

OpenCourseWare
Grado Ingeniería Informática
Estructura de Datos y Algoritmos

Tema 3 Análisis de Algoritmos

Análisis de Algoritmos

- Un **algoritmo** es un conjunto de pasos (instrucciones) para resolver un problema.
 - Debe ser robusto y correcto!!!.
 - Debe ser eficiente!!!.
- El **análisis de algoritmos** nos va a permitir **estimar cómo de eficiente** es un algoritmo.
- Un problema puede tener diferentes soluciones (algoritmos). El análisis de algoritmos nos va a permitir **comparar algoritmos y elegir el más eficiente**.

Objetivos

- 1) Determinar empíricamente la complejidad temporal de algoritmos sencillos.
- 2) Estimar la complejidad asintótica (Big-O) de algoritmos.
- 3) Comparar y clasificar algoritmos de acuerdo a su complejidad temporal.
- 4) Diferenciar conceptos de mejor y peor caso de la complejidad de un algoritmo.

Índice

- **Complejidad temporal y Complejidad Espacial**
- **Tipos Análisis de Algoritmos**
 - Análisis empírico
 - Análisis teórico

Complejidad Temporal y Espacial.

- ¿Cómo estudiar el rendimiento de un algoritmo?:
 - **Complejidad temporal:** estima el tiempo requerido por un algoritmo.
 - **Complejidad espacial:** estima el espacio en memoria principal que necesita utilizar el algoritmo.
 - Es posible estudiar el coste de otros recursos: internet, impresora, etc.



Índice

- Complejidad temporal y Complejidad Espacial
- **Tipos Análisis de Algoritmos**
 - Análisis empírico
 - Análisis teórico

Análisis Empírico

- En el análisis empírico, se calcula cuánto tiempo tarda un algoritmo en resolver un problema para distintos tamaños de entrada. Pasos:
 1. Implementar el algoritmo
 2. Incluir instrucciones que permitan medir el tiempo de ejecución.
 3. Ejecutar el programa con entradas de diferentes tamaño y obtener los tiempos de ejecución para cada tamaño
 4. Importa los resultados a un fichero excel y mostrarlos en un gráfico (X Y dispersión)

Análisis Empírico

- Dado un número n , desarrolla una función que sume los n primeros números:
 - 1) Implementa el algoritmo:

```
def sum_n(n: int) -> int:
    """devuelve los n primeros números naturales"""

    result=0

    for i in range(1,n+1):
        result += i

    return result
```


Análisis Empírico

2) Incluye instrucciones para medir el tiempo:

```
import time

def sum_n(n: int) -> int:
    """devuelve los n primeros números naturales.
    También vamos a devolver el tiempo que tarda en ejecutarse"""

    start = time.time() # devuelve la hora actual en segundos
    result=0

    for i in range(1,n+1):
        result+=i

    end = time.time()

    # transformo tiempo a milisegundos
    return result, (end-start)*1000
```

Análisis Empírico

3) Ejecuta el programa para tamaños diferentes

```
for i in range(1,8):  
    n=10**i  
    print(n, sum_n(n))
```

```
10 (55, 0.005245208740234375)  
100 (5050, 0.008344650268554688)  
1000 (500500, 0.06866455078125)  
10000 (50005000, 0.7307529449462891)  
100000 (5000050000, 11.969804763793945)  
1000000 (500000500000, 104.25615310668945)  
10000000 (50000005000000, 1153.6884307861328)
```



Análisis Empírico

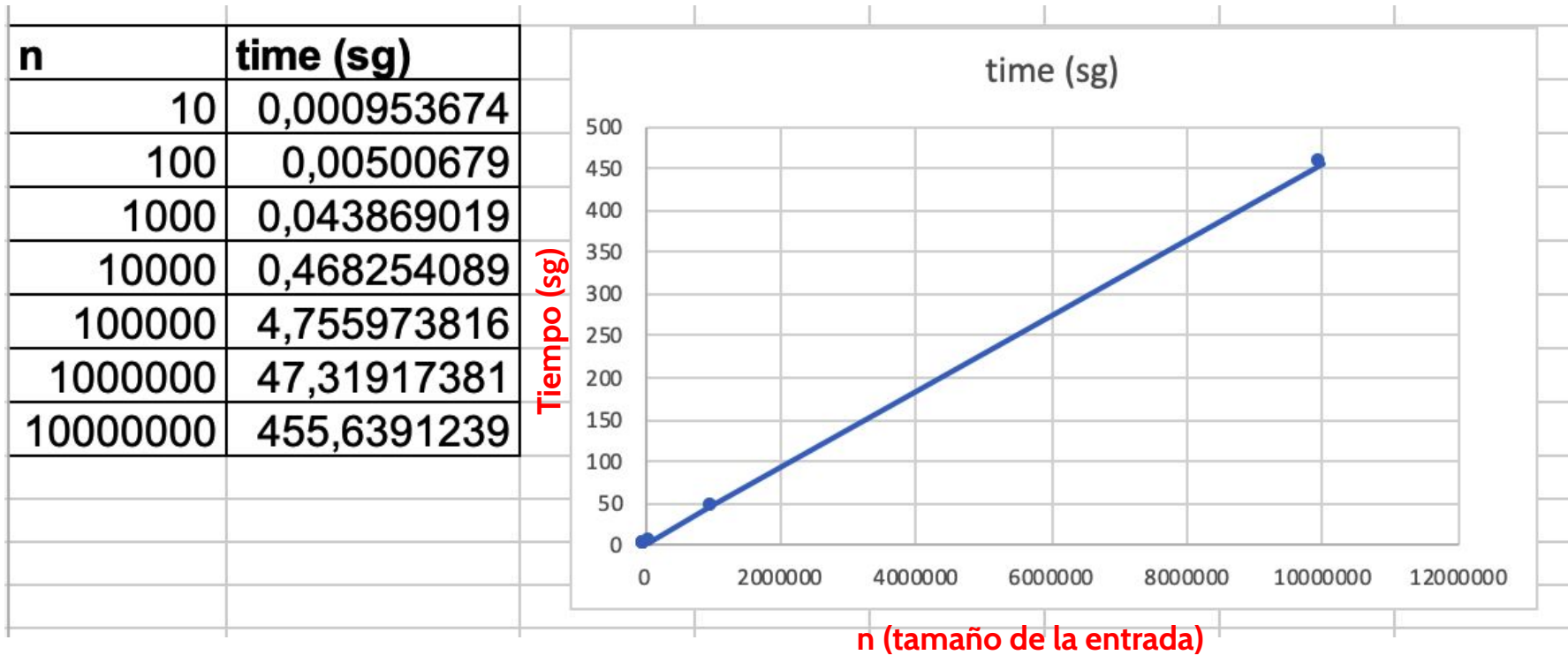
3) Ejecuta el programa para tamaños diferentes (mostramos el tiempo en milisegundos)

```
for i in range(1,8):  
    n=10**i  
    print(n, sum_n(n))
```

```
sumN(10)=55, time=0.00095367431640625  
sumN(100)=5050, time=0.0040531158447265625  
sumN(1000)=500500, time=0.04291534423828125  
sumN(10000)=50005000, time=0.45299530029296875  
sumN(100000)=5000050000, time=4.713773727416992  
sumN(1000000)=500000500000, time=49.21984672546387  
sumN(10000000)=50000005000000, time=456.09593391418457
```

