver un problema
- Debe ser correcto !!!
- Debe ser eficiente !!!

Análisis de Algoritmos

- Estudiar el rendimiento de los algoritmos (tiempo de ejecución y espacio en memoria)
- Comparar algoritmos
- Enfoque basado en el tiempo: ¿cómo estimar el tiempo requerido para un algoritmo?
  - Análisis empírico
  - Análisis teórico

Contenidos

- Análisis de Algoritmos
  - **Análisis Empírico de Algoritmos**
  - Análisis Teórico de Algoritmos

Análisis Empírico de Algoritmos

1. Escribe el programa
2. Incluye instrucciones para medir el tiempo de ejecución
3. Ejecuta el programa con entradas de diferentes tamaños
4. Representa gráficamente los resultados

Análisis Empírico de Algoritmos

- Dado un número n, desarrolle un método para sumar de 1 a n

1. Escribe el programa:

```
public static long sum(long n) {
    long result=0;
    for (long i=1; i<=n; i++) {
        result = result + i;
    }
    return result;
}
```

Análisis Empírico de Algoritmos

2. Incluye instrucciones para medir el tiempo de ejecución

Use System.currentTimeMillies() or System.nanoTime().

```
long startTime = System.currentTimeMillis();

//code lines whose time you want to measure

long endTime = System.currentTimeMillis();
System.out.println("Took "+(endTime - startTime) + " ms");
```

Análisis Empírico de Algoritmos

2. Incluye instrucciones para medir el tiempo de ejecución

```
public static long sum(long n) {
    long startTime = System.nanoTime();

    long result=0;
    for (long i=1; i<=n; i++) {
        result = result + i;
    }

    long endTime = System.nanoTime();
    long total=endTime - startTime;

    System.out.println("sum("+n+")  took "+total + " ns");

    return result;
}
```

Análisis Empírico de Algoritmos

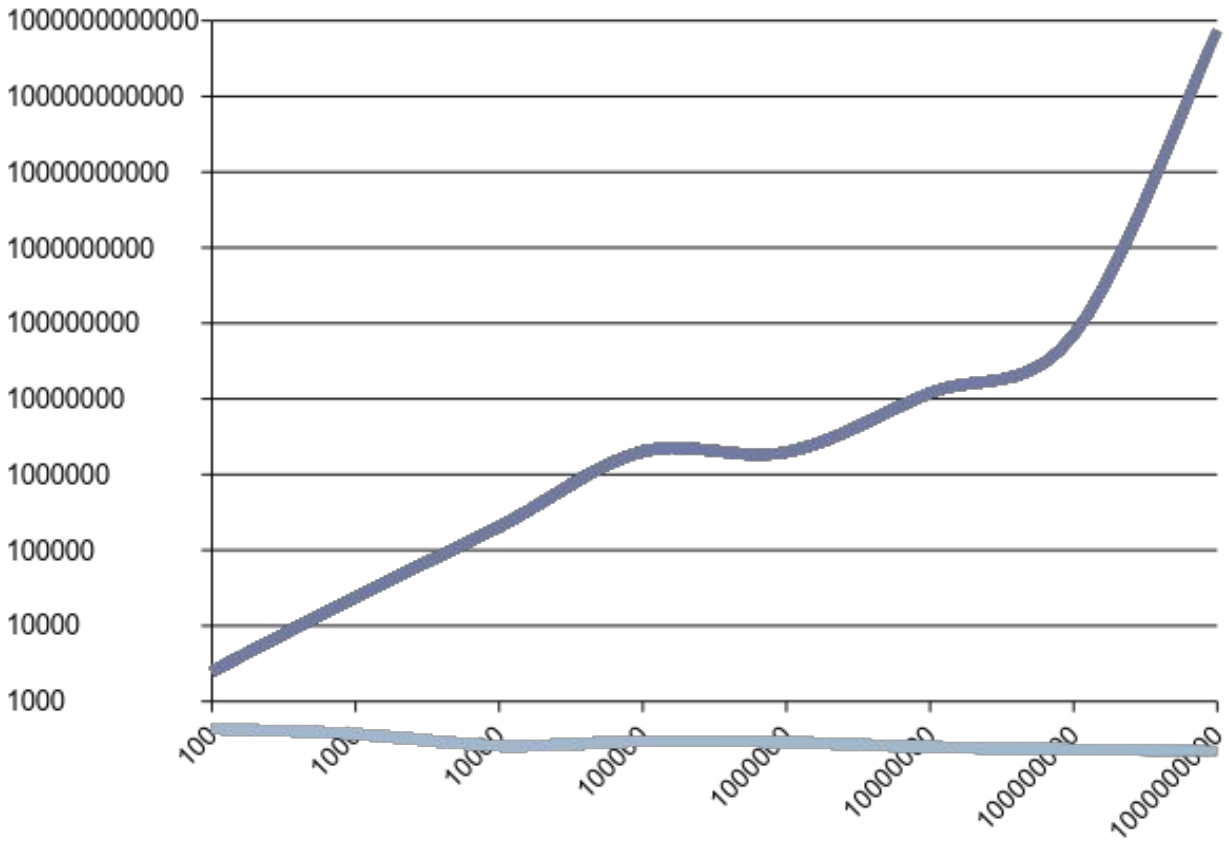
3. Ejecuta el programa con entradas de diferentes tamaños

```
long MAX=1000000000;
for (int n=1000; n<=MAX; n=n*10)
    sum(n);
```

n	tiempo (ns)
100	2485
1.000	23996
10.000	204102
100.000	2022441
1.000.000	1973428
10.000.000	12012791
100.000.000	69984715
1.000.000.000	743431482

Análisis Empírico de Algoritmos

1. Representa gráficamente los resultados



Análisis Empírico de Algoritmos

- Cuando se necesita mostrar rangos muy grandes (como en el ejemplo anterior), utilizar un gráfico Log-log
- El **gráfico Log-log** usa escalas logarítmicas en los ejes horizontal y vertical
- ¿Cómo se puede hacer un gráfico Log-log en Excel?

