uc3m | Universidad Carlos III de Madrid

OpenCourseWare
Grado Ingeniería Informática
Estructura de Datos y Algoritmos

Tema 3 Análisis de Algoritmos

Profesores EDA

Análisis de Algoritmos

- Un **algoritmo** es un conjunto de pasos (instrucciones) para resolver un problema.
  - Debe ser robusta y correcta!!!.
  - 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

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

Análisis de Algoritmos

- ¿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

- 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:**

```python
def sum_n(n: int) -> int:
    """returns the sum of the first n numbers"""

    if not isinstance(n, int) or n < 0:
        return None

    result = 0
    for i in range(n+1):
        result += i
    return result
```

Análisis Empírico

2) Incluye instrucciones para medir el tiempo:

```
import time


def sum_n(n: int) -> (int, float):
    start = time.time()  # returns the time in milliseconds
    if not isinstance(n, int) or n < 0:
        return None

    result = 0

    for i in range(n+1):
        result += i
    end = time.time()
    return result, end-start  # we also return the time
```

Análisis Empírico

3) Ejecuta el programa para tamaños diferentes

```python
if __name__ == '__main__':
    for i in range(1, 8):
        input_size = pow(10, i)
        result_sum, execution_time = sum_n(input_size)
        print("sumN({})={}, time={} ms".format(input_size, result_sum, execution_time))
```

```
sumN(10)=55, time=9.536743164062e-07 ms
sumN(100)=5050, time=3.814697265625e-06 ms
sumN(1000)=500500, time=4.291534423828125e-05 ms
sumN(10000)=50005000, time=0.0004360675811767578 ms
sumN(100000)=5000050000, time=0.004608695526123047 ms
sumN(1000000)=500000500000, time=0.04788565635681152 ms
sumN(10000000)=50000005000000, time=0.4881730079650879 ms
```

Análisis Empírico

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

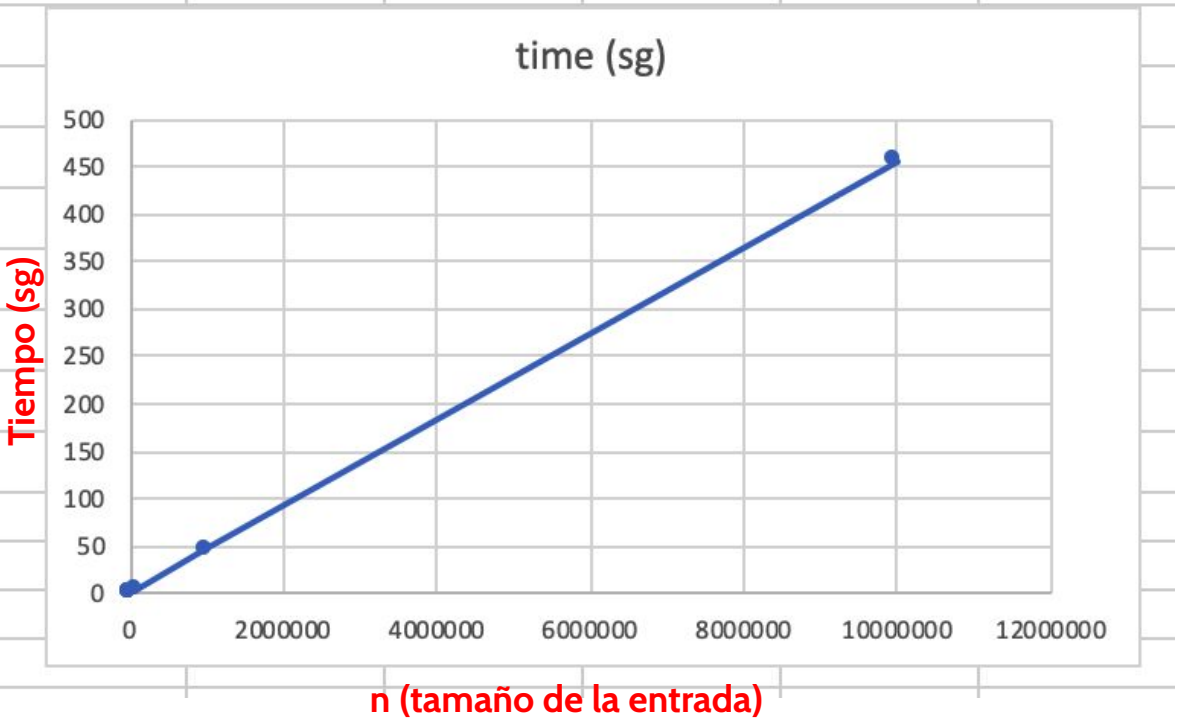
```
if __name__ == '__main__':
    for i in range(1, 8):
        input_size = pow(10, i)
        result_sum, execution_time = sum_n(input_size)
        print("sumN({})={}, time={} sg".format(input_size, result_sum, execution_time*1000)
```

```
sumN(10)=55, time=0.0009536743164062 5 sg
sumN(100)=5050, time=0.004053115844726562 5 sg
sumN(1000)=500500, time=0.04291534423828125 sg
sumN(10000)=50005000, time=0.4529530029296875 sg
sumN(100000)=5000050000, time=4.713773727416992 sg
sumN(1000000)=500000500000, time=49.21984672546387 sg
sumN(10000000)=50000005000000, time=456.09593391418457 sg
```

Análisis Empírico

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

n	time (sg)
10	0,000953674
100	0,00500679
1000	0,043869019
10000	0,468254089
100000	4,755973816
1000000	47,31917381
10000000	455,6391239



