

# **Análisis de la eficiencia de algoritmos**

# Motivación

- Podemos diseñar diferentes algoritmos para resolver un problema.
- Cada uno de ellos puede consumir más o menos recursos, tiempo... ¿Cuál de ellos es el mejor?
- Ejemplo: asignar 5 trabajadores a 5 trabajos distintos, según sus capacidades. Fácil, ¿verdad?
- Pero, ¿y si tenemos que asignar 50 trabajadores a 50 puestos? ¡¡Serían 50! Posibilidades!!  
¡¡Aproximadamente,  $10^{64}$ !!
- ¡¡¡Si un ordenador evalúa 1 billón de posibilidades por segundo, necesitará más de 15000 millones de años!!!

# Tamaño del problema

- Un algoritmo no tiene un tiempo de ejecución fijo. Éste depende del tamaño del problema.
- Si queremos comparar dos algoritmos, no podemos hacerlo para un único tamaño de problema.
- Debemos expresar el tiempo de ejecución como una función,  $f(n)$ , del tamaño del problema,  $n$ .

$$\begin{aligned} f : N &\rightarrow \mathfrak{R}_0^+ \\ n &\rightarrow f(n) \end{aligned}$$

- Para comparar algoritmos, comparamos sus tiempos de ejecución (compararemos sus funciones).

# Tamaño del problema

Tamaño	$\log_2 n$	$n$	$n \log_2 n$	$n^2$	$n^3$	$2^n$
10	3.3 ns	10 ns	33 ns	100 ns	1 $\mu$ s	1 $\mu$ s
20	4.3 ns	20 ns	86 ns	400 ns	8 $\mu$ s	1 ms
30	4.9 ns	30 ns	147 ns	900 ns	27 $\mu$ s	1 s
40	5.3 ns	40 ns	213 ns	2 $\mu$ s	64 $\mu$ s	18.3 min
50	5.6 ns	50 ns	282 ns	3 $\mu$ s	125 $\mu$ s	13 días
100	6.6 ns	100 ns	664 ns	10 $\mu$ s	1 ms	$40 \times 10^{12}$ años
1000	10 ns	1 $\mu$ s	10 $\mu$ s	1 ms	1 s	
10000	13 ns	10 $\mu$ s	133 $\mu$ s	100 ms	16.7 min	
100000	17 ns	100 $\mu$ s	2 ms	10 s	11.6 días	
1000000	20 ns	1 ms	20 ms	16.7 min	31.7 años	

Tiempos de ejecución en una máquina que realiza  $10^9$  pasos por segundo (1 GHz), según el tamaño del problema y el coste del algoritmo

# Algoritmos versus implementaciones

Debemos distinguir entre:

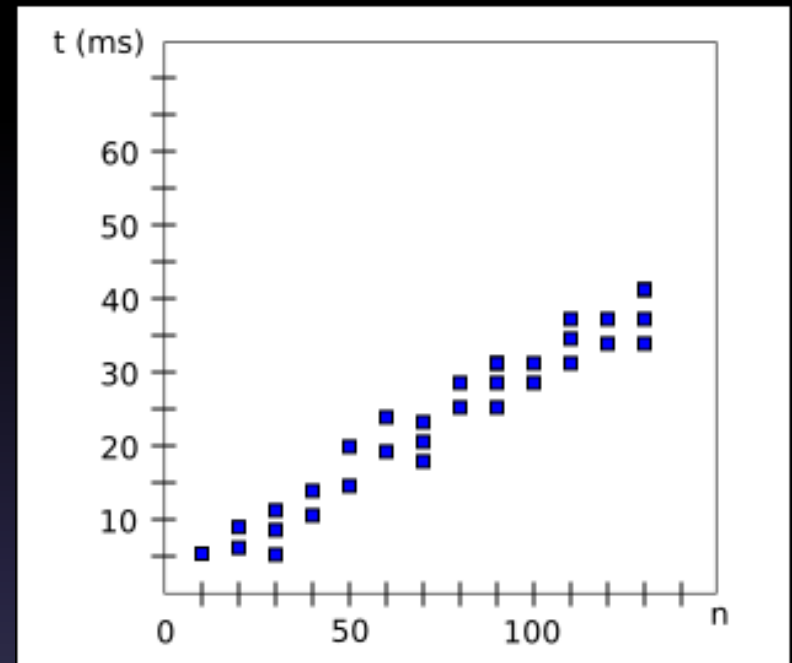
- Algoritmo: conjunto finito de pasos que nos llevan a resolver un problema.
- Implementación: realización de un algoritmo en un lenguaje de programación determinado

**CENTRAREMOS NUESTRO ANÁLISIS EN ALGORITMOS,  
Y NO EN IMPLEMENTACIONES**

# Medición experimental del tiempo de ejecución

## Estudio experimental:

- Escribir un programa que implemente el algoritmo
- Ejecutar el programa con conjuntos de datos de diferente tamaño y composición
- Usar algún método para tomar una medida adecuada del tiempo de ejecución



# Más allá de los estudios experimentales

Limitaciones de los estudios experimentales:

- Es necesario implementar y testear el algoritmo
- Los experimentos se hacen sobre un conjunto limitado de entradas, que pueden no ser suficientemente representativas
- Para comparar dos algoritmos, deben usarse los mismos entornos hardware y software

# Más allá de los estudios experimentales

Por lo tanto,

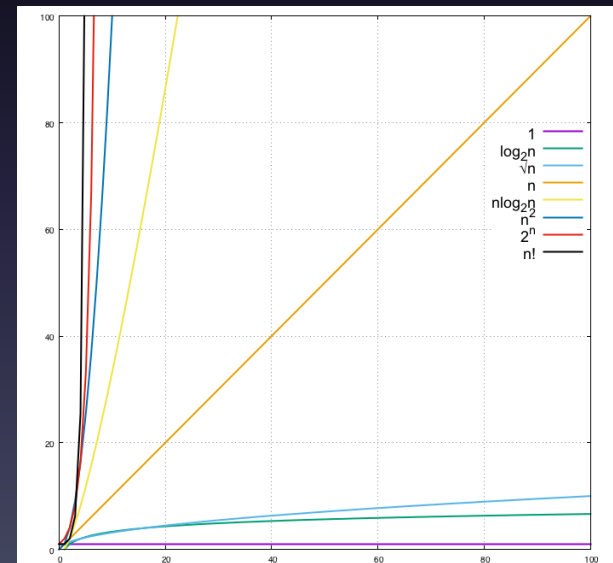
- Se desarrollará una metodología general para analizar el tiempo de ejecución de un algoritmo que:
  - Usa la descripción de alto nivel del algoritmo
  - Tiene en cuenta todas las posibles entradas
  - Permite evaluar la eficiencia de un algoritmo con independencia del hardware y del software