1. En el gráfico XY (dispersión), hacer doble clic en la escala de cada eje
2. En el cuadro Formato de ejes, seleccionar la pestaña Escala y luego verifique la escala logarítmica

Análisis Empírico de Algoritmos

- ¿Hay otros algoritmos que resuelven este problema?

```
public static long sum(long n) {
    long result=0;
    for (long i=1; i<=n; i++) {
        result = result + i;
    }
    return result;
}
```

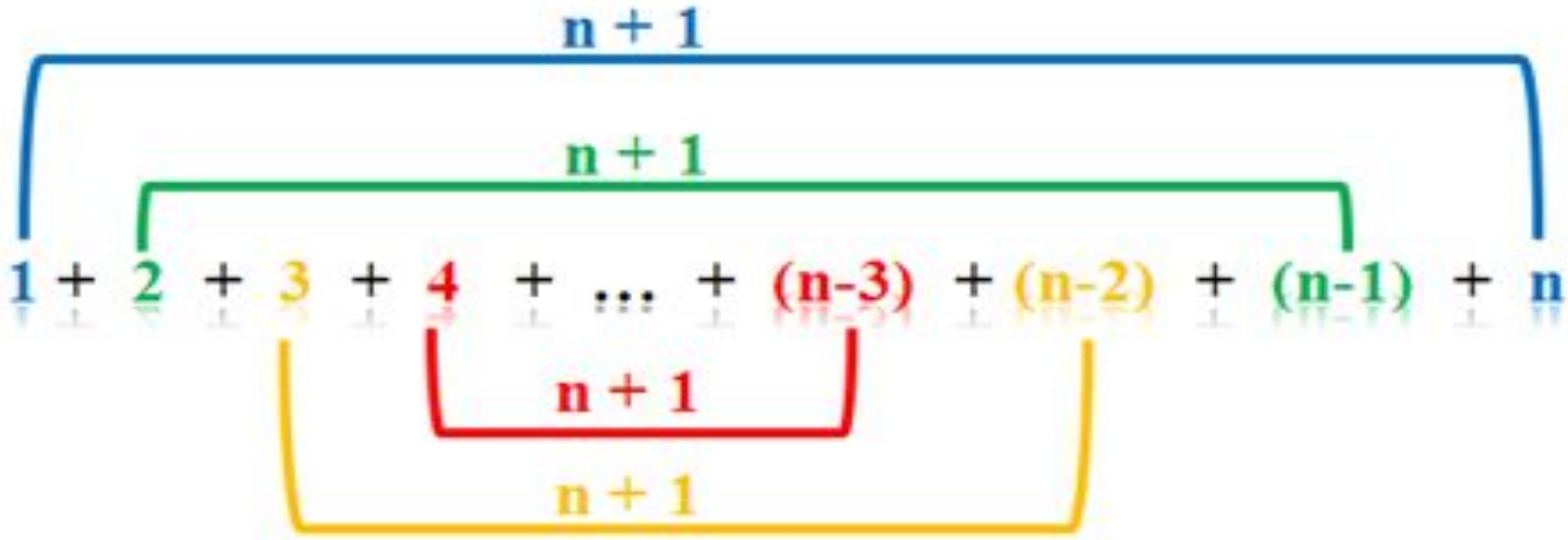
Análisis Empírico de Algoritmos

- La solución de Gauss para sumar números del 1 al n

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

Nota: Se puede encontrar una explicación fácil en:
http://mathandmultimedia.com/2010/09/15/sum-first-n-positive-integers/