Análisis Empírico

4) Importa los resultados a un fichero excel para mostrarlos en un gráfico

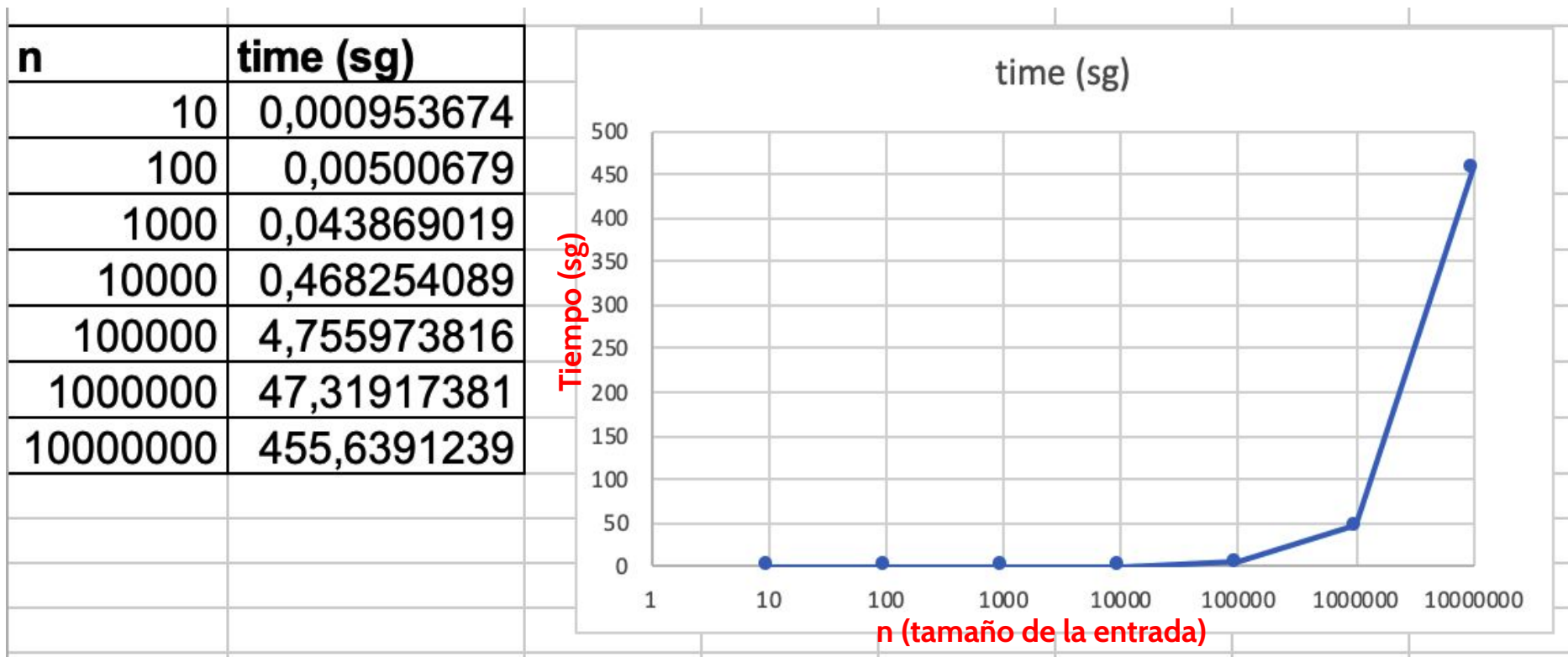


Análisis Empírico

- Si tus datos tienen valores que varían en varios órdenes de magnitud, la escala logarítmica puede ser útil para visualizar patrones y tendencias que de otro modo serían difíciles de ver en una escala lineal.
- En el ejemplo anterior, los valores de x abarcan un amplio rango de valores (mientras que el rango de valores para y es muy pequeño).
- En estos casos, es útil transformar el gráfico a **escala logarítmica** (haz doble click en la escala del eje X (n), y selecciona la escala logarítmica).

Análisis Empírico

4) Importa los resultados a un fichero excel para mostrarlos en un gráfico (escala logarítmica sobre el eje x)



Análisis Empírico

¿Existen otros algoritmos que resuelvan este problema?



Análisis Empírico

- ¿Existen otros algoritmos que resuelvan este problema?

Suma de Gauss:

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

- Realicemos su análisis empírico, siguiendo los cuatro pasos anteriores

Análisis Empírico

- 1) Implementa el algoritmo y 2) incluye instrucciones para medir el tiempo:

```
import time

def sumNGauss(n):
    start=time.time()

    result=n*(n+1)/2
    end=time.time()
    return result, (end-start)*1000

print('\n\tsumNGauss(n)')
for i in range(1,8):
    n=10**i
    _,ms=sumNGauss(n)

    print('{}\t{}'.format(n,ms))
```

