

# Programación Competitiva 2025

## Complejidad

*Un Algoritmo es un procedimiento paso a paso para resolver un problema en una cantidad de tiempo finita.*

Un problema se puede resolver mediante muchos algoritmos.

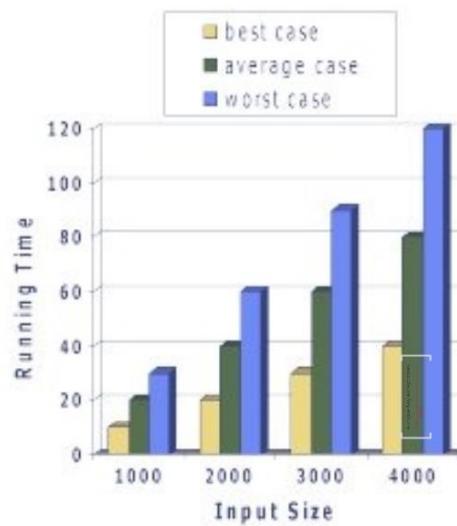
Pero distintas soluciones pueden tener diferente eficiencia

Una vez que tenemos un algoritmo solución y sabemos que es correcto, hay que determinar su

*Complejidad Computacional: cantidad de recursos que necesita para su ejecución*

- Recursos Temporales (CPU)
- Recursos Espaciales
  - Memoria
  - Disco

El tiempo de ejecución de un algoritmo generalmente crece con el tamaño de la entrada:



- Un algoritmo es o no es eficaz.
- La eficiencia es comparativa.
  - Un algoritmo es más o menos eficiente que otro .... según el/los aspectos que se consideren

## Dos soluciones eficaces para Fibonacci:

```
long fibo(int n){  
    if (n == 1 or n == 2)  
        return n;  
    else  
        return fibo(n-1)+fibo(n-2);  
}
```

```
long fibo2(int a, int b, int c){  
    if (c == 0)  
        return a;  
    else  
        return fibo2(b, a+b, c-1);  
}
```

*Cuál es la mejor si hablamos de eficiencia???*

## **COMPLEJIDAD:**

- Qué medimos?
  - Tiempo de ejecución
- De qué depende?
  - Tamaño de la entrada
  - Otros factores:
    - Velocidad de la máquina
    - Calidad del programa
    - Calidad del compilador



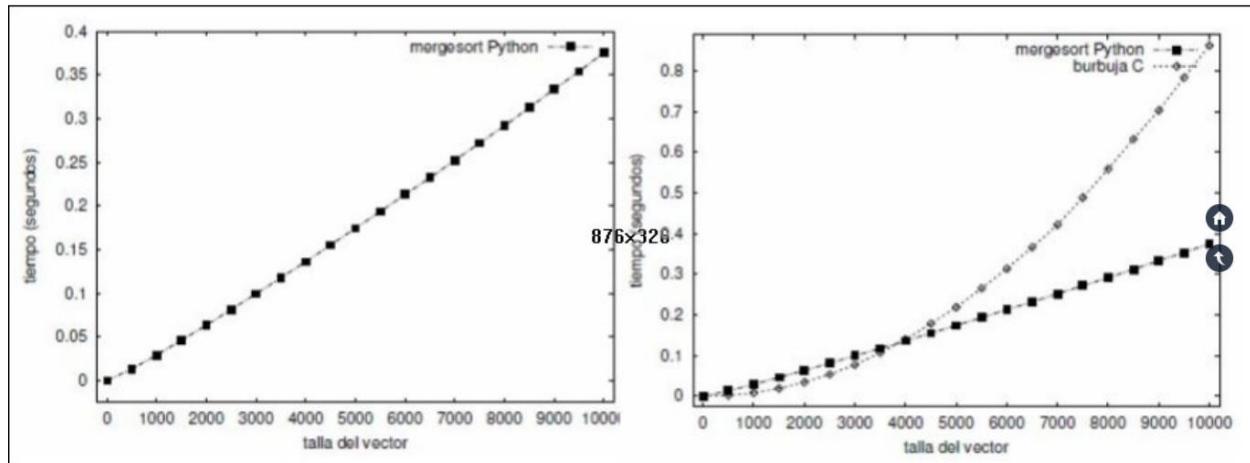
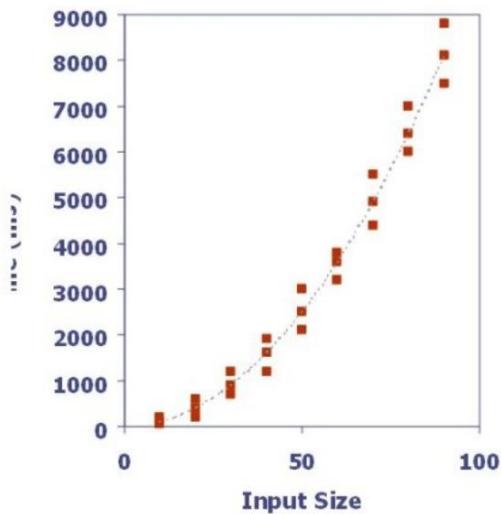
## **COMPLEJIDAD:**

- Cómo la medimos?
  - A priori: estimándola matemáticamente
  - A posteriori: experimentalmente



## Experimentalmente:

- Escribir un programa implementando el algoritmo.
- Correr el programa con entradas de diferentes tamaños y composiciones
- Tomar tiempos mediante una función tipo `clock()`
- Graficar los resultados.