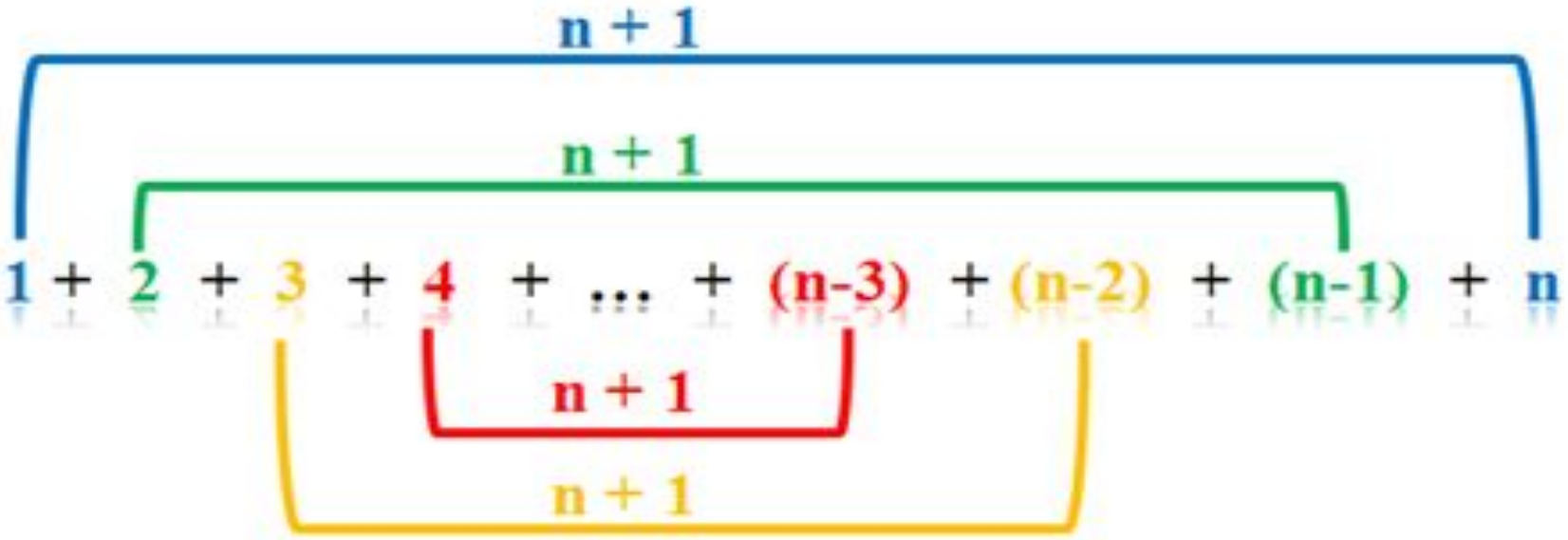
Análisis Empírico de Algoritmos



http://mathandmultimedia.com/2010/09/15/sum-first-n-positive-integers/

Si n es par, hay n / 2 pares
Si n es impar, hay (n-1) / 2 pares
Cada par suma n + 1

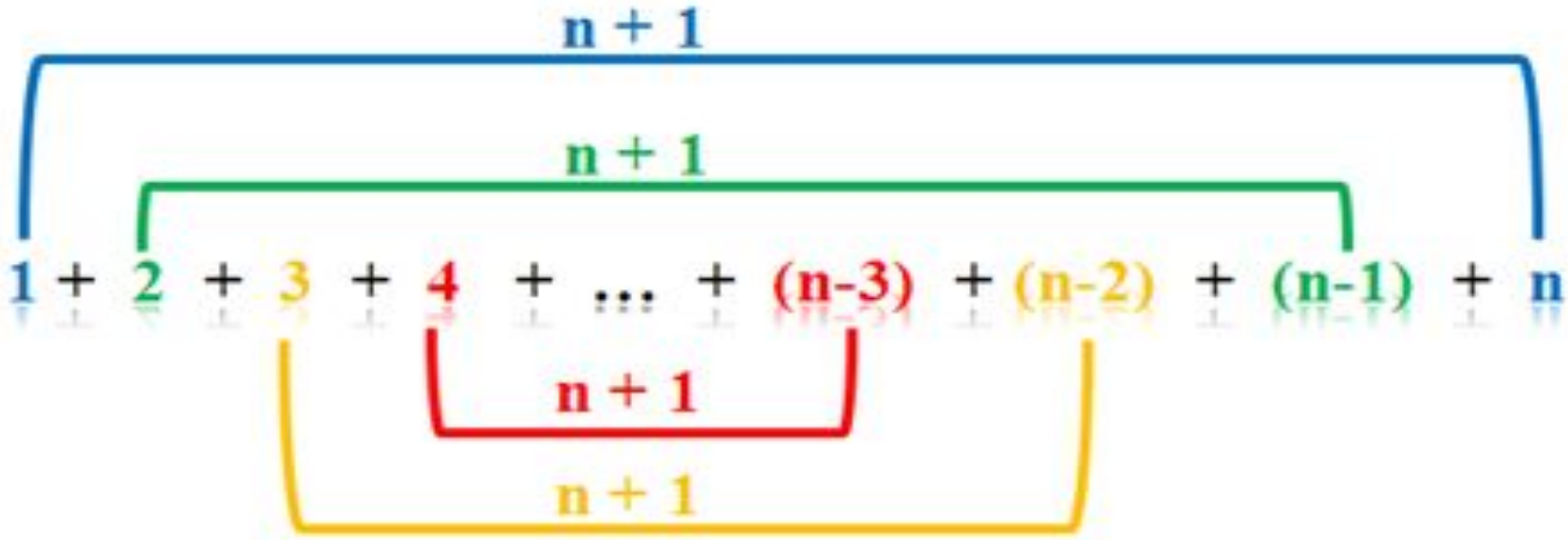
Análisis Empírico de Algoritmos



http://mathandmultimedia.com/2010/09/15/sum-first-n-positive-integers/

Si n es par, hay n / 2 pares => $\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$

Análisis Empírico de Algoritmos



http://mathandmultimedia.com/2010/09/15/sum-first-n-positive-integers/

Si n es impar, hay (n-1) / 2 pares =>

$$\sum_{k=1}^{n} k = (n-1)(n+1)/2 + (n+1) = n(n+1)/2$$

Análisis Empírico de Algoritmos

- Ya puedes implementar la solución de Gauss
- Ejecuta el programa para diferentes valores de n y mide el tiempo de ejecución
- A continuación, represente gráficamente el resultado y compáralo con la solución anterior