EFICIENCIA TEÓRICA vs. EXPERIMENTAL O EMPÍRICA

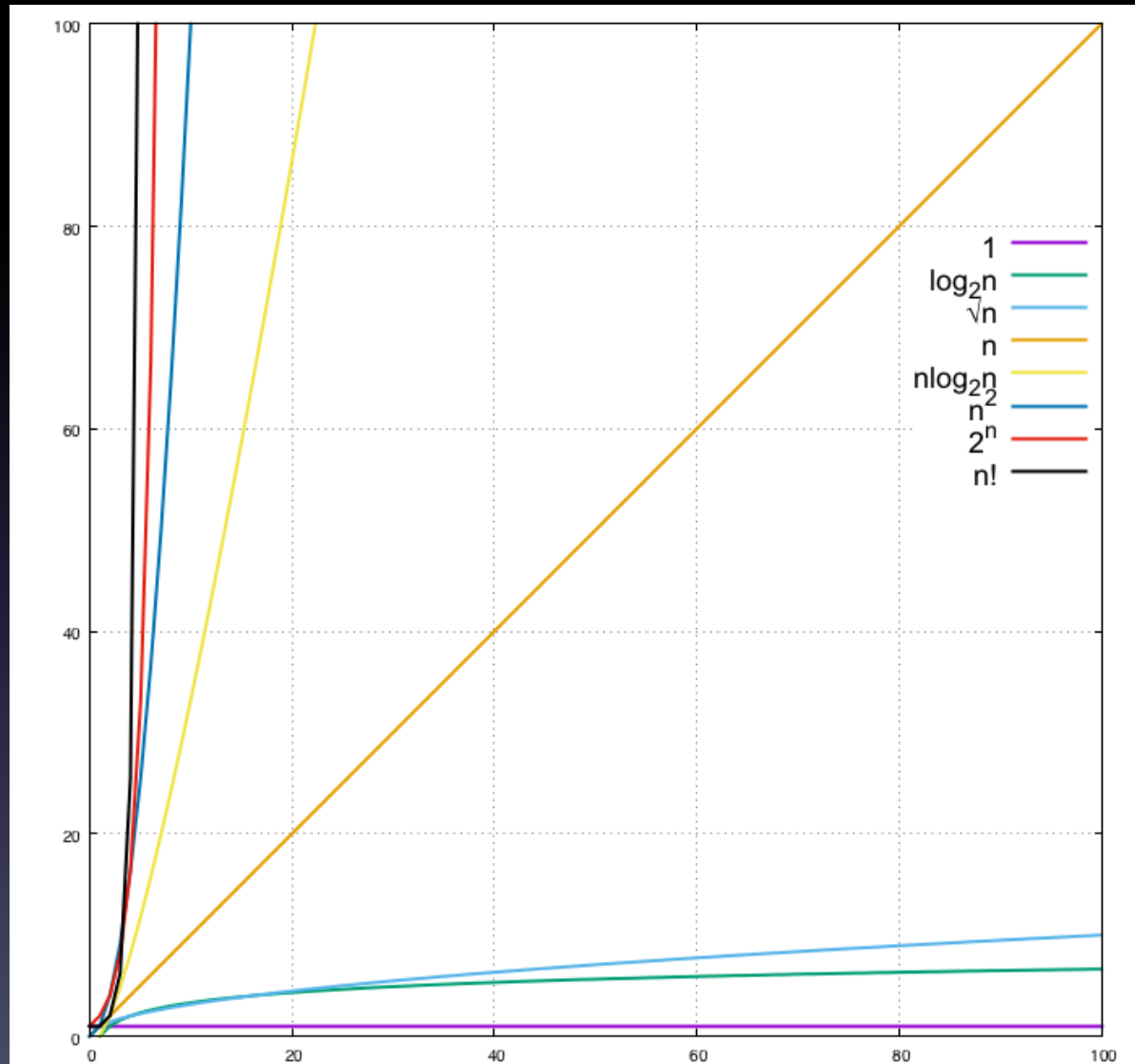


# Familias de órdenes de eficiencia

- Un algoritmo tiene un **tiempo de ejecución**  $t(n)$  si existe una constante positiva,  $c$ , y una implementación del algoritmo capaz de resolver cualquier ejemplo del problema en un tiempo acotado por  $ct(n)$ , siendo  $n$  el tamaño del problema
- Algunos tiempos de ejecución habituales:
  - $n$ : lineal
  - $n^2$ : cuadrático
  - $n^k$ : polinómico ( $k$  natural)
  - $\log n$ : logarítmico
  - $c^n$ : exponencial



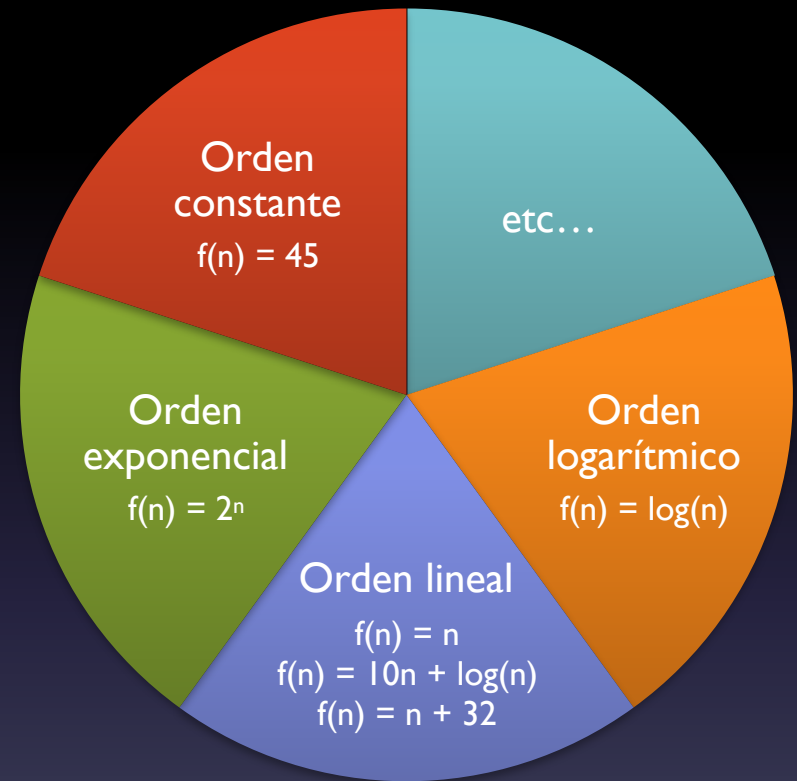
# Familias de órdenes de eficiencia



# Familias de órdenes de eficiencia

Cuando queramos estudiar el tiempo de ejecución de un algoritmo independientemente de la implementación, tendremos que determinar la clase de equivalencia a la que pertenece la función de su tiempo de ejecución, esto es, determinar su

**ORDEN DE EFICIENCIA**



# Comparación de órdenes de eficiencia

- La determinación de la eficiencia de un algoritmo dependerá del comportamiento de su orden de eficiencia cuando  $n$  crece, esto es, cuando  $n \rightarrow \infty$
- Comparamos **perfiles de crecimiento**: un algoritmo es más eficiente que otro si su perfil de crecimiento tiene un crecimiento menor
- Condiciones para comparar dos órdenes de eficiencia:
  - El resultado no puede depender del resultado para un número finito de valores de la función
  - El resultado no puede depender de las funciones concretas que representan las correspondientes clases