Análisis Empírico

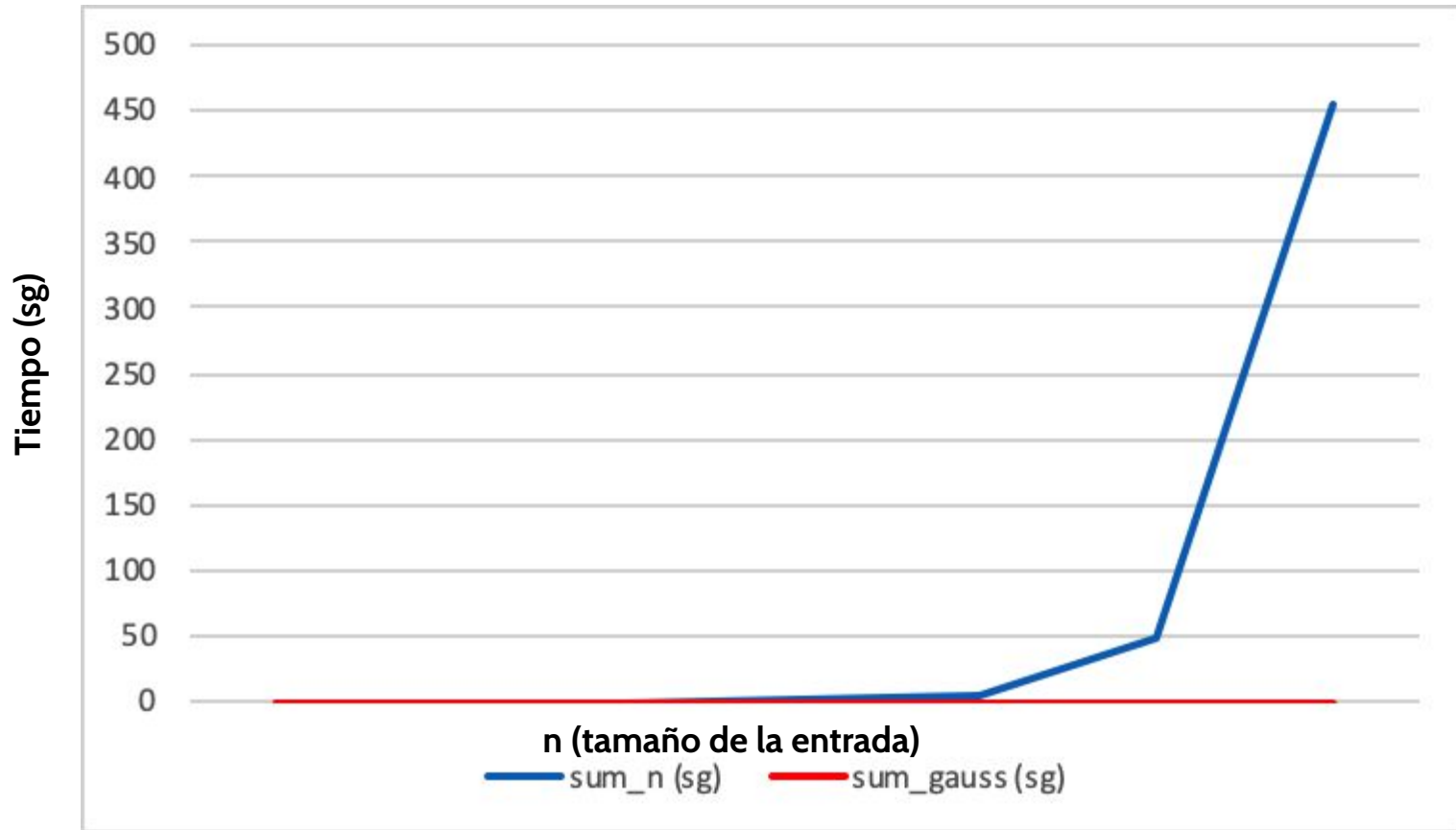
```
for i in range(1,8):  
    n = 10 ** i  
    print(n, sum_gauss(n))
```

```
10 (55.0, 0.0016689300537109375)  
100 (5050.0, 0.00095367431640625)  
1000 (500500.0, 0.000476837158203125)  
10000 (50005000.0, 0.0002384185791015625)  
100000 (5000050000.0, 0.0007152557373046875)  
1000000 (500000500000.0, 0.000476837158203125)  
10000000 (50000005000000.0, 0.0002384185791015625)
```

- Podemos observar que el tiempo se mantiene casi constante con valores muy pequeños (próximos a 0 segundos).



Análisis Empírico



La gráfica muestra claramente que la solución de Gauss es más eficiente que el primer algoritmo.

Análisis Empírico

- ¿Cuáles son las ventajas del análisis empírico?
- ¿Cuáles son sus desventajas?



Análisis Empírico - Ventajas

- El proceso es sencillo (medir el tiempo para distintos tamaños de la entrada).
- Nos permite obtener gráficas para razonar sobre el tiempo de ejecución de un algoritmo.
- Las gráficas nos ayudan a comparar fácilmente el tiempo de ejecución de distintos algoritmos para un mismo problema.



Análisis Empírico - Desventajas

- Sin embargo, algunas **desventajas**:
 - Es necesario implementar los algoritmos.
 - Mismo entorno (hardware + software) para comparar los algoritmos.
 - La gráfica se construye sobre un conjunto finito de valores de n . Los resultados podrían no ser representativos para todas las posibles entradas.



Índice

- Análisis de Algoritmos
 - Análisis empírico
 - **Análisis teórico**

Análisis Teórico

- Usa pseudocódigo (no se implementan los algoritmos).
- Consiste en obtener la **función temporal $T(n)$** , que representa el **número de operaciones** que deben ser ejecutadas para una **entrada de tamaño n** .
- Al ser una función, es posible considerar todas las posibles entradas (tamaños).
- Como no se miden tiempos de ejecución, es independiente del entorno Hardware/Software.

Análisis Teórico - función temporal

- ¿Cómo contamos operaciones / instrucciones en un algoritmo?.
- En cualquier algoritmo, nos podemos encontrar con los siguientes tipos de operaciones:
 - Operaciones primitivas
 - Operaciones bucles
 - Operaciones selectivas o condicionales (if)

Análisis Teórico - operaciones primitivas

- Ejemplos:

- Asignar valor a una variable: **$x=2$**
- Indexar un elemento en un array: **$vector[3]$**
- Devolver un valor en una función: **$return\ x$**
- Evaluar una expresión aritmética: **$x+3$**
- Evaluar una expresión lógica: **$0<i<index$**

Este tipo de operaciones se van a contabilizar como una operación.

Análisis Teórico - operaciones primitivas

- ¿Qué ocurre si una operación primitiva se puede descomponer de varias operaciones primitivas?

return vector[3] + vector[5]

- En este caso, podríamos contar:
 - 1 operación por acceder a vector[3]
 - 1 operación por acceder a vector[5]
 - 1 operación por la operación de suma
 - 1 operación por el return.
- Es decir, en total tendríamos cuatro operaciones.

Análisis Teórico - operaciones primitivas

Operación	# operaciones primitivas
<code>x = 2</code>	
<code>x = y + 3</code>	
<code>return a[0] + 1</code>	
<code>print('Hola')</code>	
<code>return node is None or node.elem > 5</code>	

Análisis Teórico - operaciones primitivas

Operación	# operaciones primitivas
<code>x = 2</code>	1
<code>x = y + 3</code>	2 (1 suma + 1 asignación)
<code>return a[0] + 1</code>	3 (1 acceso + 1 suma + 1 return)
<code>print("Hola")</code>	1
<code>return node is None or node.elem >5</code>	5 (1 node is None + 1 obtener node.elem + 1 comparar node.elem >5 + 1 operación or + 1 return)



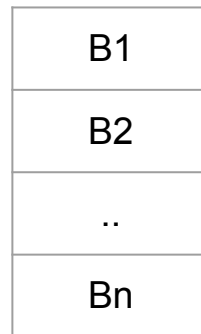
Análisis Teórico - operaciones primitivas

- La función temporal de una operación primitiva siempre tiene un valor constante, que no depende de n , el tamaño de la entrada.

$$T(n) = c$$

Análisis Teórico

- Si nuestro algoritmo se compone de varios bloques, su función $T(n)$, será la suma de las funciones $T(n)$ de cada bloque.