Análisis Empírico de Algoritmos

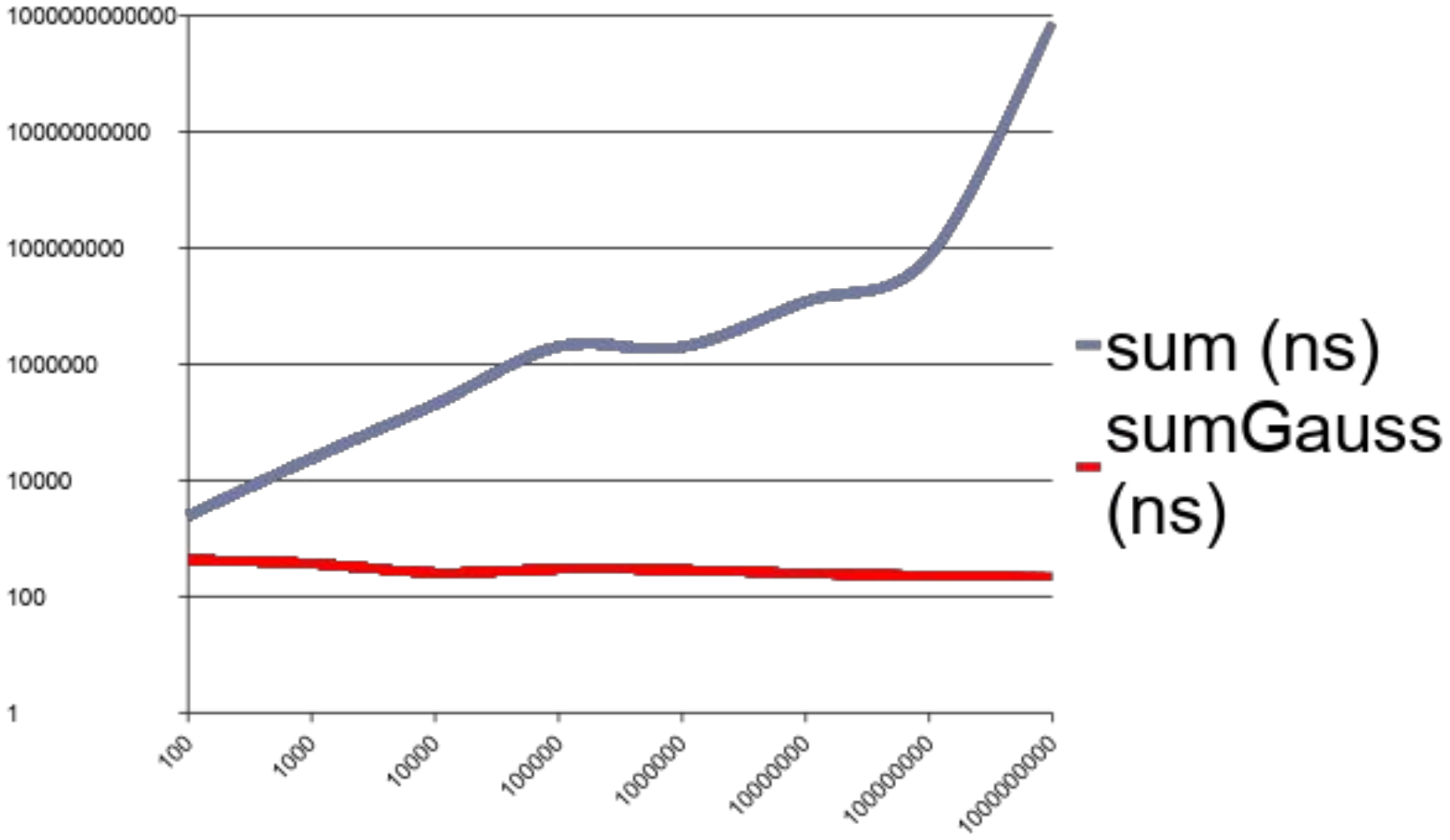
```
public static long sumGauss(long n) {
    long startTime = System.currentTimeMillis();

    long result=n*(n+1)/2;

    long endTime = System.currentTimeMillis();
    long total=endTime - startTime;

    System.out.println("sum("+n+")  took "+ total + " ms");

    return result;
}
```

Análisis Empírico de Algoritmos

```
long MAX=1000000000;
for (int n=100; n<=MAX; n=n*10)
    sumGauss(n);
```

n	tiempo (ns)
100	436
1.000	371
10.000	259
100.000	298
1.000.000	290
10.000.000	250
100.000.000	233
1.000.000.000	222

Análisis Empírico de Algoritmos



Análisis Empírico de Algoritmos

- Sin embargo, algunas desventajas:

1. Necesitas implementar el algoritmo
2. Los resultados pueden no ser indicativos para otras entradas
3. Mismo entorno para comparar dos algoritmos

Outline

- Análisis de Algoritmos
  - Análisis Empírico de Algoritmos
  - Análisis Teórico de Algoritmos
    - Función de tiempo de ejecución

Análisis Teórico de Algoritmos

- Toma en cuenta todas las entradas posibles
- **Pseudocódigo**
  - Define **T(n)**, función del tiempo de ejecución
  - Buscamos la independencia con el entorno de hardware / software

Análisis Teórico de Algoritmos

- Ejecución de la función del tiempo T(n)
  - Representar el tiempo de ejecución de un algoritmo en función del tamaño de entrada
  - T(n)= **número de operaciones ejecutadas por un algoritmo para procesar una entrada según tamaño**

Análisis Teórico de Algoritmos

- Las operaciones primitivas toman una cantidad constante de tiempo
- Ejemplos:
  - Declaring a variable: *int x;*
  - Evaluating an expression: *x+3*
  - Assigning a value to a variable: *x=2*
  - Indexing into an array: *vector[3]*
  - Calling a method: *sumGauss(n)*
  - Returning from a method: *return x;*

Análisis Teórico de Algoritmos

- **Reglas generales para la estimación:**
  - **Declaraciones consecutivas:** el tiempo de ejecución es igual a la suma de los tiempos de ejecución de las distintas instrucciones
  - **Bucles:** el tiempo de ejecución de un bucle es como máximo el tiempo de ejecución de las instrucciones dentro de ese bucle multiplicado por el número de iteraciones
  - **Bucles anidados:** el tiempo de ejecución de un bucle anidado que contiene una instrucción en el bucle más interno es el tiempo de ejecución de la instrucción multiplicado por el producto del tamaño de todos los bucles
  - **If/Else**: el tiempo de ejecución de una instrucción selectiva es, como máximo, el tiempo de las condiciones y el mayor de los tiempos de ejecución de los bloques de instrucciones asociadas