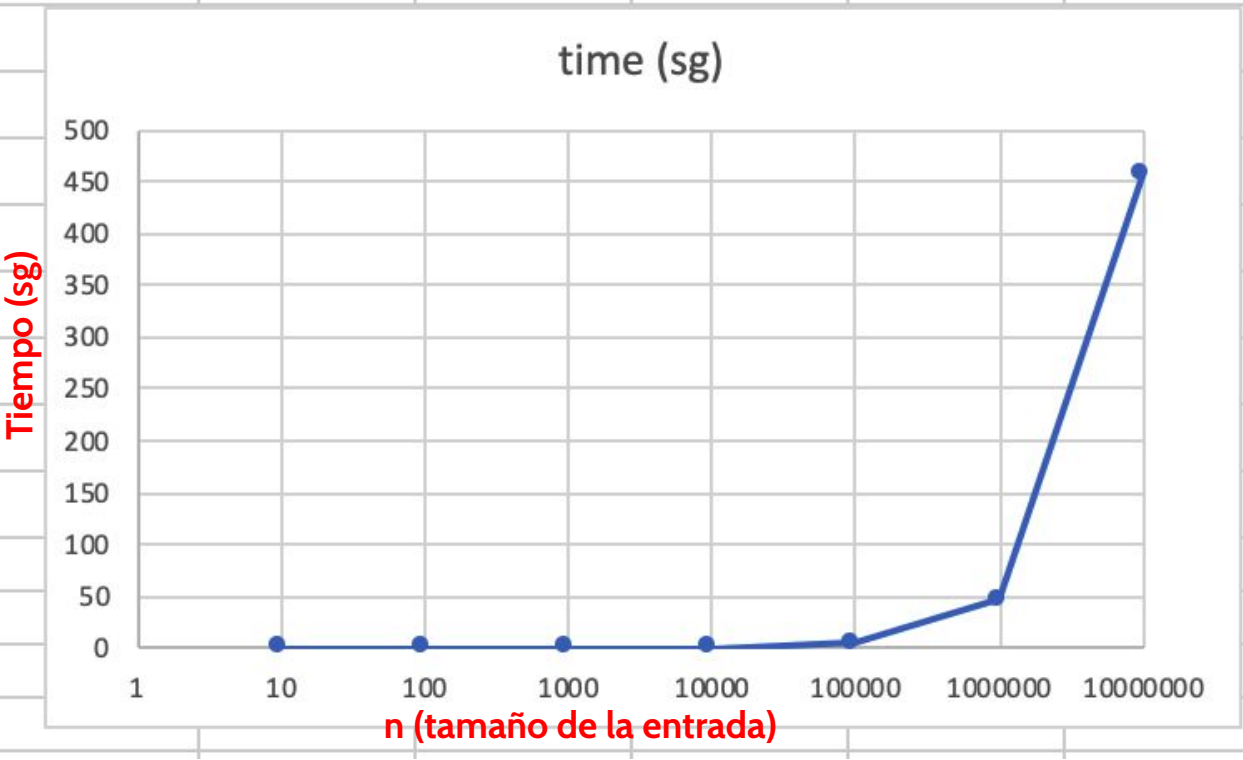
Análisis Empírico

- Cuando necesites mostrar rangos muy grandes de datos (como en el ejemplo anterior), transforma tu gráfico a la **escala logarítmica**.
- En tu gráfico, 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)

n	time (sg)
10	0,000953674
100	0,00500679
1000	0,043869019
10000	0,468254089
100000	4,755973816
1000000	47,31917381
10000000	455,6391239



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:**

```
def sum_gauss(n: int) -> (int, float):
    """returns the sum of the first n numbers, by
    applying Gauss sum"""
    start = time.time()  # returns the time in milliseconds
    if not isinstance(n, int) or n < 0:
        return None


    result = n*(n+1) / 2
    end = time.time()
    # we return the result and the execution time in seconds
    return result, (end-start)*1000
```