## *Limitaciones de los experimentos:*

- Es necesario implementar el algoritmo, lo cual puede ser dificultoso y costoso...
- Los resultados pueden no ser indicativos de los tiempos de corrida de otras entradas no incluídas en el experimento...
- Para comparar 2 algoritmos debe utilizarse el mismo software y el mismo hardware...



#### *Análisis Teórico (a Priori):*

- Utiliza una descripción de alto nivel del algoritmo, en lugar de una implementación.
- Caracteriza los tiempos de ejecución como una función del tamaño de la entrada:  $T(n)$
- Toma en cuenta todas las posibles entradas y se emplea el peor caso (es más fácil de calcular que el caso promedio).
- Nos permite evaluar la velocidad de un algoritmo independientemente del entorno de hardware y software (Interesa la velocidad de crecimiento del tiempo de ejecución en función del tamaño de la entrada).
- Comportamiento asintótico (para un tamaño de entrada suficientemente grande)



#### *Estimación Teórica - Cómo medimos?*

- Para la descripción del algoritmo se emplea un seudocódigo.
- Tiene menos nivel de detalle que un programa.
- Los seudocódigos indican:
  - Control de flujo
  - Declaraciones de funciones
  - Llamadas a funciones
  - Retorno de valores
  - Expresiones





$T(n)$ = El tiempo de ejecución, lo expresamos como una función de  $n$ , el tamaño o talla del problema.



***n*** puede ser:

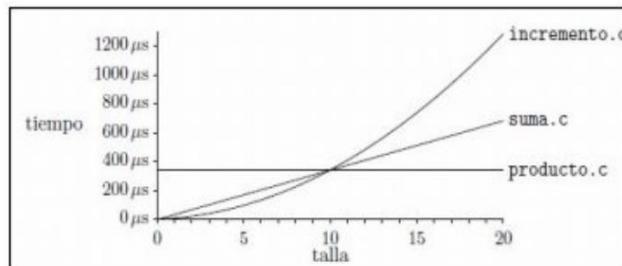
- la cantidad de números a procesar
- la cantidad de dígitos del número
- la cantidad de elementos de una matriz
- la longitud de un string
- la cantidad de elementos de un conjunto
- .....



## Estimación Teórica - Cómo medimos?

- Problema: cálculo de  $n^2$
- Instancia del problema:  $n=10$
- Tamaño o talla del problema: valor máximo que puede tomar  $n$

Operación	Producto	Suma	Incremento	Asignación	Comparación
Tiempo	$342\mu s$	$31\mu s$	$1\mu s$	$1\mu s$	$1\mu s$



## Cálculo de $n^2$ : Conclusiones

- Un método que tarda un **tiempo constante** siempre termina siendo mejor que uno cuyo tiempo depende **linealmente** de la entrada.
- El primer método es **asintóticamente** más eficiente que el segundo.
- Conclusión:
  - Nos interesa el comportamiento de los programas para tamaños grandes de entrada.
  - Podemos simplificar nuestro análisis.



## Estimación Teórica - Cómo medimos?

El concepto de paso:

Como no nos interesa lo que tarda cada operación elemental, es conveniente introducir el concepto de paso:

*"Un paso es un segmento constante de código cuyo tiempo de proceso no depende de la talla del problema considerado y está acotado por alguna constante"*

Ejemplos:

- operaciones aritméticas, lógicas
- acceso a una variable escalar
- acceso a un elemento de un vector
- asignaciones, etc

## Estimación Teórica - Cómo medimos?

Costo computacional temporal:

- número de pasos expresado en función de la talla del problema.
- Es una medida independiente del lenguaje.

suma.c	
1 <b>#include</b> <stdio.h>	1 paso
2 <b>int</b> main( <b>void</b> )	<u>2n + 2 pasos</u>
3   {	
4 <b>int</b> m, n, i;	
5 <b>scanf</b> ("%d", &n);	
6     m = 0;	
7 <b>for</b> (i=0; i<n; i++)	<u>2 pasos,      n veces</u>
8       m = m + n;	
9 <b>printf</b> ("%d\n", m);	
10	
11 }	

O sea, un total de  $4n + 3$  pasos.

## Estimación Teórica - Cómo medimos?

- Costo computacional temporal: número de pasos expresado en función de la talla del problema.

	producto.c	suma.c	incremento.c
pasos	1	$4n + 3$	$2n^2 + 4n + 4$

- Cualquier secuencia de pasos cuya longitud no depende de la talla del problema cuenta como una cantidad constante de pasos.

	producto.c	suma.c	incremento.c
pasos	$c_0$	$c_1 \cdot n + c_2$	$c_3 \cdot n^2 + c_4 \cdot n + c_5$

## Estimación Teórica - Independencia del LP

Programa	Coste
sumatorio1.c	$c_0 \cdot n + c_1$
sumatorio2.c	$c_2 \cdot n + c_3$
sumatorio.py	$c_4 \cdot n + c_5$

???