Análisis Teórico de Algoritmos

```
public static long sum(long n) {
    long result=0;
    for (long i=1; i<=n; i++) {
        result = result + i;
    }
    return result;
}
```

operaciones
2
2+n+n
n
1

$$T_{Sum}(n) = 3n + 5$$

Este algoritmo requiere 3n + 6 ns para una entrada de tamaño n

Análisis Teórico de Algoritmos

```
public static long sumGauss(long n) {
    long result=n*(n+1)/2;
    return result;
}
```

 # operaciones

2

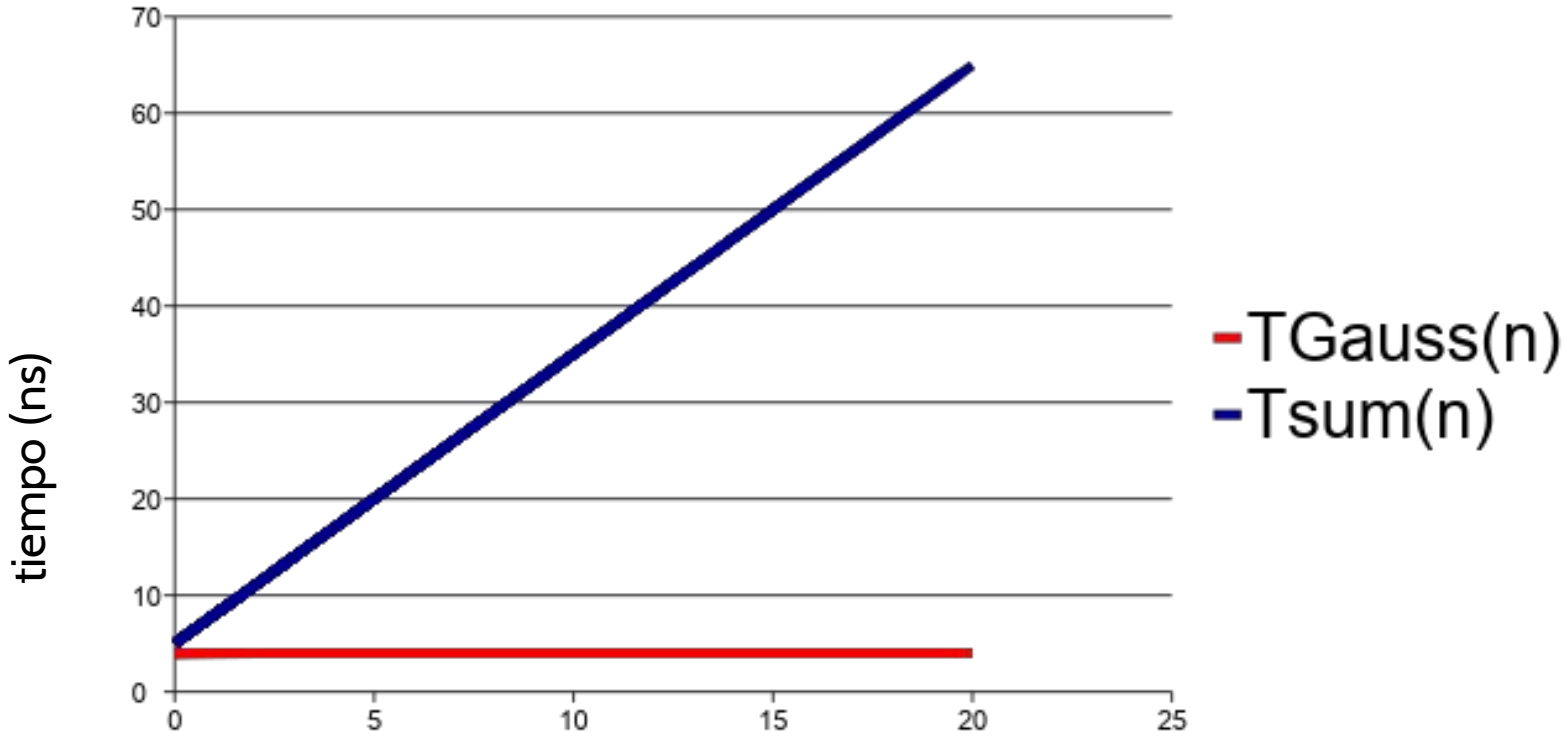
1

$$T_{Gauss}(n) = 3$$

La solución de Gauss requiere 3 ns para cualquier entrada

Análisis Teórico de Algoritmos

Los requisitos de tiempo en función del tamaño del problema n



$T_{sum}(n)=3n+5$

$T_{Gauss}(n)=4$

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;
}
```

	#
int index=-1;	2