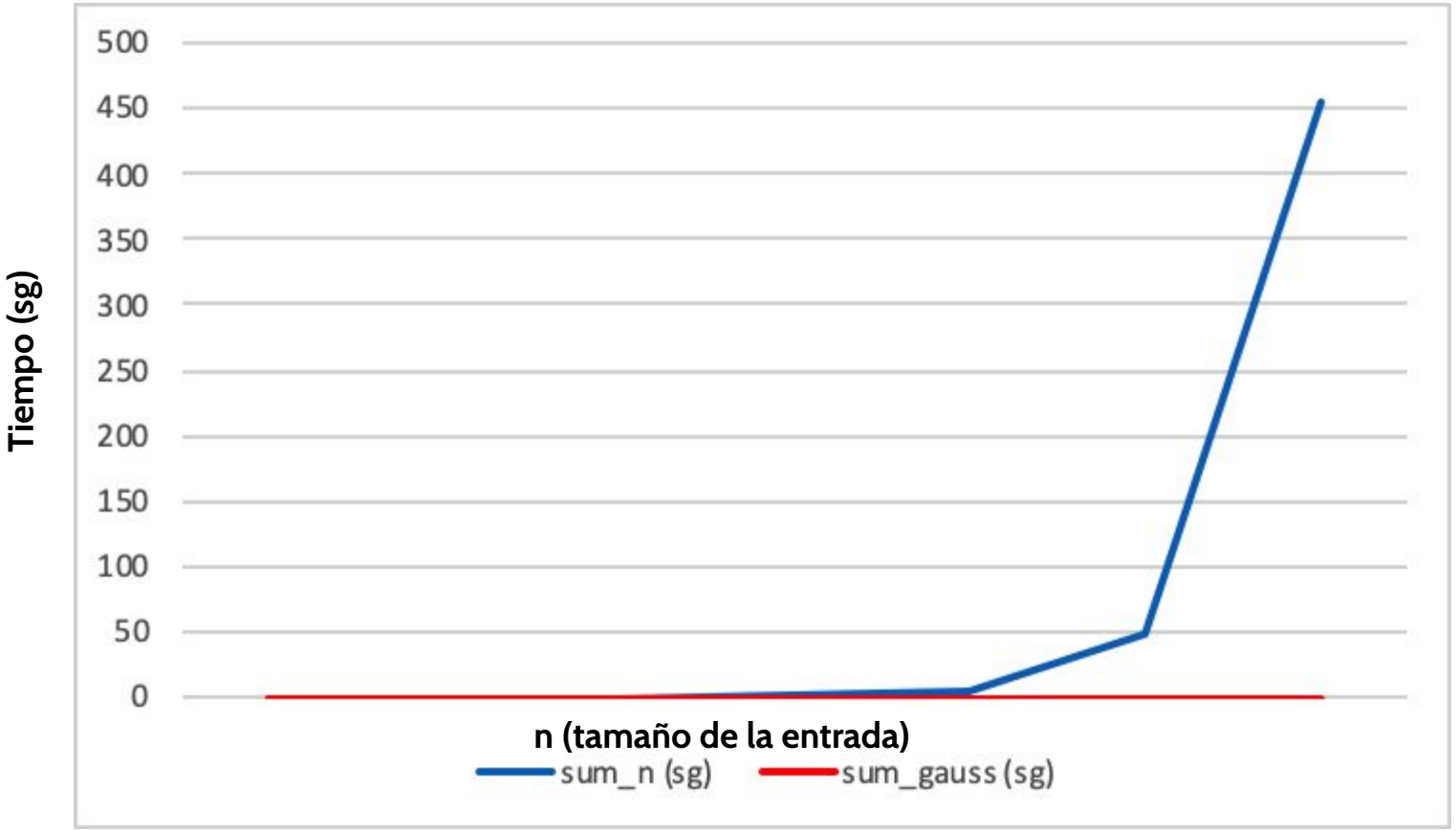
Análisis Empírico

```python
if __name__ == '__main__':
    for i in range(1, 8):
        input_size = pow(10, i)
        result_sum, execution_time = sum_gauss(input_size)
        print("sum_gauss({})={}, time={} sg".format(input_size, result_sum, execution_time))
```

```
sum_gauss(10)=55.0, time=7.152557373046875e-07 sg
sum_gauss(100)=5050.0, time=0.0 sg
sum_gauss(1000)=500500.0, time=0.0 sg
sum_gauss(10000)=50005000.0, time=0.0 sg
sum_gauss(100000)=5000050000.0, time=0.0 sg
sum_gauss(1000000)=500000500000.0, time=0.0 sg
sum_gauss(10000000)=50000005000000.0, time=0.0 sg
```

Si medimos el tiempo en segundos, normalmente implica 0 segundos para cualquier tamaño.

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

- El análisis empírico nos permite obtener gráficas para razonar sobre el tiempo de ejecución de un algoritmo.
- Estas gráficas nos van a permitir comparar fácilmente el tiempo de ejecución de distintos algoritmos para un mismo problema.
- Sin embargo, algunas **desventajas**:
  - Es necesario implementar los algoritmos.
  - Mismo entorno para comparar los algoritmos.
  - 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 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**
x = 2	
x = y + 3	
return a[0] + 1	
return node is None or node.elem >5	
print(“Hola”)	

Análisis Teórico - operaciones primitivas

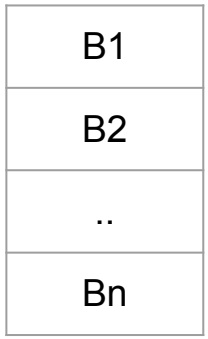
Operación	**# operaciones primitivas**
x = 2	1
x = y + 3	2 (1 suma + 1 asignación)
return a[0] + 1	3 (1 acceso + 1 suma + 1 return)
return node is None or node.elem >5	5 (1 node is None + 1 obtener node.elem + 1 node.elem >5 + 1 or + 1 return)
print("Hola")	1

Análisis Teórico - operaciones primitivas

- Por simplificar el cálculo de la función T(n), siempre podríamos contabilizar este **tipo de operaciones primitivas (que se componen de otras primitivas) como una única operación** (veremos más adelante por qué podemos hacer esta simplificación).
- La **función temporal de una operación primitiva siempre tiene un valor constante, que no depende de n, el tamaño de la entrada.**

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)+...+T(Bn)

Análisis Teórico

Algorithm *swap*(a,b)	**# operations**
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_{loop} (n) = T(B) * 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(n+1):
    result = result + i
```

- La función range(n+1) devuelve los siguientes valores: 0,1,2, ..., n. Por tanto, el número de iteraciones será n+1.

- La función T(n) del bloque interno (`result=result+i`) es T(n) = 2.

Análisis Teórico - bucle

- Por tanto, la función **T(n) = (n+1) * 2**.
- En este caso, la función temporal **sí depende del valor de n.**

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

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) = 2
```

- Su función temporal T(n) es:

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

- Es decir, 2 del 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, ... B_k) se ejecutará.

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

$$T_{if\text{-}else}(n) = \max(T_{B1}(n), T_{B2}(n), ..., T_{Bk}(n))$$

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 B_3

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
```

- Por tanto, T(n) = 1 + (evaluar la primera condición)
  1 + (evaluar la segunda condición)
  1 + (evaluar la tercera condición)
  n (función temporal 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
```

- Por tanto, T(n) = 3 + n
- Más tarde veremos que en realidad T(n) = 3 + n ó T(n) = n (si no hubiéramos tenido en cuenta las condiciones), implican el mismo coste.