## Principio de Invarianza :

“ Dos implementaciones diferentes del mismo algoritmo difieren en eficiencia en no más que una constante multiplicativa ”

## Ejemplo: Análisis Teórico sobre Seudocódigo

Algoritmo arrayMax(A,n)

Input: array A de n enteros

Output: elemento máximo de A

current=A[0]

for i=1 to n-1 do

    if current < A[i]

        then current=A[i]

    incrementar i

return current

Nº Operaciones

2

n-1

2(n-1)

2(n-1)

2(n-1)

1

Total: 7n-4

Caso Óptimo: 5n-2



### Estimación del Tiempo de Corrida

- El algoritmo arrayMax ejecuta  $7n-4$  operaciones primitivas en el peor caso.
- Si definimos:
  - a = Tiempo de la operación primitiva más lenta
  - b = Tiempo de la operación primitiva más rápida
- Sea  $T(n)$  el tiempo del peor caso de arrayMax, luego:  $b(7n-4) \leq T(n) \leq a(7n-4)$
- El tiempo de corrida  $T(n)$  está limitado por 2 funciones lineales.
- Cambiando el HW y SW, se afectará  $T(n)$  por un factor constante pero no cambia la tasa de crecimiento.
- La tasa de crecimiento lineal del tiempo de ejecución  $T(n)$  es una propiedad intrínseca del algoritmo arrayMax.

ines

entre  
4(n-1)  
y 6(n-1)

-2

### *Funciones típicas de tiempos de ejecución:*

- Constante: 1
- Logarítmica:  $\log n$
- Lineal:  $n$
- N-Log-N:  $n \log n$
- Cuadrática:  $n^2$
- Cúbica:  $n^3$
- Exponencial:  $2^n$

### *Ejemplo: bajar un fichero de Internet*

Retraso inicial (por conexión) : 1,2 seg

Descarga: 120 K/seg

Tamaño del fichero:  $nK$

Tiempo de descarga :  $T(n) = n/120 + 1,2$

- función lineal, proporcional al tamaño del archivo.

# Programación Competitiva 2025

## Complejidad - 2da. parte

### *COMPLEJIDAD - Otros ejemplos:*

- Dados N puntos del plano,  
encontrar el par de puntos más  
cercaos entre ellos.
  - Orden ?
- Dados N puntos del plano,  
determinar si existen 3 colineales
  - Orden ?



CC

• .  
€

•  
• t  
• .

### COMPLEJIDAD - Otros ejemplos:

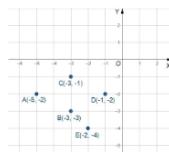
- ... el par de puntos más cercanos entre ellos:

- $N(N-1)/2$  pares  
 $T(N) \approx N^2$

- ... si existen 3 colineales:

- Ternas diferentes:  $N(N-1)(N-2)/6$   
(combinaciones de N tomadas de a 3)

$$T(N) \approx N^3$$

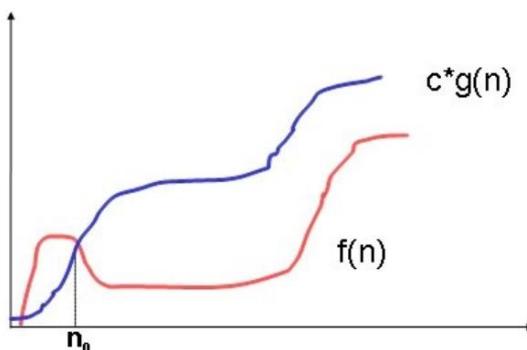


... hay soluciones más eficientes?



### Notación O

- En general interesa conocer el comportamiento del tiempo cuando la cantidad de datos (N) crece.



Dadas 2 funciones  $f(n)$  y  $g(n)$  se dice que:

$f(n)$  es  $O(g(n))$  siii existen constantes positivas  $c$  y  $n_0$  tal que:  
 $f(n) \leq c g(n)$  , para  $n \geq n_0$

## Notación O - Ejemplos

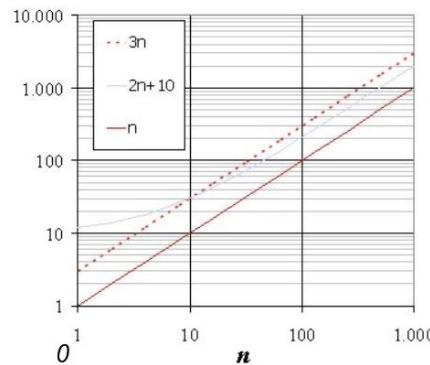
- $2n + 10$  es  $O(n)$  ?

$$2n + 10 \leq cn$$

$$10 \leq cn - 2n$$

$$10 \leq n(c - 2)$$

$$\frac{10}{c - 2} \leq n$$



Se verifica para  $c=3$  y  $n_0 = 10$

cualquier algoritmo que tenga  $T(N) = aN+b$   
es de complejidad lineal ("es  $O(N)$ ")

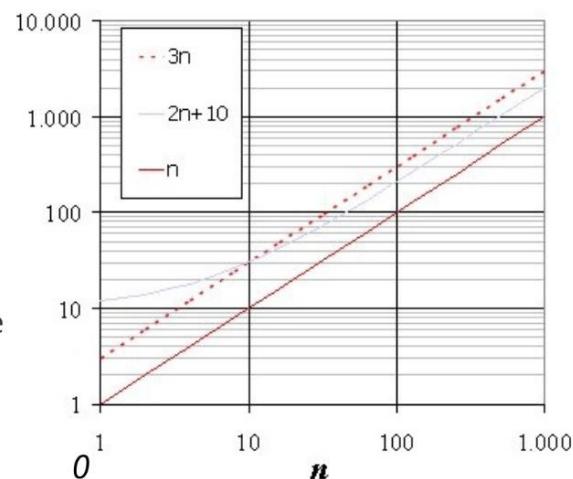
## Notación O - Ejemplos

- $n^2$  es  $O(n)$  ?