# Comparación de órdenes de eficiencia

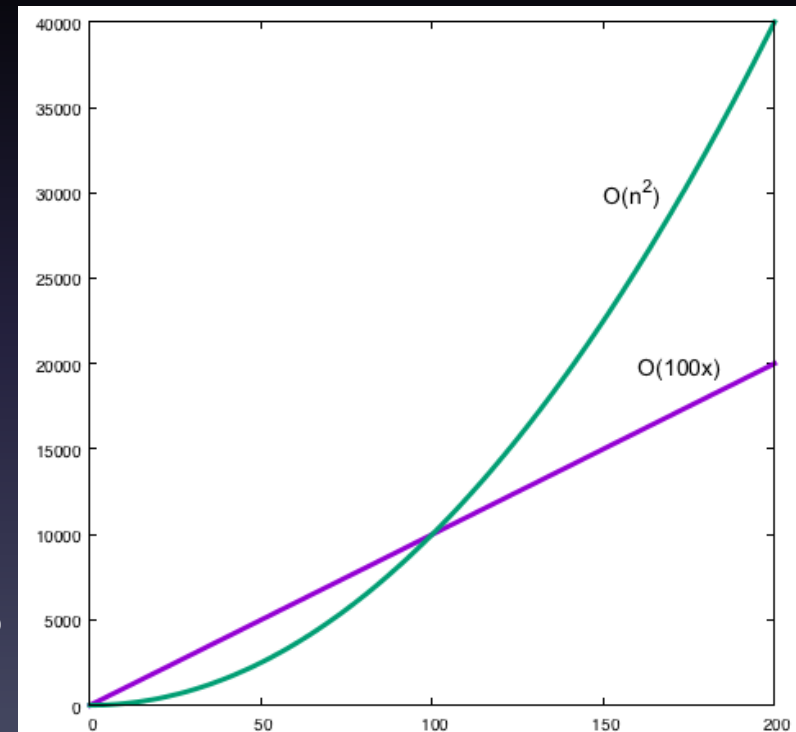
- Dadas  $f, g: \mathbb{N} \rightarrow \mathbb{R}_0^+$ , diremos que  $g(n)$  es menor o igual que  $f(n)$  si

$$\exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}, \text{ tales que } \forall n \geq n_0, \quad g(n) \leq c \cdot f(n)$$

$g(n)=100n$  es menor que  $f(n)=n^2$  ya que para  $c=100$  y  $n_0=1$  se cumple que

$$\forall n \geq 1 \quad 100 \cdot n \leq 100 \cdot n^2$$

Este orden es sólo parcial, ya que habrá parejas de funciones que no podrán ordenarse



# Notación $O$

- Objetivo: poder indicar de forma clara y precisa el grado de eficiencia obtenido en el análisis del algoritmo
- Pasos:
  1. Análisis del tiempo de ejecución: estudiar la función que indica el tiempo necesario para cada  $n$
  2. Análisis de eficiencia: clasificar ese tiempo de ejecución en una familia de funciones

Si tenemos ordenadas las familias de funciones, podremos comparar algoritmos

# Notación O

- Objetivo: eliminar información innecesaria, simplificando el análisis. Ej.:  $3n^2 + 5 \simeq n^2$

## **NOTACIÓN O GRANDE (Big O)**

- Dadas dos funciones,  $f(n)$  y  $g(n)$ , decimos que  $f(n)$  es  $O(g(n))$  si  $f(n) \leq cg(n)$  para  $n \geq n_0$ , con  $c$  y  $n_0$  constantes  
Ej.:  $(n+1)^2$  es  $O(n^2)$  [ $n_0=1, c=4$ ]  
Ej.:  $3n^3+2n^2$  es  $O(n^3)$  [ $n_0=0, c=5$ ]  
Ej.:  $3^n$  no es  $O(2^n)$
- Aunque es cierto que, por ejemplo,  $7n+5$  es  $O(n^5)$ , la idea es buscar el menor orden posible, esto es, la menor cota superior posible

# Notación $O$

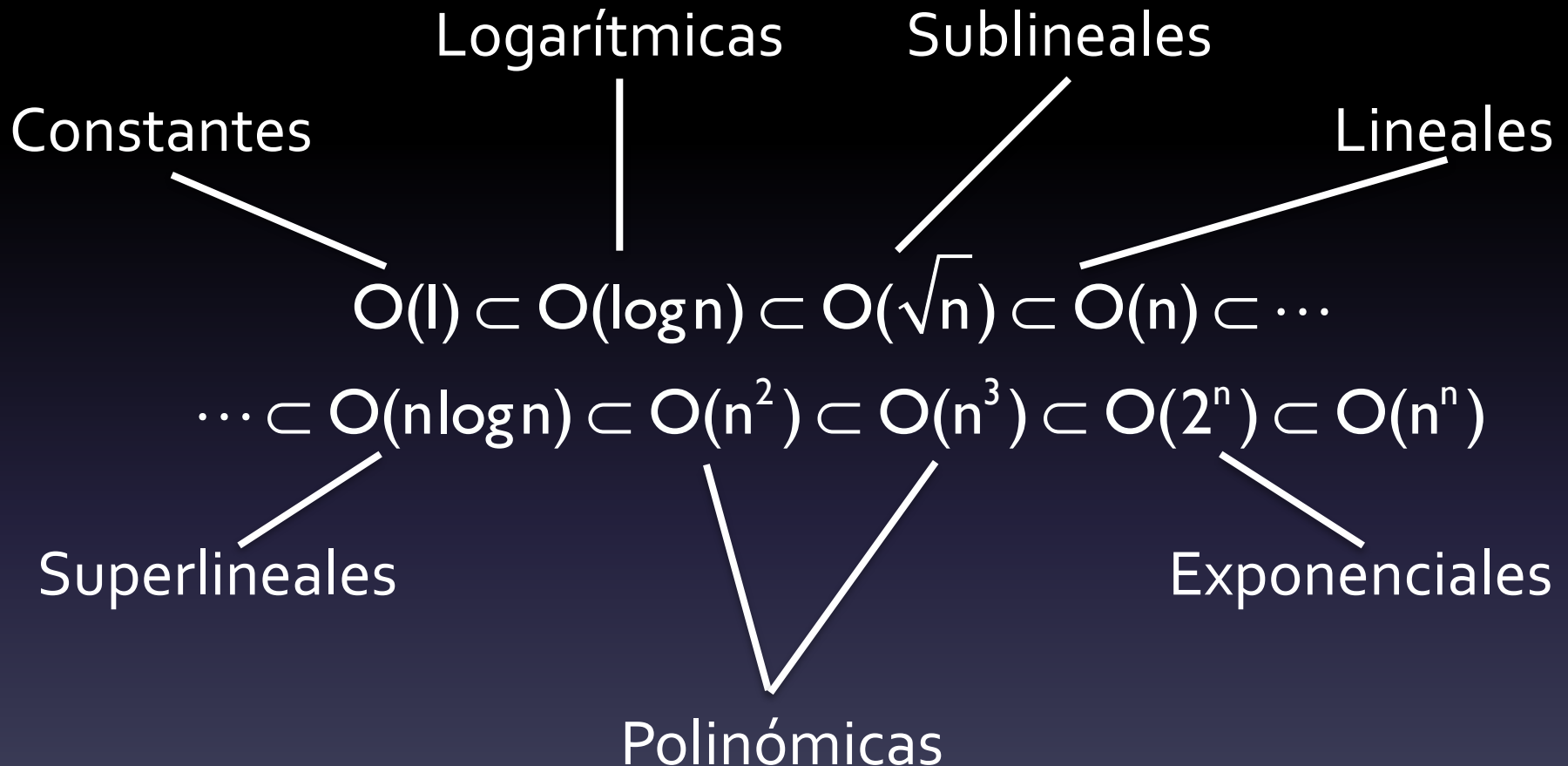
- Una primera regla muy sencilla: *eliminar los términos de orden menor y los factores constantes. Así:*
  - $7n-3$  es  $O(n)$
  - $8n^2\log_2 n + 5n^2 + n$  es  $O(n^2\log_2 n)$
  - $3n^2 + 2n$  es  $O(n^2)$
  - $n/2 + \log_2 n$  es  $O(n)$
  - $225$  es  $O(1)$
- Algunos casos especiales:
  - Logarítmico  $O(\log n)$
  - Lineal  $O(n)$
  - Cuadrático  $O(n^2)$
  - Polinomial  $O(n^k)$ ,  $k > 1$
  - Exponencial  $O(a^n)$