Análisis Teórico - operaciones condicionales

- En el ejemplo anterior, estamos suponiendo que las funciones temporales de las condiciones tienen un valor constante 1.
- Para ser más rigurosos, deberíamos agregar las funciones temporales de las condiciones.
- Entonces, la función temporal final será:

$$T_{if\text{-}else}(n) = \sum T_{condition\_i}(n) + \max(T_{B1}(n), T_{B2}(n), \ldots, T_{Bk}(n))$$

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:
    """returns the sum of the first n numbers"""
    result = 0                  # 1
    for j in range(n+1):        # loop will be executed n+1 times
        result += j             # 2

    return result               # 1
```

Análisis Teórico

```
def sum_n(n: int) -> int:
    """returns the sum of the first n numbers"""
    result = 0                  # 1
    for j in range(n+1):        # loop will be executed n+1 times
        result += j             # 2

    return result               # 1
```

- Por tanto, T(n) = 2(n+1) + 2 = 2n + 4
- Depende del valor de n.

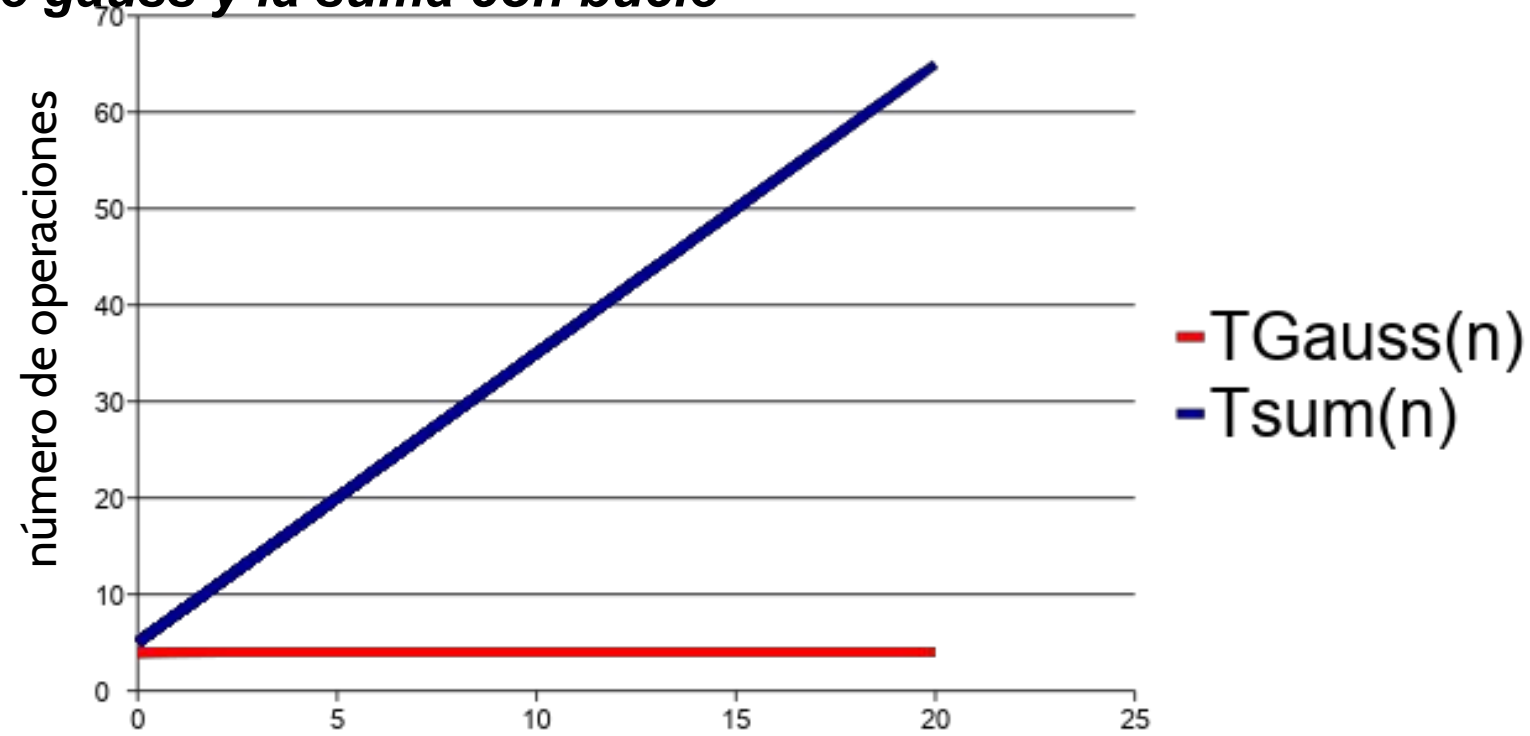
Análisis Teórico

```
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



$T_{sum}(n)=2n+4$

$T_{Gauss}(n)=4$

Análisis Teórico

- Si suponemos que cada operación tarda un tiempo constante, por ejemplo, c=1 ns, entonces, podemos estimar que la solución de gauss siempre tardará 4 ns, para cualquier tamaño n, mientras que la solución basada en el bucle tardará 2n+4 ns.
- 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