$$n^2 \leq cn$$

$$n \leq c$$

imposible de cumplir porque  
c debe ser constante



## $7n^2$

$7n^2$  es  $O(n)$

Sea  $c > 0$  y  $n_0 \geq 1$  tal que  $7n^2 \leq c \cdot n$  para  $n \geq n_0$

Esto es cierto para  $c = 7$  y  $n_0 = 1$

## $3n^3 + 20n^2 + 5$

$3n^3 + 20n^2 + 5$  es  $O(n^3)$

Sea  $c > 0$  y  $n_0 \geq 1$  tal que  $3n^3 + 20n^2 + 5 \leq c \cdot n^3$  para  $n \geq n_0$

Esto es cierto para  $c = 4$  y  $n_0 = 21$

## $3 \log_2 n + 5$

$3 \log_2 n + 5$  es  $O(\log n)$

Sea  $c > 0$  y  $n_0 \geq 1$  tal que  $3 \log_2 n + 5 \leq c \cdot \log_2 n$  para  $n \geq n_0$

Esto es cierto para  $c = 8$  y  $n_0 = 2$



1.000



### *La notación O :*

- Se utiliza para representar el índice de crecimiento.
- Permite establecer un orden relativo entre las funciones de tiempo, comparando el término dominante,
  - Para valores pequeños es difícil comparar ej:  $N+2500$  contra  $N^2$  cuando  $N < 50$ 
    - A partir de un punto la función lineal será siempre mejor que la cuadrática.
    - $N^2$  es demasiado lento para miles de elementos (ej: ordenamiento por inserción para 100.000 elementos)

*NO PERDER TIEMPO  
INTENTANDO OPTIMIZAR EL  
CÓDIGO sino TRATAR DE  
MEJORAR EL ORDEN DEL  
ALGORITMO!!!*

---

*Justificación de Notación Asintótica:*

- Para un  $N$  suficientemente grande, el valor de la función está completamente determinado por su término dominante.
  - El valor del coeficiente del término dominante no se conserva al cambiar de máquina (ej: la calidad del compilador).
  - Los valores pequeños de  $N$  generalmente no son de interés.
-

### *Factores Constantes*

- El índice de crecimiento de una función no es afectado por:
  - factores constantes
  - términos de menor orden
- Así:
  - $10^2 n + 10^5$  es una función lineal
  - $10^5 n^2 + 10^7 n$  es una función cuadrática

### *Peso del término dominante*

- $10n^3 + n^2 + 40n + 80$

para  $n=1000$ ,  $f(n)=10.001.040.080$   
( $n^3$  tiene un error de 0,01%)

## *La notación O*

- $f(n)$  es  $O(g(n))$  significa que la tasa de crecimiento de  $f(n)$  no es mayor que la tasa de crecimiento de  $g(n)$
- Se puede usar la notación O para clasificar las funciones de acuerdo a su tasa de crecimiento.

*f(n) es O(g(n)) ? g(n) es O(f(n))?*

• $g(n)$ con tasa mayor	Si	No
• $f(n)$ con tasa mayor	No	Si
• Misma tasa	Si	Si

## *Reglas de la Notación O*

- Si  $f(n)$  es polinomial de grado  $d$ , luego  $f(n)$  es  $O(n^d)$
- Usar la clase de funciones más pequeña posible
  - Se dice que  $2n$  es  $O(n)$  en lugar de  $O(2n)$
- Usar la expresión más simple de la clase.
  - Se dice que  $3n+5$  es  $O(n)$  en lugar de  $O(3n)$
- El análisis asintótico de un algoritmo determina el tiempo de ejecución en notación O.
- Para el análisis asintótico:
  - Se halla el número de operaciones primitivas para el peor caso.
  - Se expresa esta función con notación O.
  - Se desagrega el algoritmo para el conteo.

## *Clasificación de las Cotas*

Sublineales		Constantes	$O(1)$
		Logarítmicas	$O(\log n)$
			$O(\sqrt{n})$
Lineales			$O(n)$
Superlineales			$O(n \log n)$
		Polinómicas	$O(n^2)$
		Cúbicas	$O(n^3)$
	Exponenciales		$O(2^n)$
			$O(n^n)$

## *Crecimiento de Funciones*

Log n	$\sqrt[n]{n}$	n	$n \log n$	$n^2$	$n^3$	$2^n$
1	1,4	2	2	4	8	4
2	2,0	4	8	16	64	16
3	2,8	8	24	64	512	256
4	4,0	16	64	256	4.096	65.536
5	5,7	32	160	1.024	32.768	4.294.967.296
6	8,0	64	384	4.096	262.144	$1,8 \cdot 10^{19}$
7	11,0	128	896	16.536	2.097.152	$3,4 \cdot 10^{38}$

## Tamaño máximo de un problema

Tiempo de ejecución	1 segundo	1 minuto	1 hora
$400 n$	2.500	150.000	9.000.000
$20 n [\log n]$	4.966	166.666	7.826.087
$2 n^2$	707	5.477	42.426
$n^4$	31	88	244
$2^n$	19	25	31

- Suponemos que cada operación puede realizarse en  $1\mu s$  (microsegundo=millonésima parte de un segundo)
- Limitamos tiempo máximo de ejecución
- Vemos máximo tamaño de entrada que admite en cada tiempo

## Tamaño máximo de un problema

- Aumento en el tamaño máximo de un problema que se puede resolver en determinado tiempo, usando una computadora 256 veces más rápida, para distintos órdenes del algoritmo.
- Cada entrada se da en función de  $m$ , el tamaño máximo anterior del problema.