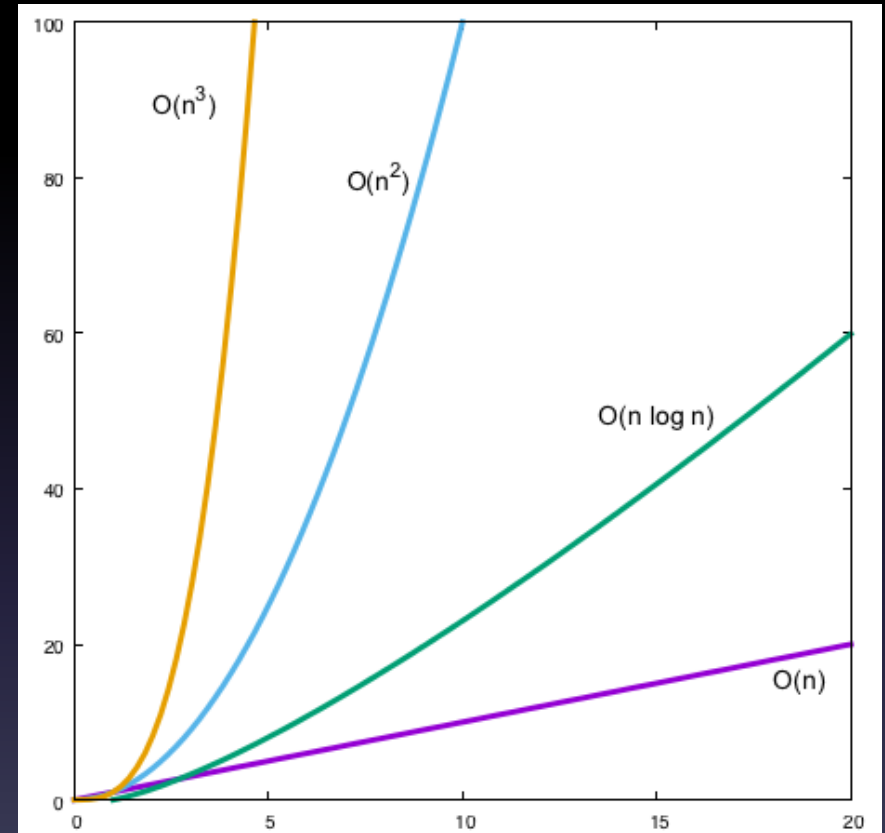
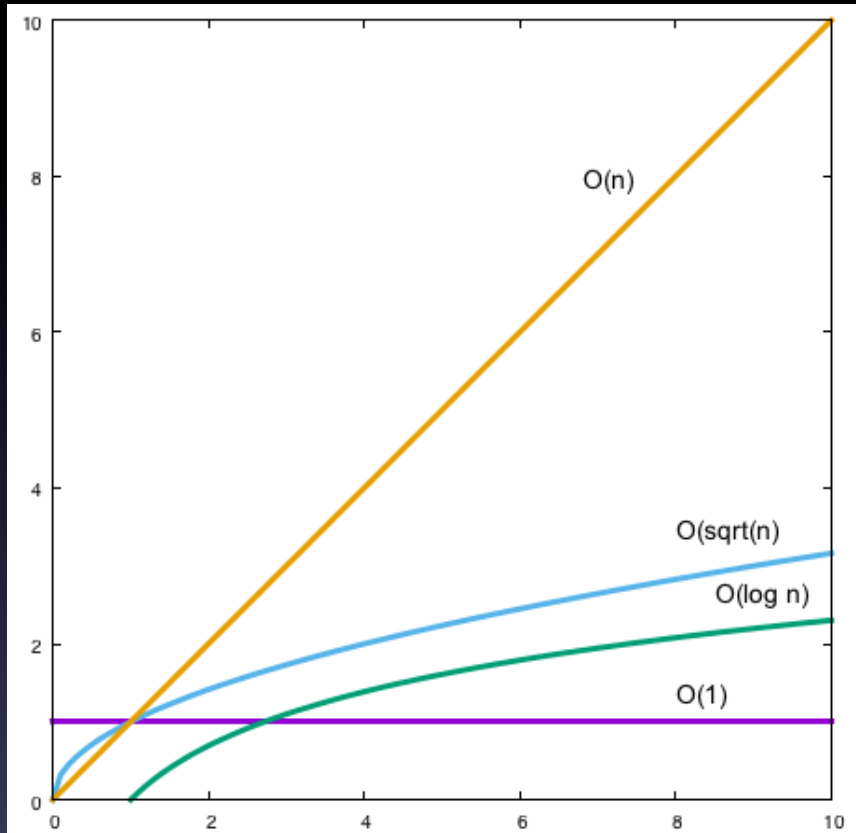
# En resumen...

- La notación  $O$  nos permite acotar y caracterizar el tiempo de ejecución de un algoritmo de forma independiente de la máquina, la implementación, etc.
- El signo  $\leq$  implica la idea de eficiencia en el peor caso
- Aunque, p.ej.,  $2n^2+3$  es  $O(n^2)$  y también es  $O(n^3)$ , la primera es mejor, ya que caracteriza mejor la función
- La notación  $O$  nos permite expresar el número de operaciones primitivas en función del tamaño del problema
- La notación  $O$  nos permitirá comparar tiempos de ejecución:  $O(n)$  es mejor que  $O(n^2)$ , ó  $O(\log n)$  es mejor que  $O(n)$ ... → JERARQUÍA DE FUNCIONES