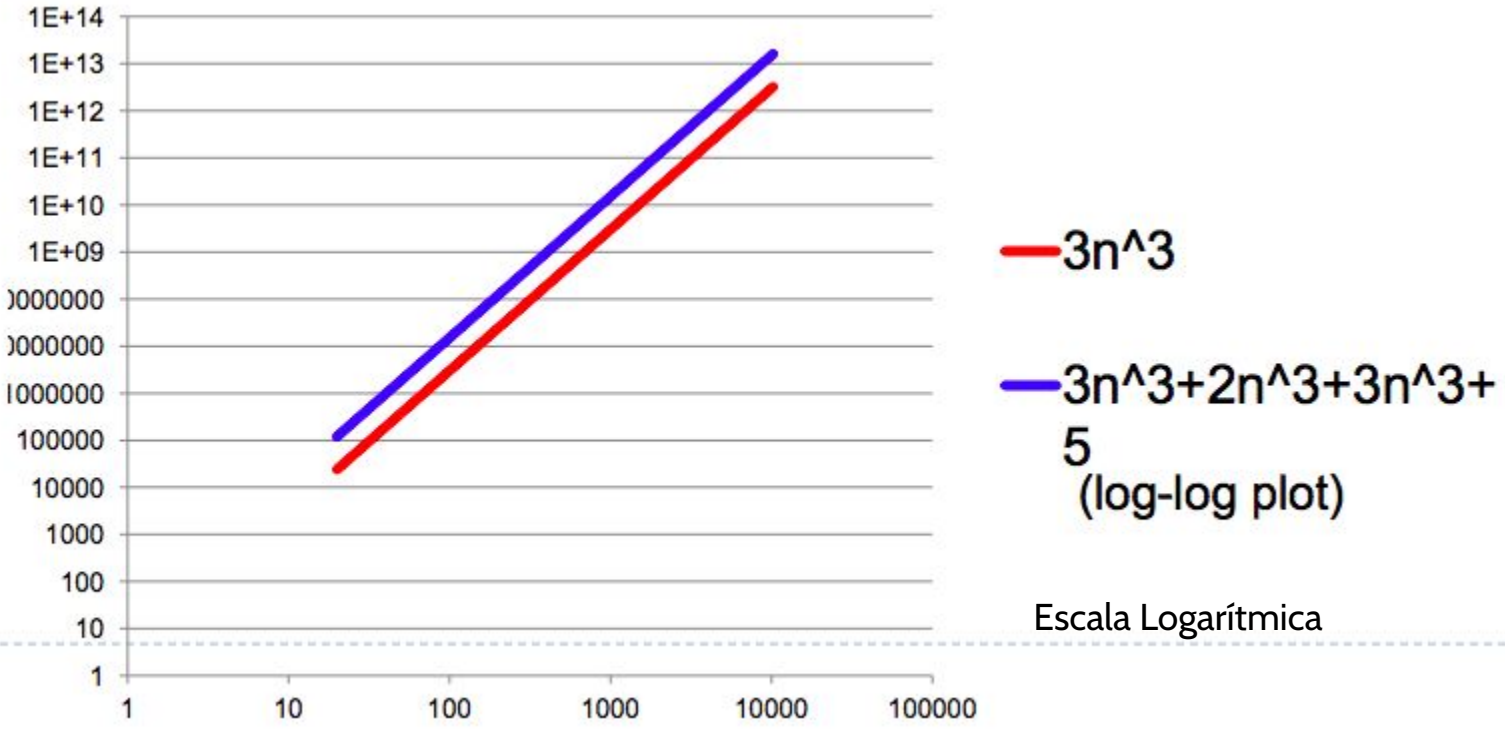
- Supón que tienes dos algoritmos, A y B, con las siguientes funciones de tiempo:
  - $T_A(n)=3n^3$
  - $T_B(n)=3n^3+2n^2+3n+5$

¿Qué algoritmo es más eficiente?

Análisis Teórico

$T_A(n) = 3n^3$

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

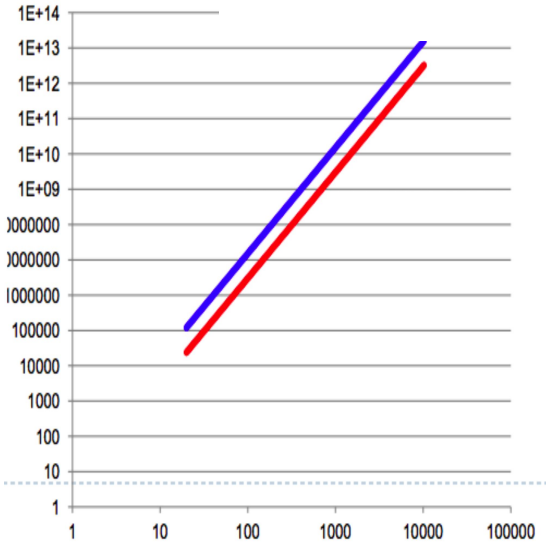


Escala Logarítmica

Análisis Teórico

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

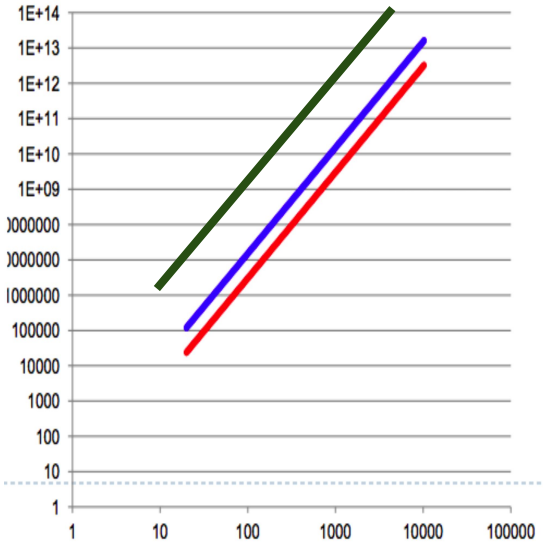
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1$$