Tiempo de ejecución	Nuevo tamaño máximo del problema
$400 n$	$256 m$
$20 n [\log n]$	Aprox. $256 ((\log m)/(7+\log m))m$
$2 n^2$	$16 m$
$n^4$	$4 m$
$2^n$	$m + 8$

## Otras Notaciones utilizadas

Expresión matemática	Índice de crecimiento relativo
$t(N) = O(f(N))$	El crecimiento de $t(N)$ es $\leq$ que el de $f(N)$ • Se dice que $F(N)$ es una cota superior
$t(N) = \Omega(f(N))$ (omega)	El crecimiento de $t(N)$ es $\geq$ que el de $f(N)$ • Se dice que $F(N)$ es una cota inferior
$t(N) = \Theta(f(N))$ (zeta)	El crecimiento de $t(N)$ está acotado superior e inferiormente por funciones proporcionales a $f(N)$
$t(N) = o(f(N))$	El crecimiento de $t(N)$ es $<$ que el de $f(N)$

## Otras Notaciones utilizadas

Expresión matemática	Definición
$f(n) = O(g(n))$	Existe $c$ positiva y $n_0 \geq 1$ : $f(n) \leq c g(n)$ para $n \geq n_0$ $f(n)$ es $O(g(n))$
$f(n) = \Omega(g(n))$	Existe $c$ positiva y $n_0 \geq 1$ : $f(n) \geq c g(n)$ para $n \geq n_0$ $g(n)$ es $\Omega(f(n))$
$f(n) = \Theta(g(n))$	Existe $c'$ y $c''$ positivas y $n_0 \geq 1$ : $c' g(n) \leq f(n) \leq c'' g(n)$ para $n \geq n_0$ $f(n)$ es $O(g(n))$ y $f(n)$ es $\Theta(g(n))$
$f(n) = o(g(n))$	Para cualquier $c$ positiva, existe $n_0 \geq 1$ : $f(n) < c g(n)$ para $n \geq n_0$

Relación de inclusión entre los órdenes:

$$\begin{aligned}O(1) &\subset O(\log n) \subset O(\sqrt{n}) \subset O(n) \subset O(n \log n) \subset \\&O(n^2) \subset O(n^3) \subset O(2^n) \subset O(n^n)\end{aligned}$$

Relación de inclusión entre los omegas:

$$\begin{aligned}\Omega(1) &\supseteq \Omega(\log n) \supseteq \Omega(\sqrt{n}) \supseteq \Omega(n) \supseteq \Omega(n \log n) \supseteq \\&\Omega(n^2) \supseteq \Omega(n^3) \supseteq \Omega(2^n) \supseteq \Omega(n^n)\end{aligned}$$

### *Comprobación del análisis de un algoritmo*

T(N)/F(N)  
(se duplica el tamaño N en cada experimento)

{  
• T(N) Tiempo medido experimentalmente  
• F(N) tiempo estimado

- T(N)/F(N) converge **a valor positivo**  
**→ F(N) es una estimación correcta**
- T(N)/F(N) converge **a cero**  
**→ F(N) es una sobreestimación**
- T(N)/F(N) **diverge**  
**→ F(N) es una subestimación**

**TABLA PARA SELECCIONSORT (RECURSIVO)**

Entrada(N)	Tiempo de Cpu(ms)(T)	T/(Log * N)	T/N	T/(N * Log N)	T/(N^1,25)	T/(N^2)	T/(N^3)
100	0	0,0000000000	0,0000000000	0,0000000000	0,0000000000	0,0000000000	0,0000000000
200	1	0,4345879897	0,0050000000	0,0021729399	0,0013295740	0,0002500000	0,0000012500
400	1	0,3843108934	0,0025000000	0,0009607772	0,0005590170	0,0000062500	0,0000001560
800	6	2,0667633545	0,0075000000	0,0025834542	0,0014102262	0,0000093750	0,0000001170
1600	27	8,4266507329	0,0168750000	0,0052666567	0,0026681718	0,0000105469	0,0000000660
3200	124	35,3765176289	0,0387500000	0,0110551618	0,0051520992	0,0000121094	0,0000000380
6400	468	122,9579271603	0,0731250000	0,0192121761	0,0081756235	0,0000114258	0,0000000180
12800	1748	425,5930456241	0,1365625000	0,0332494567	0,0128389340	0,0000106689	0,0000000080
25600	7325	1661,6609026822	0,2861328125	0,0649086290	0,0226207850	0,0000111771	0,0000000040