int n=v.length;	2
int i=0;	2
i<n && index==-1	n+1
i++	n
if (x==v[i]) index=i;	3n
T(n)	5n+7

Análisis Teórico de Algoritmos

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

Análisis Teórico de Algoritmos

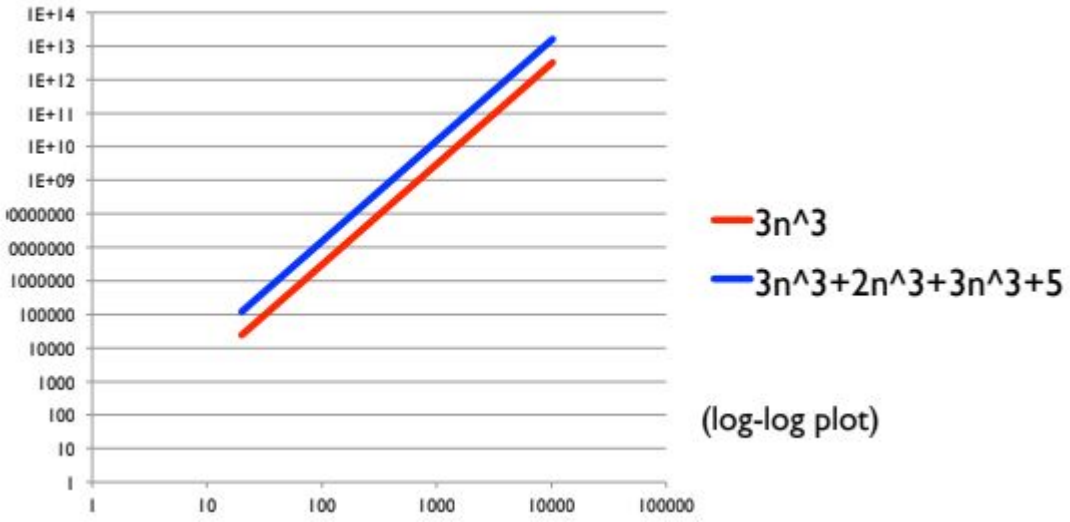
- T(n) también depende de:

1) El ordenador en el que se ejecuta el programa
2) El compilador utilizado para generar el programa

➢ Encontrar una función de aproximación para T(n), una cota superior (Big-Oh)

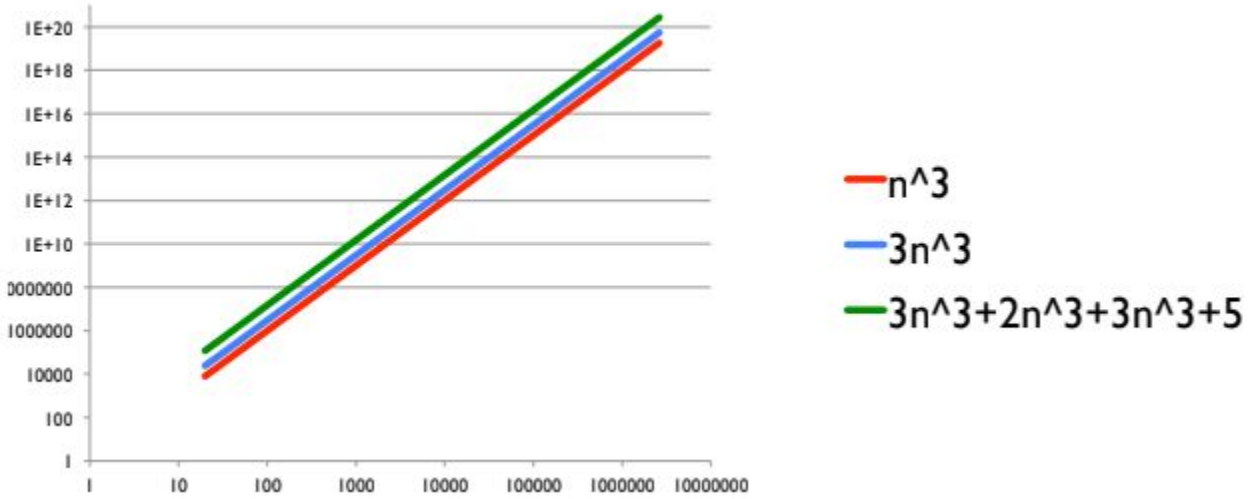
Análisis Teórico de Algoritmos

- Ignorar términos de orden inferior :
  - Si n es pequeño, no nos importa
  - Si n es grande, los términos inferiores son insignificantes



Análisis Teórico de Algoritmos

- Ignorar términos de orden inferior
- Establecer el coeficiente del término a1