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

Análisis Teórico

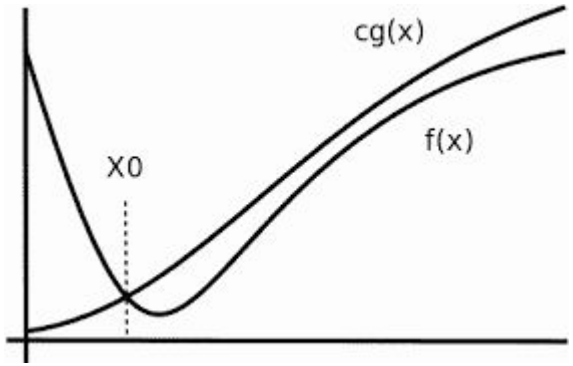
- Tiempo de ejecución depende:
  - De la máquina en la que se ejecuta el programa.
  - Del compilador utilizado para generar el programa.
- Para facilitar la comparación de las funciones temporales, vamos a aproximar cada función temporal a una cota superior (análisis asintótico)



Análisis Teórico - cota superior asintótica

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

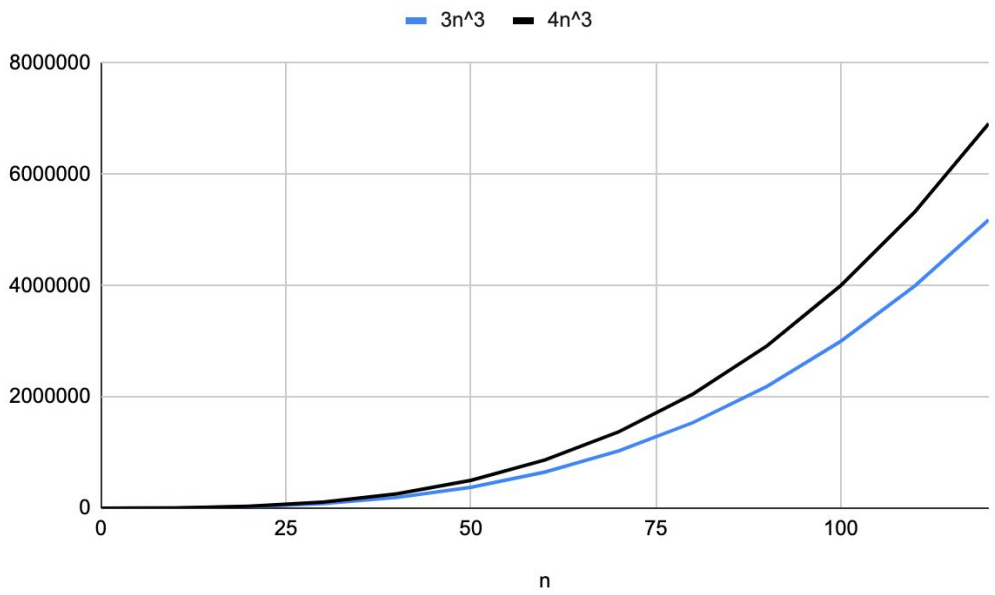
$$f(n) <= cg(n), \text{ para todo } n >= n_0$$



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

Análisis Teórico - cota superior asintótica

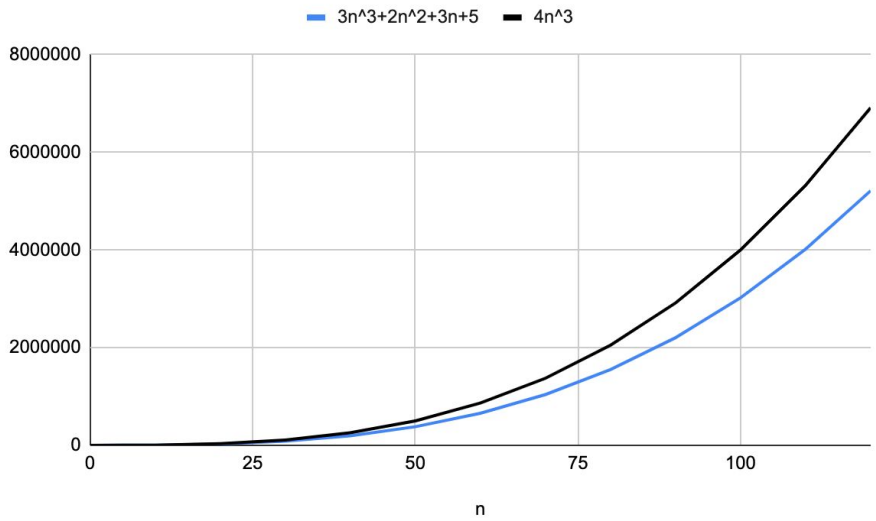
$T_A(n)= 3n^3$ es de orden superior n^3 porque **existen $n_0=0$, $c=4$**, tales que $T_A(n) <= cn^3$, para todo $n>=n_0$



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 $T_B(n) <= cn^3$, para todo $n>=n_0$



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.