---

**TABLA PARA SHELLSORT**

Entrada(N)	Tiempo de Cpu(ms)(T)	T/(Log * N)	T/N	T/(N * Log N)	T/(N^1,25)	T/(N^2)	T/(N^3)
1000	2	0,6666666667	0,002	0,0006666667	0,000355656	0,0000020000	0,0000000200
2000	5	1,514678754	0,0025	0,000757339	0,000373837	0,0000012500	0,0000000060
4000	10	2,776189187	0,0025	0,000694047	0,000314358	0,0000006250	0,0000000020
8000	28	7,173803344	0,0035	0,000896725	0,00037008	0,0000004375	0,0000000010
16000	59	14,03385256	0,0036875	0,000877116	0,00032787	0,0000002305	0,0000000000
32000	143	31,7414516	0,00446875	0,00099192	0,000334117	0,0000001396	0,0000000000
64000	351	73,03097302	0,005484375	0,001141109	0,000344812	0,0000000857	0,0000000000
128000	893	174,8508492	0,006976563	0,001366022	0,000368841	0,0000000545	0,0000000000

---

TABLA PARA SORT DE STL

Entrada(N)	Tiempo de Cpu(ms)(T)	T/(Log * N)	T/N	T/(N * Log N)	T/(N^1,25)	T/(N^2)	T/(N^3)
10000	11	2,75	0,0011	0,000275	0,00011	0,0000001100	0,0000000000
20000	25	5,812561183	0,00125	0,000290628	0,000105112	0,0000000625	0,0000000000
40000	28	6,084231855	0,0007	0,000152106	4,94975E-05	0,0000000175	0,0000000000
80000	59	12,03322806	0,0007375	0,000150415	4,3852E-05	0,0000000092	0,0000000000
160000	122	23,4429645	0,0007625	0,000146519	0,000038125	0,0000000048	0,0000000000
320000	250	45,41202347	0,00078125	0,000141913	3,28475E-05	0,0000000024	0,0000000000
640000	1028	177,0527274	0,00160625	0,000276645	5,67895E-05	0,0000000025	0,0000000000
1280000	2048	335,3413441	0,0016	0,000261985	4,75683E-05	0,0000000013	0,0000000000

Ejemplos de código con  $O(\log n)$  :

```
c= N;
while (c > 1) {
    algo_de_O(1)
    c= c/2;
}
```

```
c= 1;
while (c < N) {
    algo_de_O(1)
    c= 2*c; }
```

y con  $O(n \log n)$  :

```
for (int i=0; i<N; i++) {
    c= i;
    while (c > 1) {
        algo_de_O(1)
        c= c/2; }
}
```

## Ejemplo: Promedio de Prefijos

Dado un array A de n números, calcular otro array B, tal que:

$$B[i] = \frac{\sum_{j=0}^i A[j]}{i+1}$$

A partir de una secuencia de números A[i] calculamos otra tal que cada uno de sus elementos sea el promedio de todos los anteriores en la secuencia original

(Muy usado en estadística y economía)

## Promedio de Prefijos - Solución 1

```
Algoritmo promedioPrevios1(A, n)
    Input array A de n enteros
    Output arreglo B de promedio previos de A
    Crear B de n elementos enteros
    for i=0 to n-1 do
        s=0
        for j=0 to i do
            s=s+A[j]
        B[i]=s/(i+1)
    return B
```

# operaciones
n
n
n
1+2+...+n
1+2+...+n
n
n

- Partes:

- Inicializar y devolver array B: O(n)

- Bucle i: se ejecuta n veces

- Bucle j: se ejecuta  $1+2+3+\dots+n = n(n+1)/2$  veces: O( $n^2$ )

Total:  $O(n)+O(n)+O(n^2)=O(n^2)$  (Orden cuadrático)

## Promedio de Prefijos - Solución 2

```
Crear B de n elementos enteros          n
s=0                                     1
for i=0 to (n-1) do                   n
    s=s+A[i]                         n
    B[i]=s/(i+1)                     n
return B                                n
```

### Partes:

- Iniciar y devolver array B:  $O(n)$
- Iniciar s:  $O(1)$
- Bucle i: se ejecuta **n veces**
  - Total:  $O(n)+O(1)+O(n)=O(n)$  (**Orden lineal**)

## Complejidad

### Limitaciones del análisis asintótico

- No es apropiado para pequeñas cantidades de datos
- El coeficiente del término dominante puede intervenir cuando un algoritmo es suficientemente complejo
- A veces el análisis asintótico es una sobreestimación
  - La cota de tiempo de ejecución en el caso promedio puede ser significativamente menor que la cota en el caso peor
  - El caso peor es en ocasiones poco representativo por lo que puede ser ignorado

## Complejidad

### Ejemplos con logaritmos

- Un entero **short** de 16 bits almacena enteros hasta 32767.  
**Para representar N enteros** es necesario B bits.  $B \geq \log_2 N$ .  
Luego el número mínimo de bits es  $\lceil \log_2 N \rceil + 1 \rightarrow O(\log_2 N)$
- Comenzando con  $X=N$ , si  $N$  se divide de forma repetida por la mitad, **cuántas veces hay que dividir para hacer  $N \leq 1$ ?**  
Si la división redondea al entero más cercano, el número de divisiones necesarias es  $\lceil \log_2 N \rceil + 1 \rightarrow O(\log_2 N)$
- Aplicando el principio de **sucesivas divisiones por la mitad**, los tiempos de ejecución de estos algoritmos son  $O(\log N)$  si se tarda un tiempo constante  $O(1)$  en dividir el problema por una fracción constante  $\frac{1}{2}$
- Esto **vale para cualquier fracción constante** (la fracción está reflejada en la base del logaritmo, que no es importante)