$$T(n) = T(B1) + T(B2) + \dots + T(Bn)$$

Análisis Teórico

Algorithm <i>swap</i> (a, b)	<u># operations</u>
temp=a	1
a=b	1
b=temp	1

$$T(n) = 1+1+1 = 3$$

Análisis Teórico - bucle

- La función $T(n)$ para un bucle se calculará como el número de veces que se ejecuta el bucle (número de iteraciones) por la función $T(n)$ del bloque interno B .

while condition:
 B

for x in ... :
 B

$$T_{\text{loop}}(n) = T(B) * \text{número de iteraciones}$$

Análisis Teórico - bucle

- ¿Cuántas veces se ejecuta el bucle (número de iteraciones)?.

```
for i in range(1, n+1) :  
    result = result + i
```

Análisis Teórico - bucle

- ¿Cuántas veces se ejecuta el bucle (número de iteraciones)?

```
for i in range(1, n+1):  
    result = result + i
```
- La función `range(n+1)` devuelve los siguientes valores: 1, 2, ..., n. Por tanto, el número de iteraciones será n.
- La función $T(n)$ del bloque interno (`result=result+i`) es $T(n) = 2$.
- Por tanto, la función $T(n) = n * 2$.
- En este caso, la función temporal sí depende del valor de n.



Análisis Teórico - bucles anidados

- Dado el siguiente bucle anidado:

```
for i in range(n):  
    for i in range(n):  
        print(i*j) #T(n) = 2
```

Análisis Teórico - bucles anidados

- Dado el siguiente bucle anidado:

```
for i in range(n):  
    for i in range(n):  
        print(i*j) #T(n) = 1+1 =2
```

- Su función temporal $T(n)$ es:

$$T(n) = n * n * 2 = 2n^2$$

- Es decir, 2 de la instrucción que hay en el bucle interno, por el número de iteraciones del bucle interno, por el número de iteraciones del bucle externo.

Análisis Teórico - bucles anidados

- ¿Qué pasa si cada bucle se ejecuta un número distinto de veces?

```
for i in range(n1):  
    for i in range(n2):  
        print(i*j)
```

Análisis Teórico - bucles anidados

- En este caso, vamos a definir $n = \max(n_1, n_2)$,
- Podemos hacer la siguiente simplificación:

$$T(n) = n_1 * n_2 * 2 < n * n * 2 = 2n^2$$

```
for i in range(n1):  
    for i in range(n2):  
        print(i*j)
```

Análisis Teórico - operaciones condicionales

- **If-Else:** Sólo uno de los bloques ($B_1, B_2, \dots B_k$) se ejecutará.

```
if condition1:  
    B1  
elif condition2:  
    B2  
    ...  
else:  
    Bk
```


Análisis Teórico - operaciones condicionales

- **If-Else:** Sólo uno de los bloques ($B_1, B_2, \dots B_k$) se ejecutará.

```
if condition1:  
    B1  
elif condition2:  
    B2  
    ...  
else:  
    Bk
```

- Siempre debemos ser pesimistas, y considerar el peor caso ,es decir, el bloque B_i con la mayor función temporal.

Análisis Teórico - operaciones condicionales

- **If-Else:** Sólo uno de los bloques (B_1, B_2, \dots, B_k) se ejecutará.

```
if condition1:  
    B1  
elif condition2:  
    B2  
    ...  
else:  
    Bk
```

$$T_{\text{if-else}}(n) = \max(T_{B_1}(n), T_{B_2}(n), \dots, T_{B_k}(n))$$

Análisis Teórico - operaciones condicionales

- También se deberían añadir las funciones temporales de las condiciones. En el peor de los casos, se evaluarán todas las condiciones.

```
if condition1:  
    B1  
elif condition2:  
    B2  
    ...  
else:  
    Bk
```

$$T_{\text{if-else}}(n) = \sum T_{\text{condicioni}}(n) + \max(T_{B_1}(n), T_{B_2}(n), \dots, T_{B_k}(n))$$

Análisis Teórico - operaciones condicionales

```
if opc = "inc":  
    n = n + 1 #B1, TB1(n) = 2  
elif opc = "dec":  
    n = n - 1 #B2, TB2(n) = 2  
elif opc = "mostrar":  
    for i in range(1, n+1): #B3, TB3(n) = n*1  
        print(i)  
else:  
    print("Error: opc!!") #B4, TB4(n) = 1
```

- En este ejemplo, el bloque con la mayor función temporal es B₃

Análisis Teórico - operaciones condicionales

```
if opc = "inc":  
    n = n + 1 #B1, TB1(n) = 2  
elif opc = "dec":  
    n = n - 1 #B2, TB2(n) = 2  
elif opc = "mostrar":  
    for i in range(1, n+1): #B3, TB3(n) = n*1  
        print(i)  
else:  
    print("Error: opc!!") #B4, TB4(n) = 1
```

- En este ejemplo, el bloque con la mayor función temporal es B₃. La función temporal del if (si consideramos las condiciones):
- $T(n) = 1 + 1 + 1 + TB_3(n) = 3 + n$

Análisis Teórico

- Las funciones de tiempo de ejecución nos permiten comparar algoritmos, sin necesidad de implementarlos.
- Vamos a comparar las funciones de los algoritmos `sum_n` y `sum_gauss` (aunque estás ya las tenemos implementadas).

Análisis Teórico

```
def sum_n(n: int) -> int:
    """devuelve los n primeros números naturales"""

    result=0          # 1 operación

    for i in range(1,n+1):
        result += i    # La T del bucle será 2*n

    return result     # 1 operación
```

- Por tanto, $T(n) = 1+2n+1 = 2n + 2$
- Depende del valor de n.