Análisis Teórico de Algoritmos

- Algunos ejemplos:

T(n)	Cota superior
$n + 2$	$\sim n$
$\frac{1}{2}(n+1)(n-1)$	$\sim n^2$
$3n+\log(n)$	$\sim n$
$n(n-1)$	$\sim n^2$
$7n^4+5n^2+1$	n^4

Análisis Teórico de Algoritmos

- Buenas noticias: un pequeño conjunto de funciones:

$$1 < \log n < n < n\log n < n^2 < n^3 < \ldots < 2^n$$

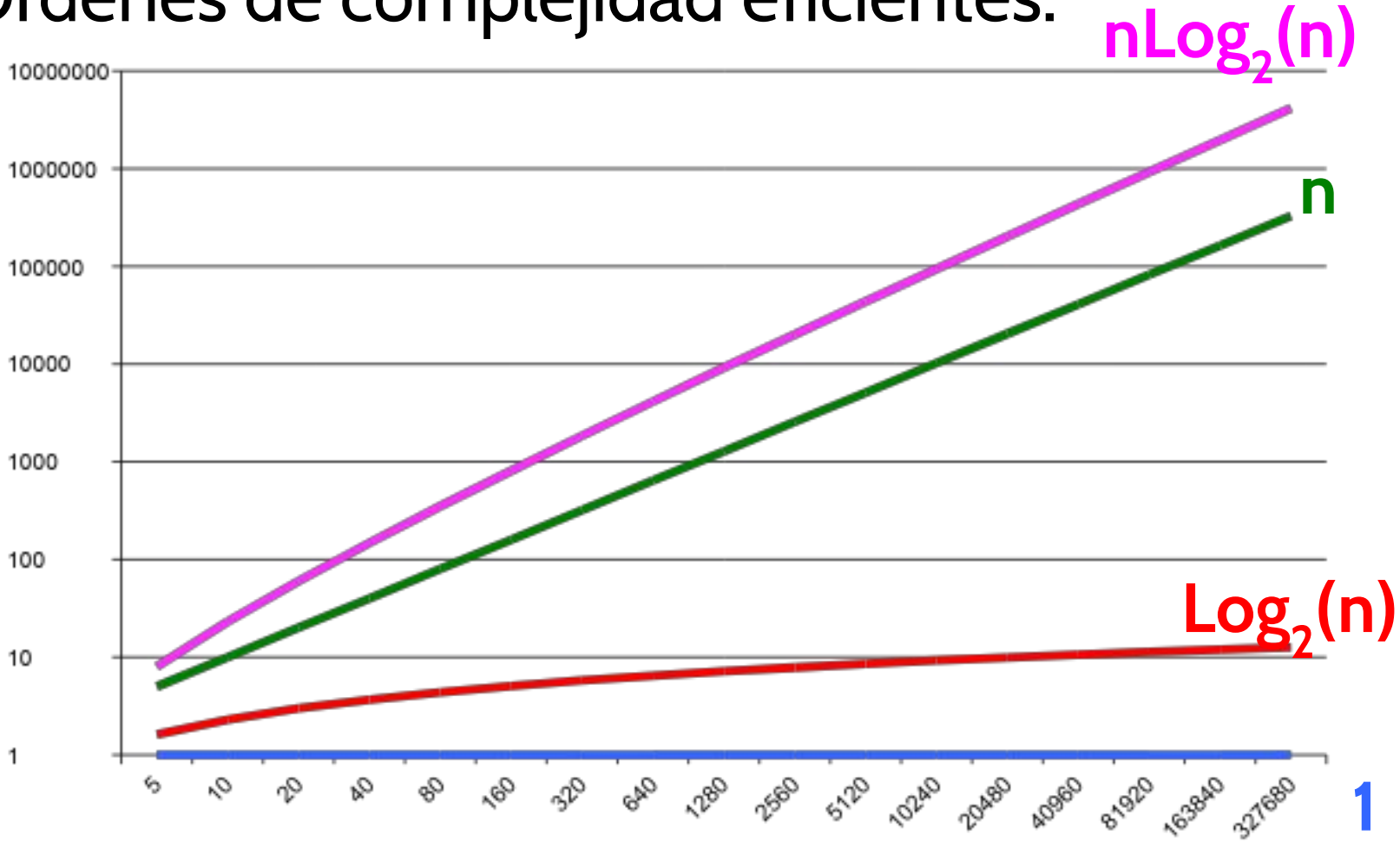
Análisis Teórico de Algoritmos

- Ordenes de complejidad eficientes:

Order	Nombre	Descripción	Ejemplo
1	Constante	Independiente del tamaño de entrada	Eliminar el primer elemento de una cola
$Log_2(n)$	Logarítmica	Dividir a la mitad	Búsqueda binaria
n	Lineal	Bucle	Suma de elementos de array
$nLog_2(n)$	Lineal Logarítmica	Divide y vencerás	Mergesort, quicksort

Análisis Teórico de Algoritmos