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

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.**

- $T_A(n)=3n^3$ -> $3n^3$ -> n^3
- $T_B(n)=3n^3+2n^2+3n+5$ -> $3n^3$ -> 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(?)
½(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)
½(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

Big O funciones

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

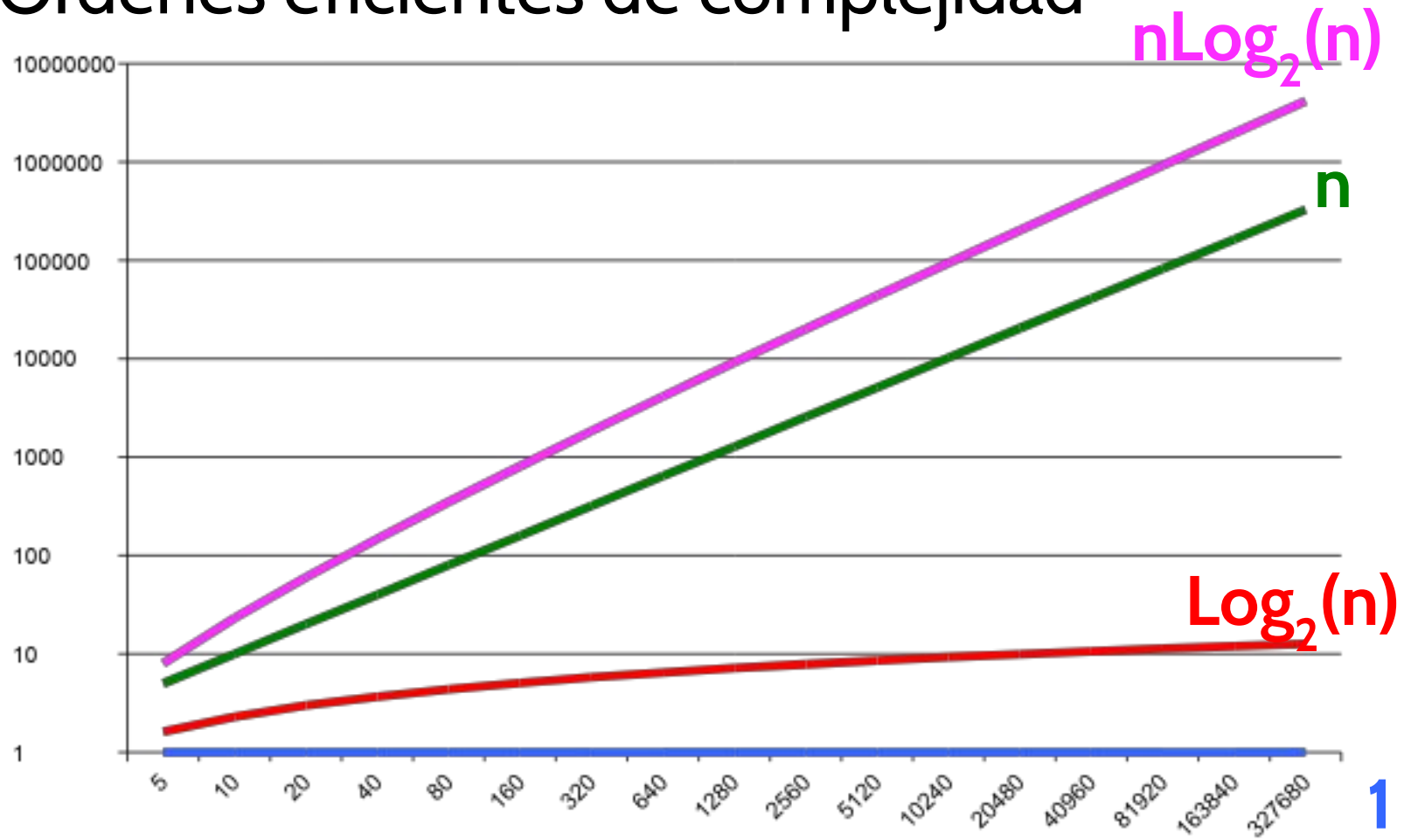
Análisis Teórico - cota superior asintótica