# Jerarquía de funciones

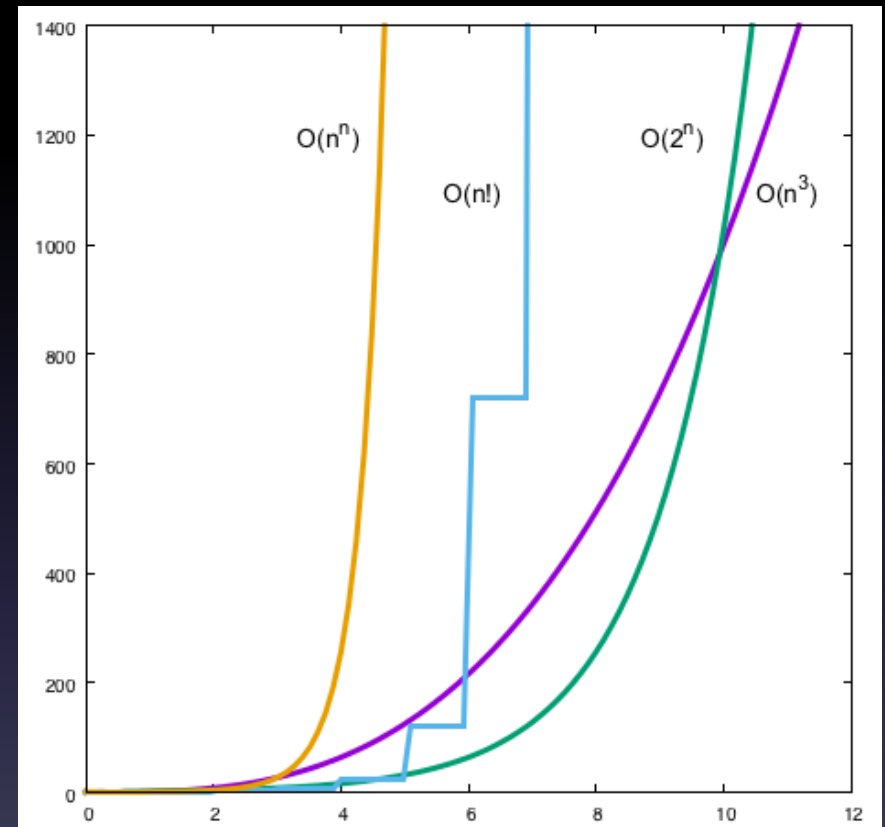
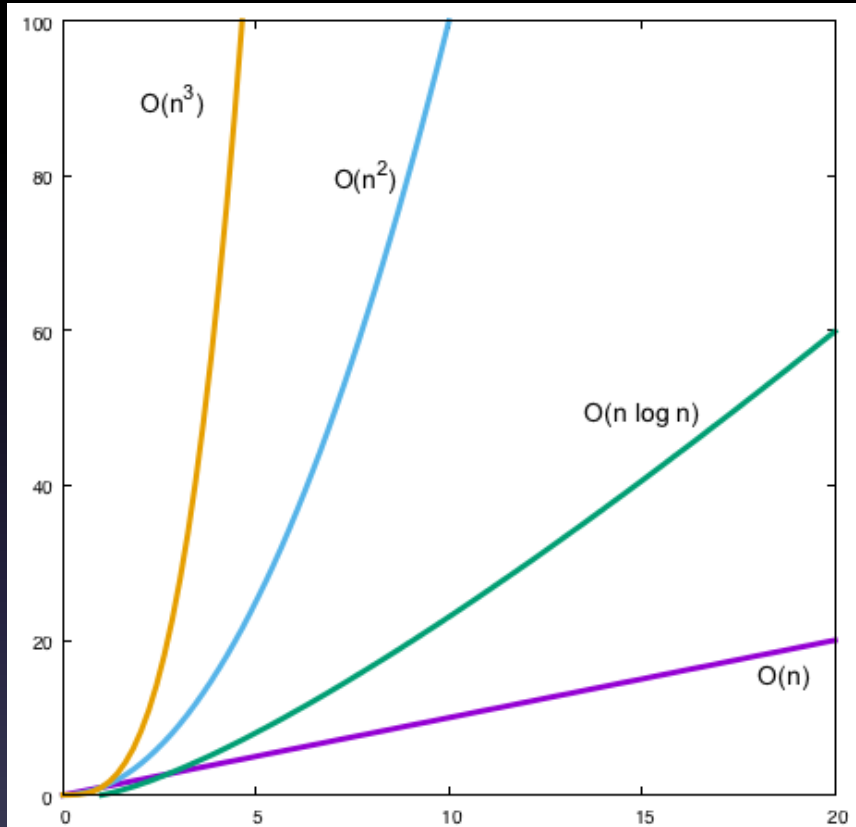


# Jerarquía de funciones



Sublineales, lineales y superlineales

# Jerarquía de funciones



Superlineales, polinómicas y exponenciales

# Algunas reglas simples...

- Transitividad  $\left. \begin{array}{l} f(n) \text{ es } O(g(n)) \\ g(n) \text{ es } O(h(n)) \end{array} \right\} f(n) \text{ es } O(h(n))$
- Polinomios  $a_d n^d + \dots + a_1 n + a_0 \text{ es } O(n^d)$
- Jerarquía de funciones  
 $n + \log_2 n \text{ es } O(n)$   $2^n + n^3 \text{ es } O(2^n)$
- Bases y potencias de logaritmos pueden ignorarse  
 $\log_a n \text{ es } O(\log_b n)$   $\log(n^2) \text{ es } O(\log n)$
- Bases y potencias de exponentes no pueden ignorarse  
 $3^n \text{ no es } O(2^n)$   $a^{n^2} \text{ no es } O(a^n)$

# Elección del mejor algoritmo

- En la práctica, debemos tener en cuenta otros factores, además de la eficiencia:
  - Tamaño de los problemas a resolver
  - Requisitos de espacio y tiempo del sistema
  - Complejidad de implantación y mantenimiento de los algoritmos
- Ejemplo:
  - Algoritmo 1 con  $t_1(n) = 100n$       Lineal [ $O(n)$ ]
  - Algoritmo 2 con  $t_2(n) = n^2/5$       Cuadrático [ $O(n^2)$ ]

En teoría está claro, pero ¿y en la práctica?

# Elección del mejor algoritmo

- El segundo algoritmo, cuadrático, puede ser preferible si:
  - El tamaño de los problemas a resolver no a pasar de 100 [¡constante multiplicativa!]
  - El algoritmo 1 tiene unos requerimientos de memoria muy superiores. P.ej., si el algoritmo 1 requiere una cantidad de espacio cuadrática respecto al tamaño, mientras que el algoritmo 2 tiene requerimiento constante
  - El algoritmo 1 tiene costes de implementación y mantenimiento muy superiores al algoritmo 2

# Notación O

- Decimos que una función,  $T(n)$ , es  $O(g(n))$  si existen constantes  $c$  y  $n_0$  tales que  $f(n) \leq cg(n)$  para  $n \geq n_0$   
 $T(n)$  es  $O(f(n)) \Leftrightarrow \exists c \in \mathbb{R}, \exists n_0 \in \mathbb{N}$  tq  $\forall n \geq n_0, T(n) \leq c \cdot f(n)$
- Seremos flexibles en la notación: Usaremos  $O(f(n))$  aun cuando en un número finito de valores de  $n$ ,  $f(n)$  no esté definida o sea negativa. Ej: logaritmos



# Notación $O$

- **Regla de la suma**

Si  $T_1(n)$  y  $T_2(n)$  son los tiempos de ejecución de dos segmentos de código tales que  $T_1(n)$  es  $O(f(n))$  y  $T_2(n)$  es  $O(g(n))$ , entonces

$$T_1(n) + T_2(n) \text{ es } O(\max(f(n), g(n)))$$

- **Regla del producto**

Si  $T_1(n)$  y  $T_2(n)$  son los tiempos de ejecución de dos segmentos de código tales que  $T_1(n)$  es  $O(f(n))$  y  $T_2(n)$  es  $O(g(n))$  (y no son negativos para ningún  $n$ ), entonces

$$T_1(n) T_2(n) \text{ es } O(f(n)g(n))$$

# Cálculo de la eficiencia

- **Operación elemental:** operación de un algoritmo cuyo tiempo de ejecución se puede acotar superiormente por una constante
- Nos interesa el número de operaciones elementales, no el tiempo concreto requerido para ejecutarlas
- No nos interesa saber la función exacta que indica el tiempo de ejecución, sino cualquiera que corresponda a la misma eficiencia, al mismo orden. Esto simplificará enormemente nuestro análisis.