57

---

## Complejidad

### Problema de búsqueda estática

Dado un entero  $X$  y un vector  $A$ , devolver la posición de  $X$  en  $A$  o una indicación de que no está presente. Si  $X$  aparece más de una vez, devolver cualquier aparición.  $A$  no puede ser modificado

- Si  $A$  no está ordenado  $\rightarrow$  **búsqueda secuencial** lineal  
**Peor caso:** búsqueda sin éxito  $\rightarrow O(N)$   
**Caso Medio:** en promedio buscamos  $N/2 \rightarrow O(N)$
- Si  $A$  está ordenado  $\rightarrow$  **búsqueda binaria**  
**Peor caso:** búsqueda sin éxito  $\rightarrow O(\log N)$   
**Caso Medio:** solo se necesita una iteración menos
  - $\frac{1}{2}$  de los elementos llevan al peor caso
  - $\frac{1}{4}$  de los elementos ahorran una búsqueda.....
  - $1/2^i$  ahorrarán  $i$  iteraciones

58

---

## Complejidad

### Problema de búsqueda estática

- *Solución 1*

```
BB ( A,X)
    prin = 1; fin = N;
    while (prin <= fin) do
        medio = (prin+fin)/2;
        si A [medio] < X then prin=medio+1
        else si A[medio] > X then fin=medio-1
        else return (medio)
    endwhile;
    error
```

59

---

## Complejidad

### Problema de búsqueda estática

- *Solución 2 - más rápida- eliminar una comprobación*

```
BB (A,X)
    prin = 1; fin = N;
    while (prin < fin) do
        medio = (prin + fin )/2;
        si A [medio] < X then prin = medio +1
        else fin = medio
    endwhile;
    si A[prin] = X then return inicio;
    error
```

- Ahorramos preguntas (cambia la constante) pero igualmente no bajamos el orden →  $O(\log N)$
- Para valores de  $N < 6$  no vale la pena
- Se puede usar una estrategia híbrida: la b. binaria termina cuando se tiene un rango pequeño, y luego se aplica secuencial para terminar

60

---

## Complejidad

### Problema de búsqueda estática

#### BÚSQUEDA INTERPOLADA (en ocasiones más rápido que bb)

##### Hipótesis:

- Cada **acceso es muy costoso** comparado con una instrucción típica (ejemplo arreglo en disco)
- Los datos además de ordenados, deben estar **uniformemente distribuidos** (ej: 1,2,4,8,16 no es uniforme)

Se aplica :

$$\text{próximo} = \text{prin} + \left[ \frac{\text{X} - \text{A}[\text{prin}]}{\frac{\text{A}[\text{fin}] - \text{A}[\text{prin}]}{\text{fin}-\text{prin}-1}} \right]$$

- El número medio de comparaciones es  **$O(\log \log N)$**
- En el peor caso de distribución, la ejecución será lineal.
- Ej: 1000 elementos entre 1000 y 1.000.000 y buscamos 12.000

61

---

## Complejidad

### Problema de búsqueda estática

#### BÚSQUEDA INTERPOLADA

- Los cálculos son más costosos que en la BB.
- Las hipótesis son muy restrictivas, por lo que en la práctica generalmente no conviene usarla.
- Pero es interesante ver que hay más de una forma de resolver un problema y que ningún algoritmo (incluso el de búsqueda binaria) es el mejor en todas las situaciones.

La mejora es muy importante: Para  $N=4$  millones,  $\log(N) \approx 22$  y  $\log(\log(N)) \approx 5$ .

62

---

## Complejidad

### Otro ejemplo: Subsecuencia máxima

- Problema:

Dada una secuencia de enteros (posiblemente negativos) encontrar (e identificar) la subsecuencia correspondiente al valor máximo de  $\sum$  de  $A_i$  hasta  $A_j$ .

Cuando todos los valores son negativos, retornar secuencia vacía y suma 0.

63

## Complejidad

### Problema: subsecuencia de suma máxima

#### Algoritmo $O(N^3)$

```
S=0;  
for i=1,N do  
    for j=i,N do  
        SA = 0;  
        for K =i, j do  
            SA = SA + A[k]          (*)  
        endfor;  
        if SA > S then S=SA; ini=i; fin=j  
        endif  
    endfor  
endfor;  
return S
```

- Medida dominante del trabajo realizado = número de veces que se ejecuta (\*) = número de ternas ordenadas  $(i,j,k)$  que satisfacen  $1 \leq i \leq k \leq j \leq N$
- Una aproximación rápida da  $NxNxN = N^3$
- Un cálculo más preciso que contempla las restricciones :  
$$\sum_{i=1,N} \sum_{j=1,N} \sum_{k=1,j} 1 = N(N+1)(N+2)/6$$
 (Var (N+2,3)). Pero el orden sigue  $O(N^3)$

Co

Un

S=0  
for

en

re

• |  
• |  
• |

64

## Complejidad

Problema: subsecuencia de suma máxima

Un algoritmo mejorado -  $O(N^2)$

)

```
S=0;
for i=1,N do
    SA = 0;
    for j=i,N do
        SA = SA + A[j];
        if SA>S then
            S=SA; ini=i; fin=j
        endif
    endfor
endfor;
return S
```