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

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

Análisis Teórico - Big O eficientes

Órdenes eficientes de complejidad

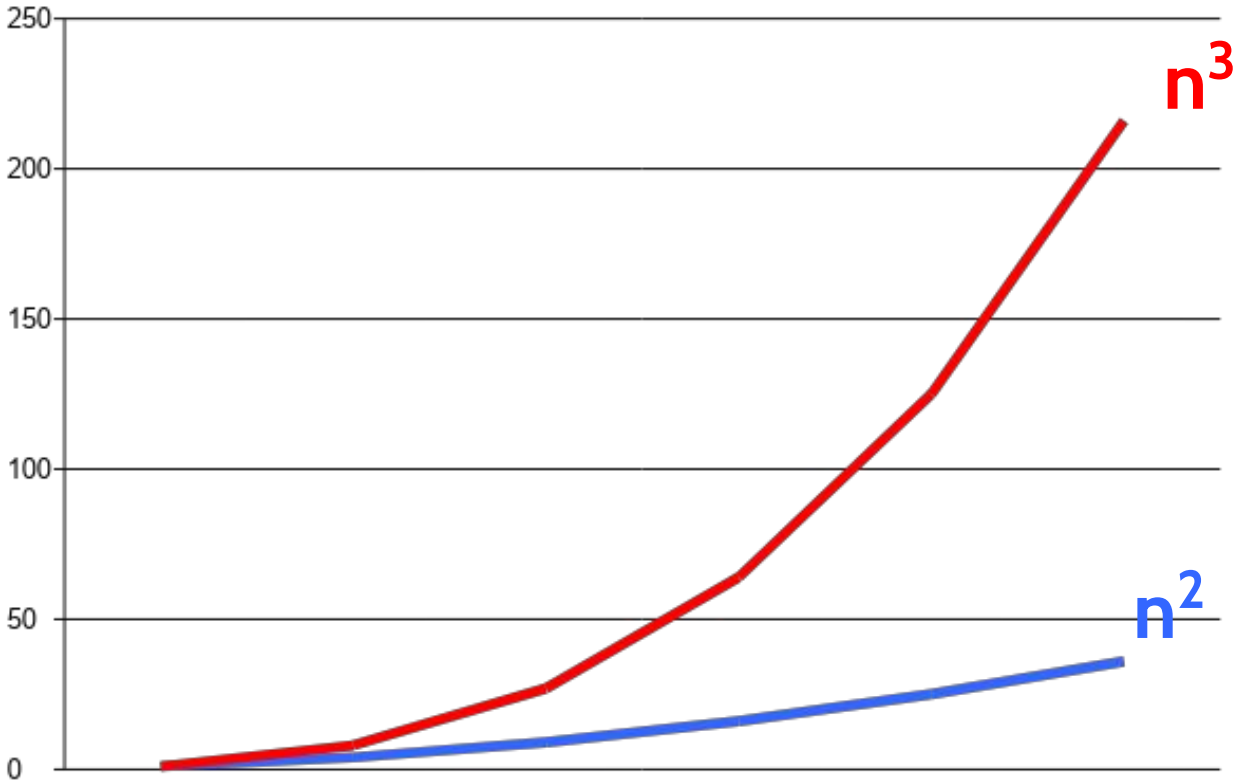


Análisis Teórico - Big O eficientes

Orden	Nombre	Descripción	Ejemplo
1	Constante	Independiente del tamaño	Borrar el primer elemento de una cola
$Log_2(n)$	Logarítmico	Dividir por la mitad	Búsqueda binaria
n	Lineal	Bucle	Suma de los elementos de una lista
$nLog_2(n)$	Lineal-logarítmico	Divide y Vencerás	Mergesort, quicksort

Análisis Teórico - Big O tratables

Órdenes tratables de complejidad

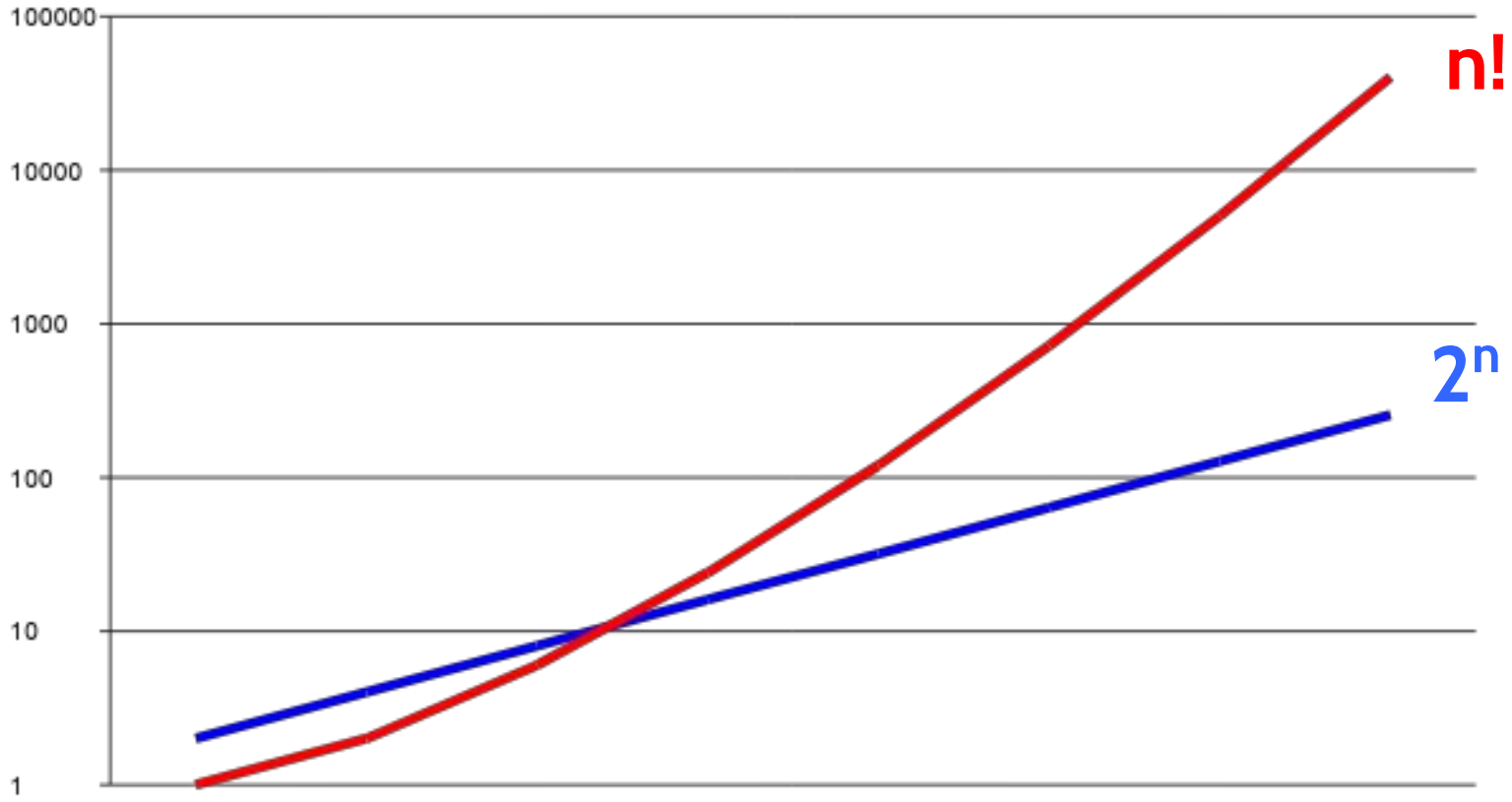


Análisis Teórico - Big O tratables

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	Enumerar todas las posibles particiones de un conjunto

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
```

- 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: 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
```

- 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.

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:
  - $1 < \log n < n < n\log n < n^2 < n^3 < \ldots < 2^n < n!$
- 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.