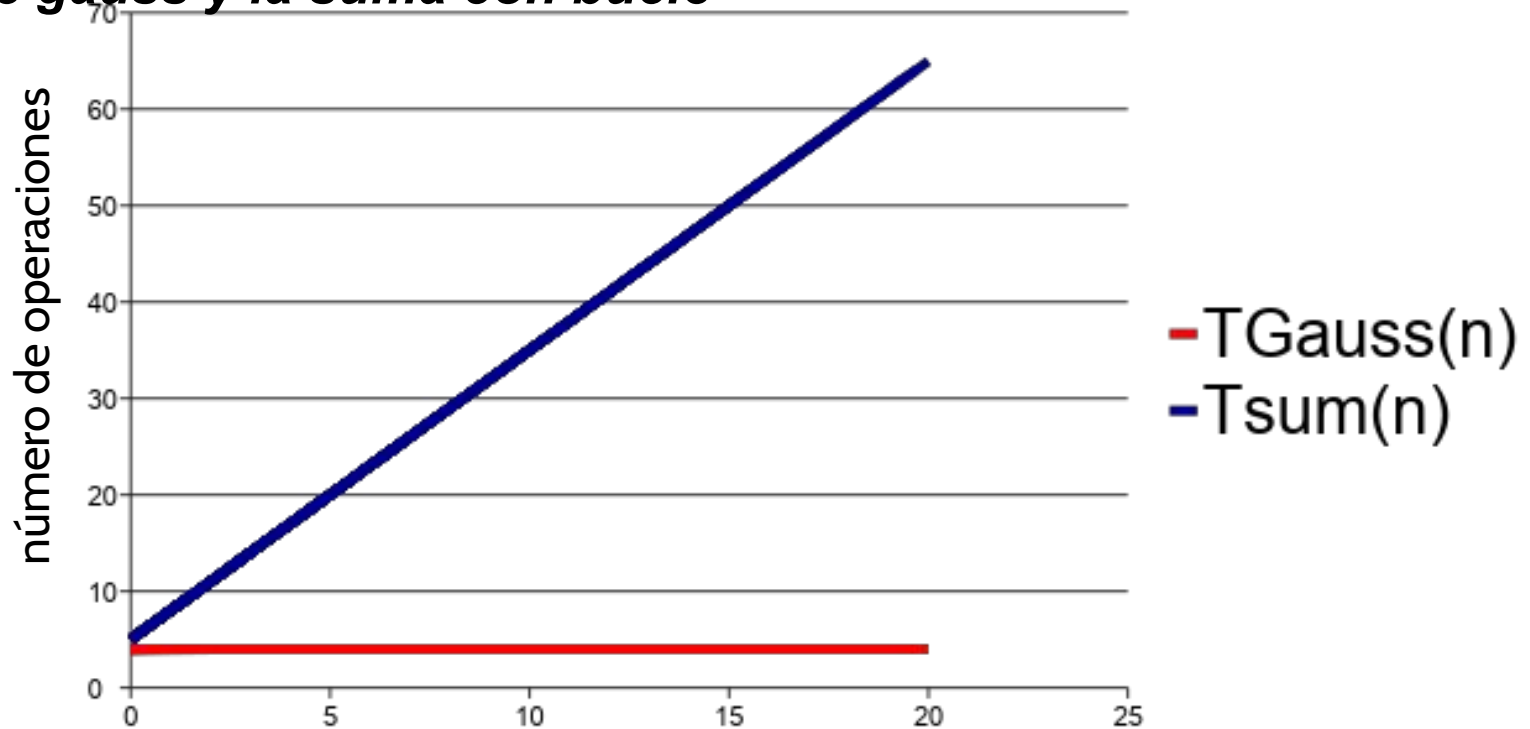
Análisis Teórico

```
def sum_gauss(n: int) -> int:  
    """returns the sum of the first n numbers, by  
    applying Gauss sum"""  
    return n*(n+1) // 2
```

- En este caso, $T(n) = 1 +$ (suma)
1 + (multiplicación)
1 + (división entera)
1 (return)
= 4
- $T(n) = 4$, no depende del valor de n .

Análisis Teórico

Comparativa de las funciones temporales de la suma de gauss y la suma con bucle



$$T_{\text{sum}}(n) = 2n + 2$$

$$T_{\text{Gauss}}(n) = 4$$



Análisis Teórico

- Podemos afirmar que el algoritmo de gauss es más eficiente porque su tiempo de ejecución será constante, mientras que el tiempo de ejecución del algoritmo basado en el bucle, siempre dependerá del valor de n .

Análisis Teórico

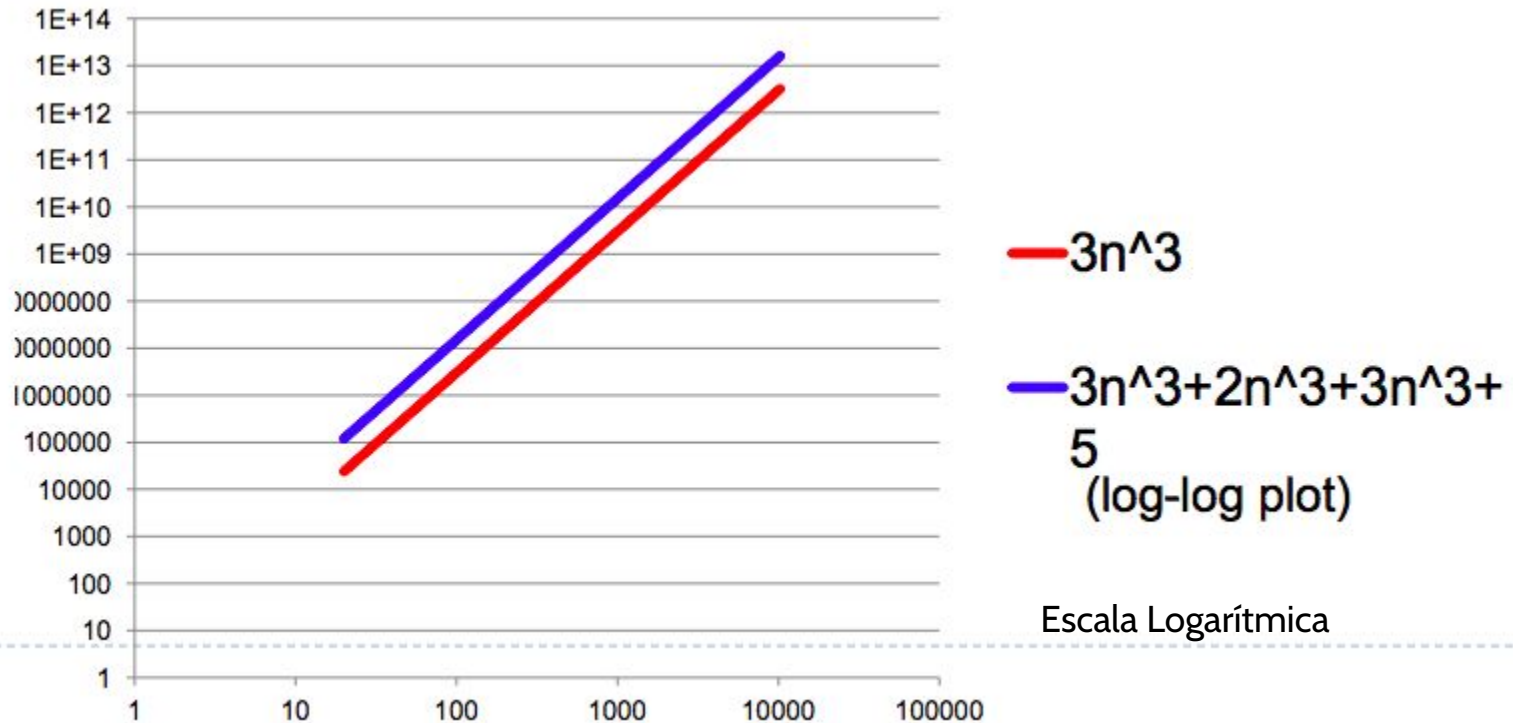
- Supón que tienes dos algoritmos, A y B, con las siguientes funciones de tiempo:
 - $T_A(n) = 3n^3$
 - $T_B(n) = 3n^3 + 2n^2 + 3n + 5$

¿Qué algoritmo es más eficiente?