# Cálculo de la eficiencia

- Ejemplo: un algoritmo con
  - $n_s$  sumas ( $t_s$ )
  - $n_a$  asignaciones ( $t_a$ )
  - $n_m$  multiplicaciones ( $t_m$ )

Podemos ver que

$$t \geq \min(t_s, t_a, t_m) \cdot (n_s + n_a + n_m)$$

y

$$t \leq \max(t_s, t_a, t_m) \cdot (n_s + n_a + n_m)$$

Por lo tanto,  $c_1 \cdot n \leq t \leq c_2 \cdot n$

Pero ¿si las constantes no importan! Por lo tanto, el algoritmo es  $O(n)$

# Cálculo de la eficiencia

- **Estructura secuencial:** Si  $S_1$  y  $S_2$  son dos segmentos de código con eficiencias  $O(f_1(n))$  y  $O(f_2(n))$ , la eficiencia de su unión secuencial es  $O(f_1(n) + f_2(n))$ .

Por la regla de la suma, tenemos  $O(\max(f_1(n), f_2(n)))$ .

- **Estructura condicional:** Si la condición requiere un tiempo  $O(E(n))$  y el camino más costoso requiere un tiempo  $O(f(n))$ , la eficiencia total de la estructura condicional es la suma de ambas.

Por la regla de la suma, tenemos  $O(\max(E(n), f(n)))$ .

# Cálculo de la eficiencia

- **Estructura iterativa:**

- Bucle for:

$$O(\text{Ini}(n)) + O(\text{Con}(n)) + O(\text{Ite}(n)) \cdot [O(\text{Cu}(n)) + O(\text{Inc}(n)) + O(\text{Con}(n))]$$

- Bucle while:

$$O(\text{Con}(n)) + O(\text{Ite}(n)) \cdot [O(\text{Cu}(n)) + O(\text{Con}(n))]$$

- Bucle do-while:

$$O(\text{Ite}(n)) \cdot [O(\text{Cu}(n)) + O(\text{Con}(n))]$$

donde:

$O(\text{Ini}(n))$ : Inicialización

$O(\text{Con}(n))$ : Condición

$O(\text{Ite}(n))$ : Iteración

$O(\text{Cu}(n))$ : Cuerpo

$O(\text{Inc}(n))$ : Incremento

# Cálculo de la eficiencia

- Ejemplo:

```
for (i=0; i<n; i++)  
    A[i][j] = 0;
```

1 + 1 + 4n operaciones



1 + 1n operaciones

- ▶ Antes de entrar en el bucle:
  - ▶ 1 asignación
  - ▶ 1 evaluación de condición
- ▶ En cada iteración
  - ▶ 1 indexación
  - ▶ 1 asignación
  - ▶ 1 incremento
  - ▶ 1 evaluación de condición

A efectos prácticos, para evaluar la complejidad, nos da igual que se ejecuten 4 operaciones en cada iteración o que se ejecute sólo 1  $\Rightarrow O(n)$

# Cálculo de la eficiencia

- Ejemplo:

$$\begin{array}{l} \text{for}(i=0; i<n; i++) \\ \quad \text{for}(i=0; i<n; i++) \\ \quad \quad A[i][j] = 0; \end{array} \left] \sum_{i=0}^n 1 = n = n \times 1 \right] \sum_{i=0}^n n = n^2 = n \times n$$

- En muchos casos, la inicialización, la condición y el incremento son operaciones simples, cuyo tiempo es  $O(1)$ , lo que simplifica mucho el análisis
- Cuando se trata de bucles simples (todas las iteraciones son iguales), el tiempo total es el producto del número de iteraciones por el tiempo del cuerpo del bucle.

# Cálculo de la eficiencia

- Ejemplo:

```
k=0;  
while (k<n && A[k] != z)  
    k++;
```

- **¡Importante!** No olvidemos que estamos buscando una cota superior. Por ello, siempre analizaremos el peor caso
- En nuestro ejemplo, consideraremos que la condición  $A[k] \neq z$  es siempre verdadera (en la práctica, esa condición actúa como un “acortador” del bucle).