- Órdenes de complejidad eficientes:



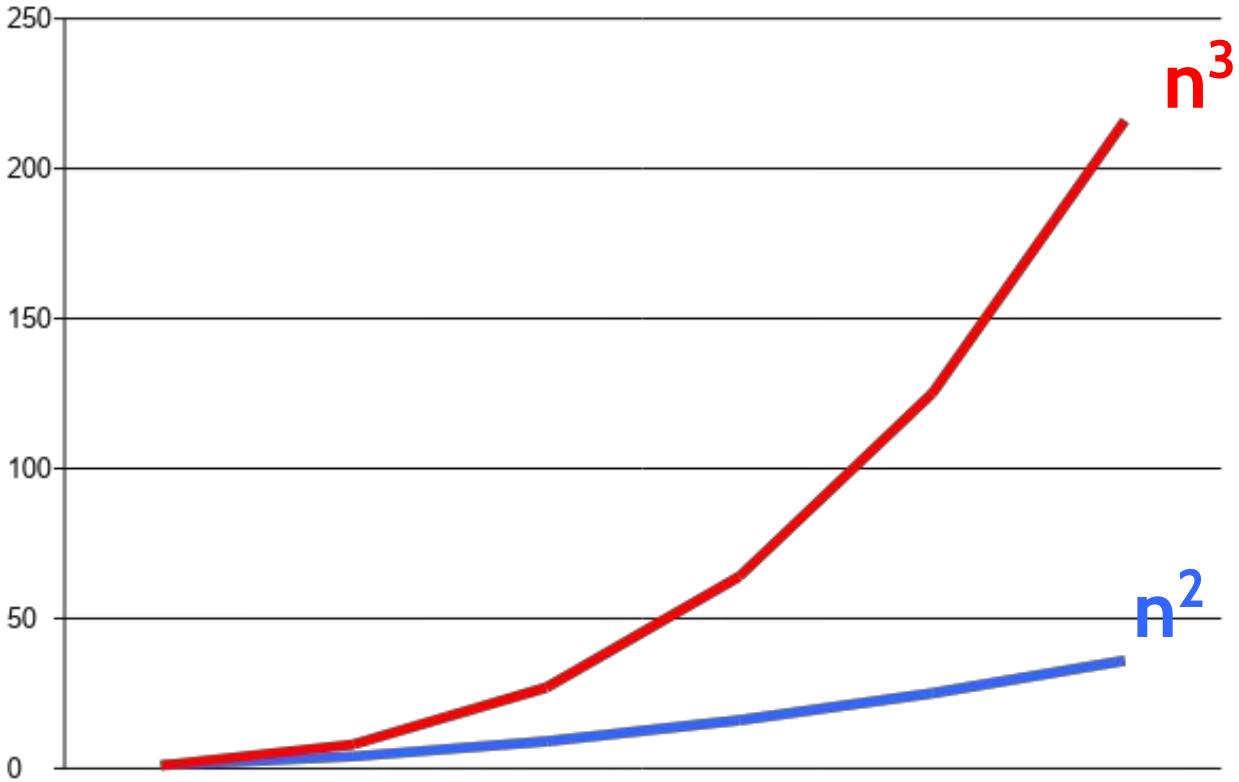
Análisis Teórico de Algoritmos

- Órdenes de complejidad manejables:

Order	Nombre	Descripción	Ejemplo
n^2	Cuadrática	Doble bucle	Agrega dos matrices; ordenamiento de burbuja
n^3	Cúbica	Triple bucle	Multiplicar dos matrices

Análisis Teórico de Algoritmos

- Órdenes de complejidad manejables:



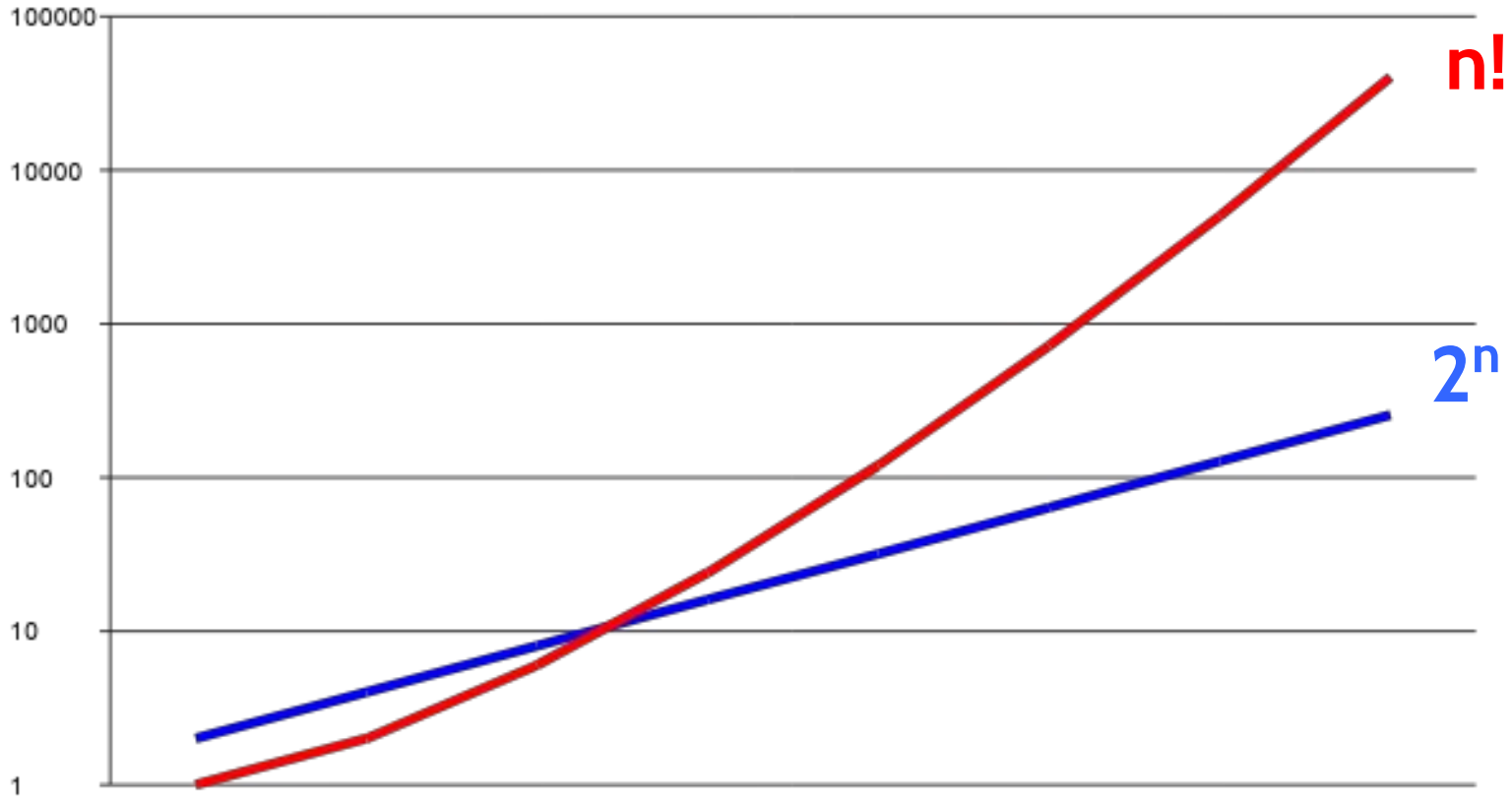
Análisis Teórico de Algoritmos

- Órdenes de complejidad no manejables:

Order	Nombre	Descripción	Ejemplo
k^n	Exponencial	Búsqueda exhaustiva	Adivinar una contraseña
n!	Factorial	Búsqueda de fuerza bruta	Enumerar todas las particiones de un conjunto

Análisis Teórico de Algoritmos

- Órdenes de complejidad no manejables:



uc3m | Universidad Carlos III de Madrid