Análisis Teórico

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

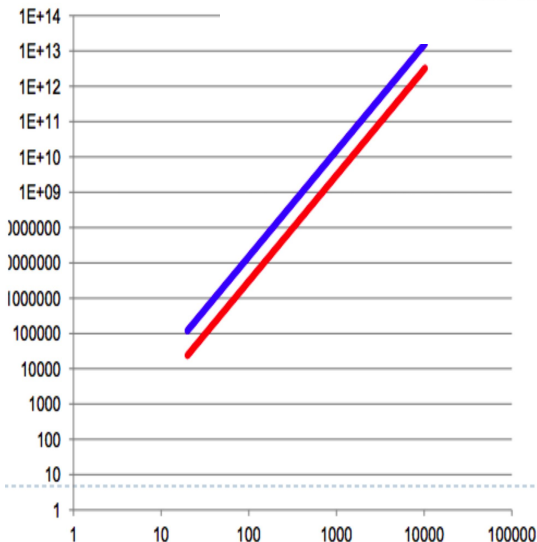
$$T_B(n) = 3n^3 + 2n^2 + 3n + 5$$



Análisis Teórico

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

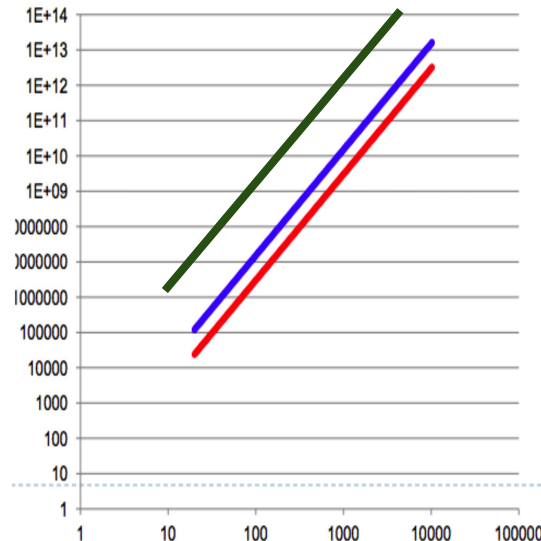
$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 1$$



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

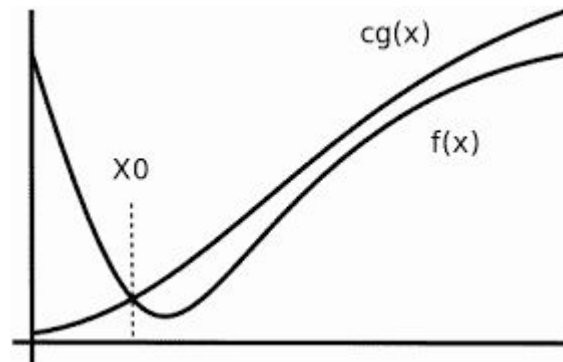
Análisis Teórico

- 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)



Análisis Teórico - cota superior asintótica

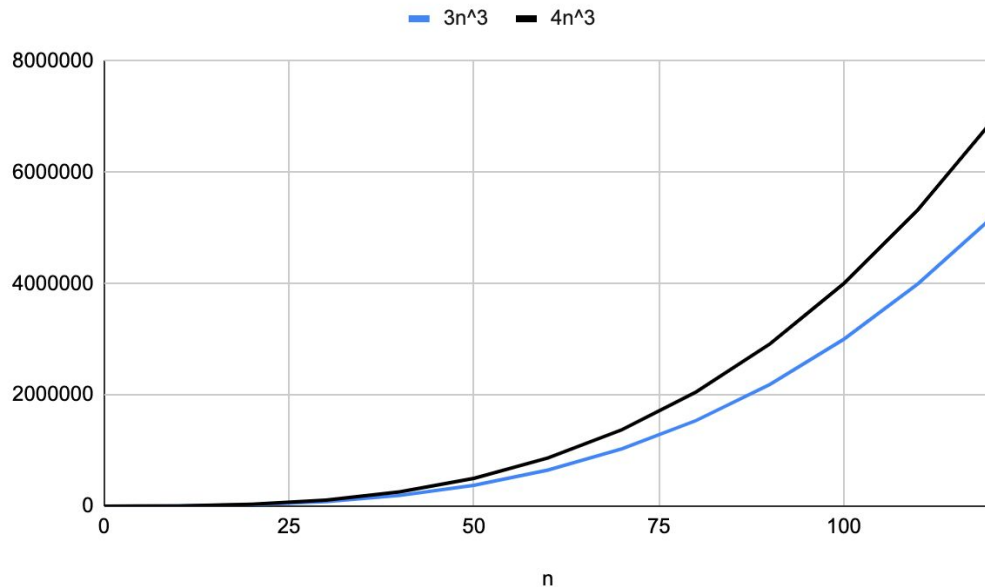
- Dadas dos funciones, $f(n)$ y $g(n)$, $f(n)$ es de **orden superior** $g(n)$ (o $g(n)$ es **cota superior** de $f(n)$), si existen $n_0 > 0$ y $c > 0$, se cumple:
$$f(n) \leq cg(n), \text{ para todo } n \geq n_0$$



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

Análisis Teórico - cota superior asintótica

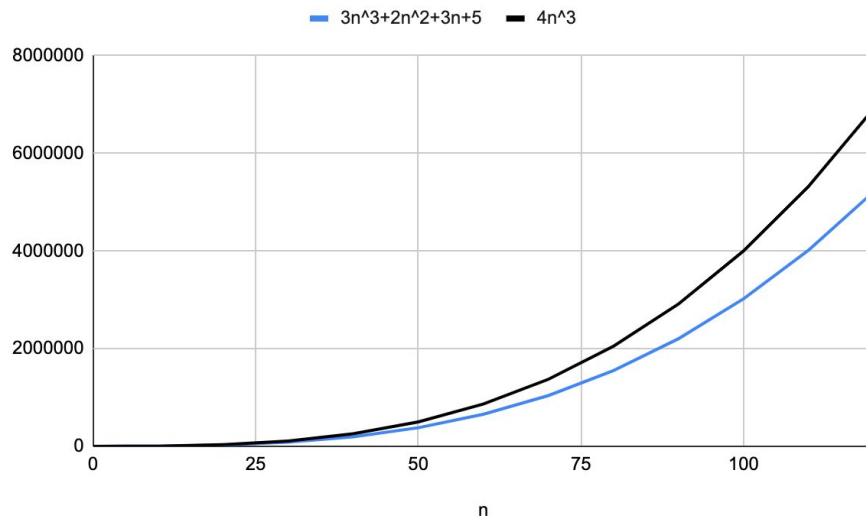
$T_A(n) = 3n^3$ es de orden superior n^3 porque existen $n_0 = 0$, $c = 4$, tales que $T_A(n) \leq cn^3$, para todo $n \geq n_0$