- Evita cálculos innecesarios ( $\sum_{k=i,j} A[k] = A[j] + \sum_{k=i,j-1} A[k]$ )
- Ambos realizan **búsqueda exhaustiva** (examinan todas las posibles subsecuencias)

54

65

---

## Complejidad

Problema: subsecuencia de suma máxima

- **Solución lineal**

```
S=0; SA=0; i=1;
for j=i,N do
    SA = SA + A[j];
    if SA>S then { S=SA; ini=i; fin=j; }
    else if SA<0 then { SA=0; i=j+1; }
    endif;
endif;
endfor;
return S
```

68

---

```
PROCEDURE Sumamax4(VAR a:vector;prim,ult:CARDINAL):CARDINAL;
```

---

```

PROCEDURE Sumamax4(VAR a:vector;prim,ult:CARDINAL):CARDINAL;
  VAR i:CARDINAL;
      suma,max_anterior,max_aux:INTEGER;
BEGIN
  max_anterior:=0;
  max_aux:=0;

  FOR i:=prim TO ult DO
    max_aux:=Max2(max_aux+a[i],0);
    max_anterior:=Max2(max_anterior,max_aux)
  END;
  RETURN max_anterior;
END Sumamax4;

```

---

### Conclusión- Complejidad

Antes de realizar un programa conviene elegir un [buen algoritmo](#), donde por bueno entendemos que utilice pocos recursos, siendo usualmente los más importantes el tiempo que lleve ejecutarse y la cantidad de espacio en memoria que requiera.

[Es engañoso](#) pensar que todos los algoritmos son "más o menos iguales" y confiar en nuestra habilidad como programadores para convertir un mal algoritmo en un producto eficiente.

Es asimismo [engañoso confiar](#) en la creciente potencia de las máquinas y el abaratamiento de las mismas como remedio de todos los problemas que puedan aparecer.

En el análisis de algoritmos se considera usualmente [el caso peor](#), si bien a veces conviene analizar igualmente el caso mejor y hacer alguna estimación sobre un caso promedio.

Para [independizarse de factores coyunturales](#) tales como el lenguaje de programación, la habilidad del codificador, la máquina soporte, etc. se suele trabajar con un cálculo asintótico que indica como se comporta el algoritmo para datos muy grandes.

[Para problemas pequeños](#) es cierto que casi todos los algoritmos son "más o menos iguales", primando otros aspectos como esfuerzo de codificación, legibilidad, etc. Los órdenes de complejidad sólo son importantes para grandes problemas. 69

## Problemas P, NP y NP-completos

Hasta aquí hemos venido hablando de algoritmos.

Cuando nos enfrentamos a un problema concreto, habrá una serie de algoritmos aplicables. Se suele decir que el **orden de complejidad de un problema es el del mejor algoritmo que se conozca para resolverlo**.

Así se clasifican los problemas, y los estudios sobre algoritmos se aplican a la realidad. Estos estudios han llevado a la constatación de que existen problemas muy difíciles, problemas que desafían la utilización de los ordenadores para resolverlos.

Esbozaremos las clases de problemas que hoy por hoy se escapan a un tratamiento informático.

- Clase P.
- Clase NP.
- Clase NP-completos.

70

---

## Problemas P, NP y NP-completos

### •Clase P.-

- Los algoritmos de **complejidad polinómica** se dice que son **tratables** en el sentido de que suelen ser abordables en la práctica. Los problemas para los que se conocen algoritmos con esta complejidad se dice que forman la clase P. Aquellos problemas para los que la mejor solución que se conoce es de complejidad superior a la polinómica, se dice que son **problemas intratables**. Sería muy interesante encontrar alguna solución polinómica (o mejor) que permitiera abordarlos.

### •Clase NP.-

- Algunos de estos problemas intratables pueden caracterizarse por el curioso hecho de que puede aplicarse un algoritmo polinómico para comprobar si una posible solución es válida o no. Esta característica lleva a un método de resolución no determinista consistente en aplicar heurísticos para obtener soluciones hipotéticas que se van desestimando (o aceptando) a ritmo polinómico. Los problemas de esta clase se denominan NP (la N de no-deterministas y la P de polinómicos).

71

---

## **Problemas P, NP y NP-completos**

### **•Clase NP-completos.**

•Se conoce una amplia variedad de problemas de tipo NP, de los cuales destacan algunos de ellos de extrema complejidad. Gráficamente podemos decir que algunos problemas se hallan en la "**frontera externa**" de la clase NP. Son problemas NP, y son los peores problemas posibles de clase NP. Estos problemas se caracterizan por ser todos "iguales" en el sentido de que si se descubriera una solución P para alguno de ellos, esta solución sería fácilmente aplicable a todos ellos. Actualmente hay un premio de prestigio equivalente al Nobel reservado para el que descubra semejante solución ... ¡y se duda seriamente de que alguien lo consiga!

•Es más, si se descubriera una solución para los problemas NP-completos, esta sería aplicable a todos los problemas NP y, por tanto, la clase NP desaparecería del mundo científico al carecerse de problemas de ese tipo. Realmente, tras años de búsqueda exhaustiva de dicha solución, es hecho ampliamente aceptado que no debe existir, aunque nadie ha demostrado, todavía, la imposibilidad de su existencia.