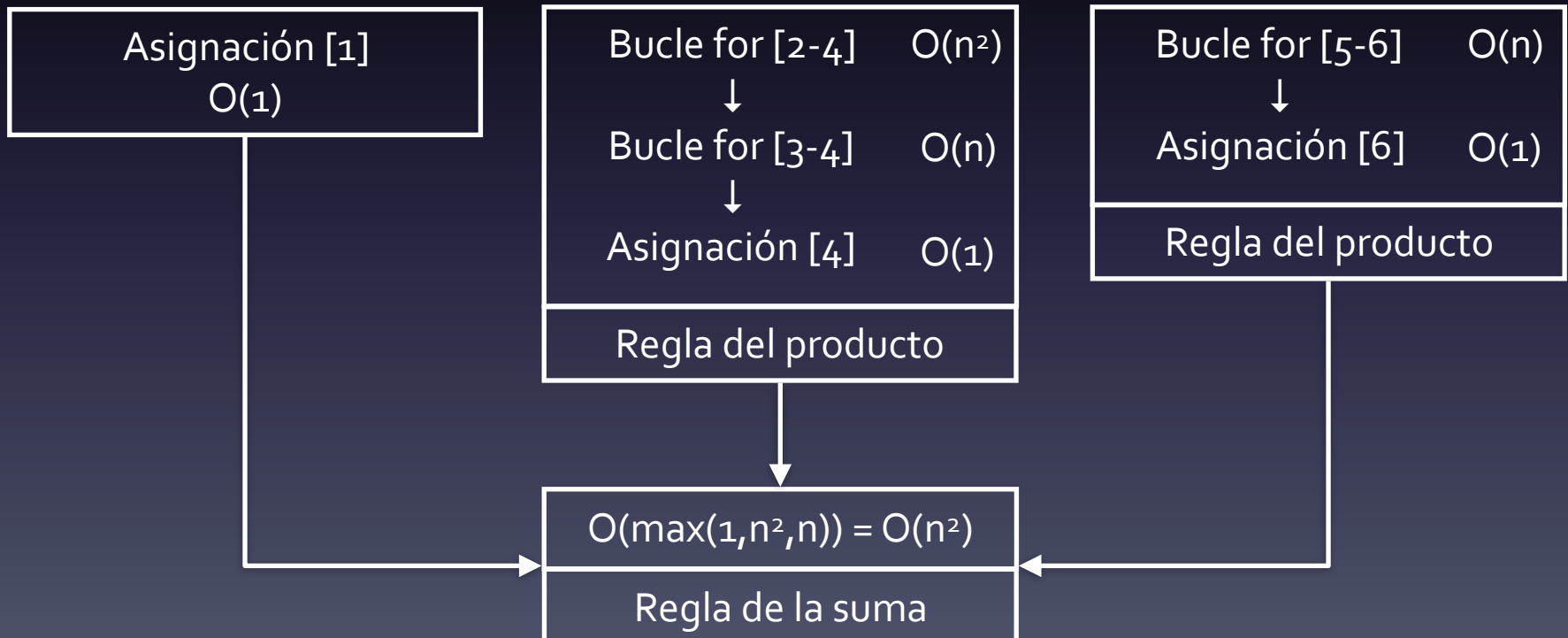


# Cálculo de la eficiencia

```
1. cin >> n; O(1)
2. for (int i=0; i<n; i++)
3.     for (int j=0; j<n; j++)
4.         A[i][j] = 0; O(1)
5. for (int k=0; k<n; k++)
6.     A[k][k] = 1; O(1)
```

Complexity analysis using curly braces:

- Lines 2-4:  $O(n)$
- Lines 3-4:  $O(n^2)$
- Lines 2-6:  $O(n^2)$



# Cálculo de la eficiencia

- Ejemplo:

```
if (A[0][0] == 0)
    for(i=0; i<n; i++)
        for(j=0; j<n; j++)
            A[i][j] = 0;
else
    for(i=0; i<n; i++)
        A[i][i] = 0;
```

$O(n)$   $O(n^2)$   $O(n^2)$

# Cálculo de la eficiencia

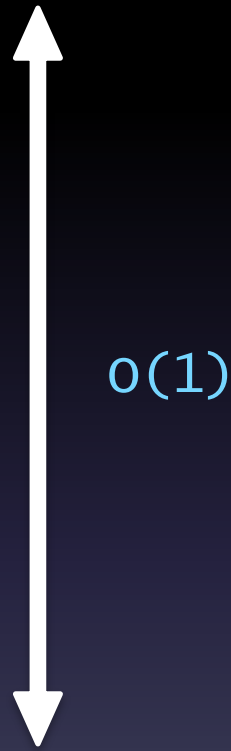
- **Funciones:** El orden de eficiencia de una función es el orden de las sentencias que la componen
- La llamada a la función y el paso de parámetros por referencia se realizan en un tiempo constante
- El paso de parámetros por valor implica una operación de copia. Debemos calcular su coste.
- Allí donde aparezca la llamada a la función (en una asignación, en la condición, inicialización o cuerpo de un bucle, etc... debe tenerse en cuenta su orden de eficiencia y contabilizarlo

# Cálculo de la eficiencia

- En particular:
  - Las asignaciones con llamadas a función deben sumar el tiempo de ejecución de cada llamada
  - En las condiciones e incrementos de bucles con llamada se deberá multiplicar el tiempo de la llamada por las iteraciones del bucle
  - La inicialización de bucles o la condición de sentencias condicionales con llamadas deben sumar el tiempo de ejecución de la llamada al tiempo total del bucle o del condicional

# Ejemplos

```
int main() {  
    double a, b, c;  
    double r1, r2;  
  
    cout << "Coeficiente de 2º grado: ";    o(1)  
    cin >> a;    o(1)  
    cout << "Coeficiente de 1º grado: ";    o(1)  
    cin >> b;    o(1)  
    cout << "Término independiente: ";    o(1)  
    cin >> c;    o(1)  
  
    if (a!=0){    o(1)    o(1)  
        r1 = (-b + sqrt(b*b-4*a*c))/2*a;    o(1)  
        r2 = (-b - sqrt(b*b-4*a*c))/2*a;    o(1)  
        cout << "Las raíces son: " << r1 << " y " << r2 << endl;    o(1)  
    }  
    else{  
        r1 = c/b;    o(1)  
        cout << "La única raíz es " << r1 << endl;    o(1)  
    }  
    return 0;  
}
```



# Ejemplos

```
int BusquedaLineal(const int v[], const int n, const int elemento){
    int i, posicion;
    bool encontrado;

    i=0;
    encontrado = false;
    while(i<n && !encontrado)
        if (v[i] == elemento){
            posicion = i;
            encontrado = true;
        }
        else
            i++;
    if (encontrado)
        return posicion;
    else
        return -1;
}
```

Complexity analysis for the worst case (element not found):

- $i=0$ :  $O(1)$
- $\text{encontrado} = \text{false}$ :  $O(1)$
- $\text{while}(i < n \ \&\& \ !\text{encontrado})$ :  $O(1)$
- Inside the while loop:
  - $\text{if}(v[i] == \text{elemento})$ :  $O(1)$
  - $\text{posicion} = i$ :  $O(1)$
  - $\text{encontrado} = \text{true}$ :  $O(1)$
  - $\text{else}$ :  $O(1)$
  - $i++$ :  $O(1)$
- $\text{if}(\text{encontrado})$ :  $O(1)$
- $\text{return posicion}$ :  $O(1)$
- $\text{else}$ :  $O(1)$
- $\text{return -1}$ :  $O(1)$

The while loop body is executed  $n$  times in the worst case. The complexity of the while loop body is  $O(1)$ . Therefore, the total complexity of the while loop is  $O(1 \times n) = O(n)$ . The overall complexity of the function is  $O(n)$ .

Debemos realizar siempre el análisis del peor caso: en nuestro ejemplo, el peor caso es que no se encuentre el elemento, lo que supone recorrer todo el vector

# Ejemplos

```
int BusquedaBinaria(const int v[], const int n, const int elemento){
    int izquierda, derecha, centro;

    izquierda=0;           o(1)
    derecha = n-1;         o(1) ] o(1)
    centro = (izquierda + derecha)/2; o(1)

    while(izquierda<=derecha && v[centro]!=elemento){
        if (v[centro] < elemento)      o(1)
            derecha = centro - 1;     o(1)
        else                          o(1)
            izquierda = centro + 1;   o(1)
        centro = (izquierda + derecha)/2;
    }

    if (izquierda > derecha)          o(1)
        return -1;                   o(1)
    else                              o(1)
        return centro;

}

Complexity analysis:
- Initialization: O(1)
- While loop: O(log2n) iterations * O(1) per iteration = O(log2n)
- Final check: O(1)
```

**\*Análisis del peor caso: si no se encuentra el elemento, vamos dividiendo el espacio de búsqueda a la mitad en cada iteración**