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

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

Análisis Teórico - cota superior asintótica

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

$$1 < \log n < n < n \log n < n^2 < n^c < k^n < n!$$



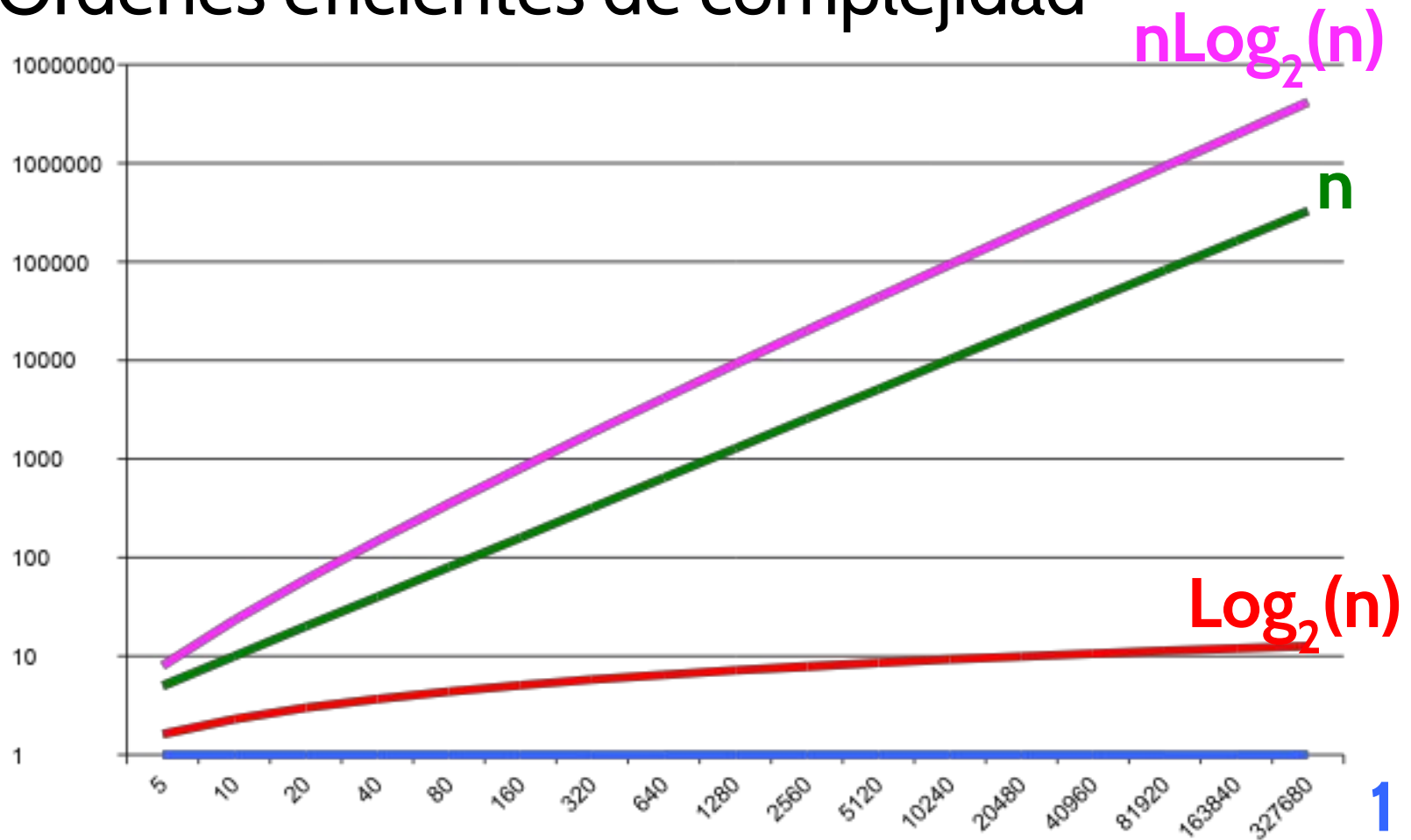
Análisis Teórico - cota superior asintótica

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
$O(k^n)$	exponencial
$O(n!)$	factorial

Análisis Teórico - Big O eficientes

Órdenes eficientes de complejidad



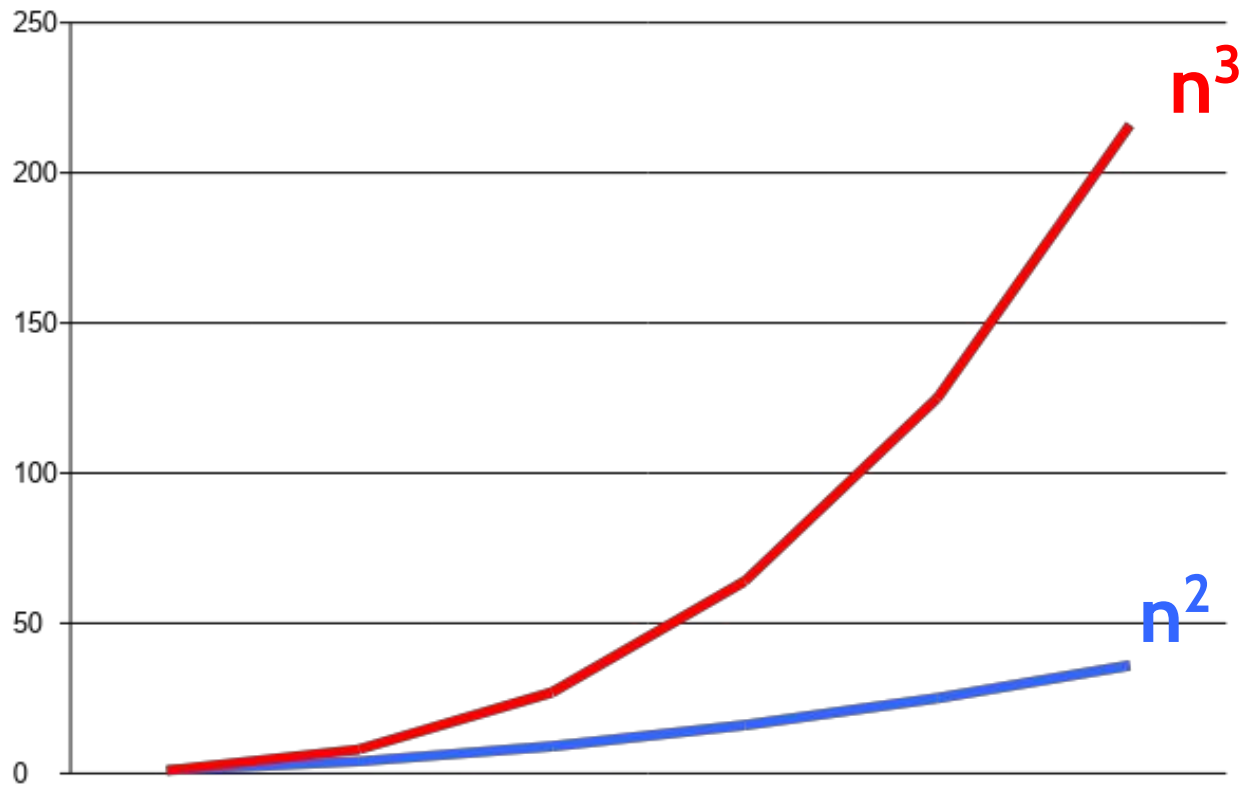
Análisis Teórico - Big O eficientes

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



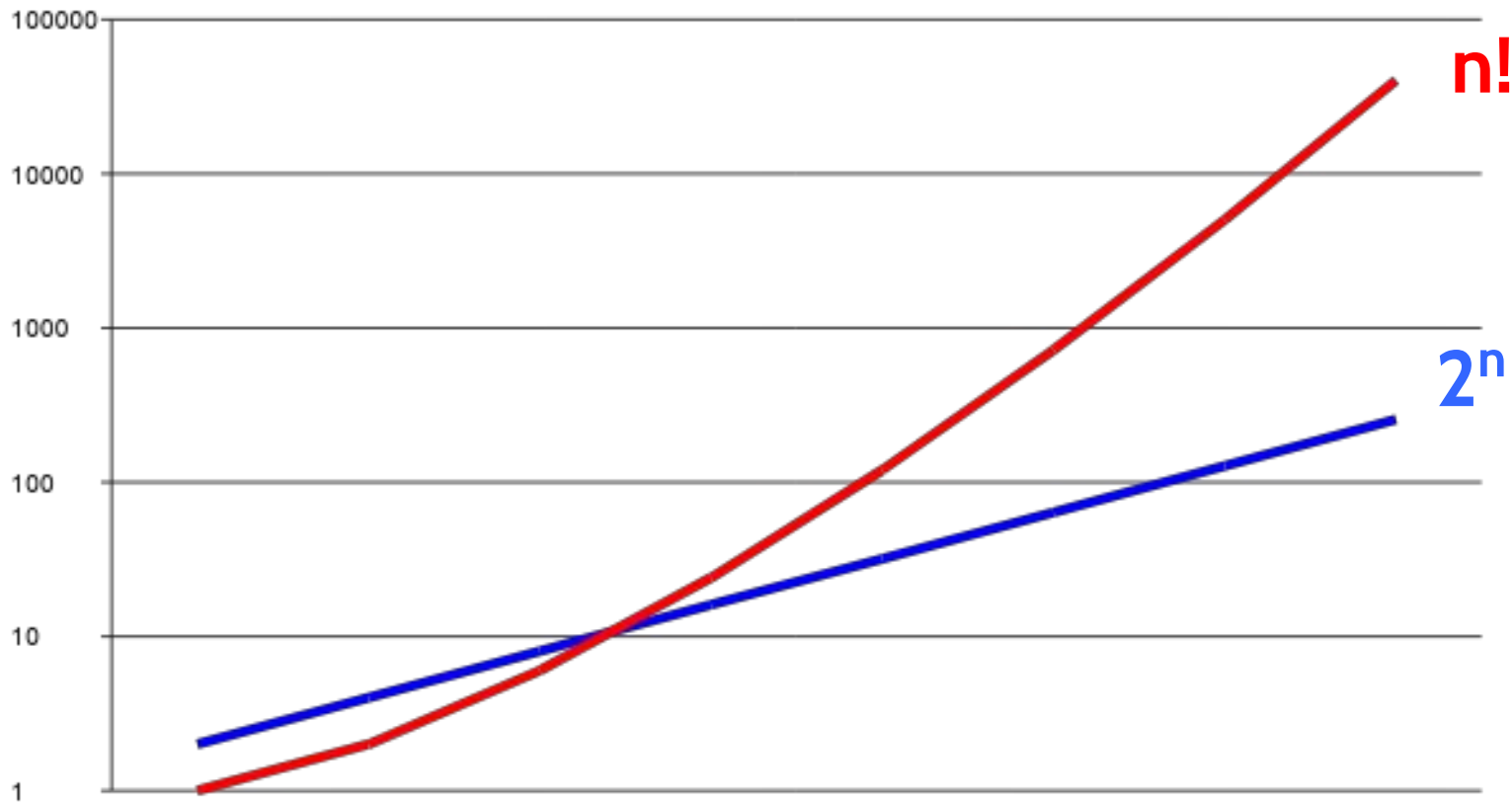
Análisis Teórico - Big O tratables

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 fuerza bruta	Adivinar una password
$n!$	Factorial	Búsqueda fuerza bruta	Problema del viajante



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.
- **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 - 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 - 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
```

- El mejor caso (si la lista no está vacía) es:
 - x es el primer elemento de data

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 es el último elemento o no está en la lista (tendremos que recorrer la lista completa)

Análisis Teórico - Mejor y peor caso

- ¿Peor y mejor caso del siguiente algoritmo?

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

Análisis Teórico - Mejor y peor caso

- **En algunos algoritmos**, todos los casos son computacionalmente igual de costosos.
- En este algoritmo, **no existe peor ni mejor caso**, porque en todos los casos siempre es necesario recorrer toda la lista (siempre que la lista no esté vacía).

```
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: list) -> int:  
    max=-999999  
    for c in data:  
        if c>max then  
            max=c  
    return max
```


Análisis Teórico - Ejercicio

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

- Sea n el tamaño de `data`
- La función temporal $T(n) = 1 + n*(2+1) + 1 = 3n+2$
- 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
```

- Sea n el tamaño de `data`.
- La función temporal $T(n) = n \cdot (1+1) + 1 = 2n + 1$
- 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.

Resumen

- El análisis de algoritmos nos permite estudiar la complejidad temporal de un algoritmo y comparar distintos algoritmos que resuelven el mismo problema, para proponer la mejor solución.
- El análisis empírico se basa en medir el tiempo de ejecución de un algoritmo para distintas entradas. Es un análisis costoso porque es necesario implementar los algoritmos y asegurar un entorno de pruebas en igual condiciones para su comparación.
- El análisis teórico se calcula sobre el pseudo-código de los algoritmos, contabilizando el número de instrucciones para diferentes tamaños de la entrada, y consiguiendo su función temporal.

Resumen

- El análisis teórico consiste en encontrar una función (cota superior) para la función temporal del algoritmo.
- El conjunto de cotas superiores (o funciones Big-O) es un conjunto pequeño:
 - $1 < \log n < n < n \log n < n^2 < n^3 < \dots < 2^n < n!$
- Este conjunto nos va a permitir clasificar y comparar las funciones temporales de nuestros algoritmos sin necesidad de implementar ni requerir un entorno de pruebas sw/hw.