# Algunas “recetas” matemáticas útiles

- Propiedades de logaritmos:

$$\log_b(xy) = \log_b x + \log_b y$$

$$\log_b x^a = a \cdot \log_b x$$

$$\log_b(x/y) = \log_b x - \log_b y$$

$$\log_b x = \frac{\log_a x}{\log_a b}$$

- Propiedades de exponenciales:

$$a^{b+c} = a^b \cdot a^c$$

$$b = a^{\log_a b}$$

$$a^{b \cdot c} = \left(a^b\right)^c$$

$$b^c = a^{c \cdot \log_a b}$$

$$a^{b-c} = a^b / a^c$$



# Algunas “recetas” matemáticas útiles

- Sumatorias: definición general

$$\sum_{i=s}^t f(i) = f(s) + f(s+1) + f(s+2) + \cdots + f(t)$$

- Progresión geométrica:  $f(i) = a^i$

Dado un entero  $n \geq 0$  y un número real  $a$  ( $0 < a \neq 1$ )

$$\sum_{i=0}^n a^i = 1 + a + a^2 + \cdots + a^n = \frac{a^{n+1} - 1}{a - 1}$$

- Progresión aritmética:  $f(i) = i$

Dado un entero  $n > 1$

$$\sum_{i=0}^n i = 1 + 2 + 3 + \cdots + n = n \cdot \frac{n+1}{2}$$

- Suma de cuadrados:

$$\forall n \geq 1 \quad \sum_{i=1}^n i^2 = 1 + 4 + 9 + \cdots = \frac{n(n+1)(2n+1)}{6}$$

# Ejemplos

```
void intercambiar(int &a, int &b){
    int aux = a;    o(1)
    a = b;          o(1)
    b = aux;        o(1)
}
```

o(1)

```
void ordenacionSeleccion (int v[], int n){
    int minimo, aux;

    for(int i=0; i<n; i++){
        minimo = i;
        for(int j=i+1; j<n; j++){
            if (v[j]<v[minimo])
                minimo = j;
        }
        intercambiar(v[minimo],v[i]);
    }
}
```

o(1)  
o(1)  
o(1) o(n-i) o(n-i) o(n<sup>2</sup>)

$$\sum_{i=0}^{n-1} (n-i) = \sum_{i=0}^{n-1} n - \sum_{i=0}^{n-1} i = n^2 - n \cdot \frac{n+1}{2} \Rightarrow O(n^2)$$

# Ejemplos

```
for(int i=1; i<n; i++)           o(n-1)
  for(int j=i+1; j<n+1; j++)     o(n-i)
    for(int k=1; k<j+1; k++)     o(j)
      // Sentencia o(1)
```

→ j iteraciones

$$\sum_{j=i+1}^n j = \frac{n+(i+1)}{2} \cdot (n-i) = \frac{1}{2}(n^2 + n - i^2 + i)$$

$$\frac{1}{2} \sum_{i=1}^{n-1} (n^2 + n - i^2 + i) = \frac{1}{2} \left( \sum_{i=1}^{n-1} n^2 + \sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i^2 + \sum_{i=1}^{n-1} i \right)$$

$$\sum_{i=1}^{n-1} n^2 = n^2(n-1)$$

$$\sum_{i=1}^{n-1} n = n(n-1)$$

$$\sum_{i=1}^{n-1} i^2 = \frac{(n-1)n(2n-1)}{6}$$

$$\sum_{i=1}^{n-1} i = \frac{1+(n-1)}{2} \cdot n - 1 = \frac{n(n-1)}{2}$$

$$\left. \begin{array}{l} \sum_{i=1}^{n-1} n^2 = n^2(n-1) \\ \sum_{i=1}^{n-1} n = n(n-1) \\ \sum_{i=1}^{n-1} i^2 = \frac{(n-1)n(2n-1)}{6} \\ \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \end{array} \right\} n^2(n-1) + n(n-1) + \frac{(n-1)n(2n-1)}{6} + \frac{n(n-1)}{2}$$

**O(n<sup>3</sup>)**

# Ejemplos

```
for(int i=1; i<n+1; i++)  
  if(i%2==0){  
    for(int j=i; j<n+1; j++)  
      x++;  
    for(int j=1; j<i+1; j++)  
      y++;  
  }
```

Diagram illustrating the complexity analysis of the nested loops:

- The inner loop `for(int j=i; j<n+1; j++)` has a complexity of  $O(1)$  per iteration, and it runs  $O(n-i+1)$  times for each  $i$ .
- The inner loop `for(int j=1; j<i+1; j++)` has a complexity of  $O(1)$  per iteration, and it runs  $O(i)$  times for each  $i$ .
- The total complexity for each  $i$  is  $O(n-i+1) + O(i) = O(n+1)$ .

Puesto que se ejecuta  $n/2$  veces, tenemos

$$O(n/2(n-1)) \supset O(n^2)$$

# Ejemplos

```
void mi_funcion(const int n){  
    int x, contador;  
    contador = 0;           o(1)  
    x = 2;                  o(1)  
    while (x<=n){           o(1)  
        x = 2 * x;          o(1)  
        contador++;         o(1)  
    }  
    cout << contador << endl; o(1)  
}
```

Diagram illustrating the complexity analysis of the code:

- The `while` loop body (lines 5-7) is grouped by a bracket labeled  $O(?)$ .
- The entire function body (lines 2-8) is grouped by a larger bracket labeled  $O(?)$ .

# Ejemplos

## Producto Matricial:

```
for(int i=0; i<n; i++)  
  for(int j=0; j<n; j++){  
    C[i][j] = 0;  
    for( int k=0; k<n; k++)  
      C[i][j] += A[i][k]*B[k][j];  
  }  
}
```

$O(1)$   $O(n)$   $O(n)$   $O(n^2)$   $O(n^3)$

# Ejemplos

$$\left. \begin{array}{l} M_1 \rightarrow 10 \times 20 \\ M_2 \rightarrow 20 \times 50 \\ M_3 \rightarrow 50 \times 1 \\ M_4 \rightarrow 1 \times 100 \end{array} \right\} M_1 \times M_2 \times M_3 \times M_4$$

$$\left. \begin{array}{ccccccc} M_1 & \times & M_2 & \times & M_3 & \times & M_4 \\ (10 \times 20) & & (20 \times 50) & & (50 \times 1) & & (1 \times 100) \\ \hline & & M_{12} [10000] & & & & \\ & & 10 \times 50 & & & & \\ & & \hline & & & & M_{123} [500] & & \\ & & & & 10 \times 1 & & \\ & & & & \hline & & & & & & M_{1234} [1000] & \\ & & & & & & 10 \times 100 & \end{array} \right\} [11500]$$

# Ejemplos

$$\left. \begin{array}{l} M_1 \rightarrow 10 \times 20 \\ M_2 \rightarrow 20 \times 50 \\ M_3 \rightarrow 50 \times 1 \\ M_4 \rightarrow 1 \times 100 \end{array} \right\} M_1 \times M_2 \times M_3 \times M_4$$

$$\begin{array}{ccccccc} M_1 & \times & M_2 & \times & M_3 & \times & M_4 \\ (10 \times 20) & & (20 \times 50) & & (50 \times 1) & & (1 \times 100) \\ & & \underbrace{\hspace{2cm}} & & & & \\ & & M_{23}[1000] & & & & \\ & & 20 \times 1 & & & & \\ \underbrace{\hspace{2cm}} & & & & & & \\ M_{123}[200] & & & & & & \\ 10 \times 1 & & & & & & \\ & & & & \underbrace{\hspace{2cm}} & & \\ & & & & M_{1234}[1000] & & \\ & & & & 10 \times 100 & & \\ & & & & & & \underbrace{\hspace{2cm}} [2200] \end{array}$$