ANÁLISIS DE LA EFICIENCIA DE ALGORITMOS

- Comparación de algoritmos: **EFICIENCIA**. Expresada en función del tamaño de las instancias.

- **Eficiencia** { - Empírica / - Teórica }

- **Principio de invarianza**: Dos implementaciones de un mismo algoritmo no diferirán más que en una constante multiplicativa.

- Importancia de la eficiencia.

Sea un problema que se puede resolver empleando dos algoritmos.

- *Algoritmo* 1: $T_1(n) = 10^{-4} \times 2^n$ s

n	10	20	30	38
T(n)	0'1 s	2 min.	> 1 día	1 año

- *Algoritmo* 2: $T_2(n) = 10^{-2} \times n^3$ s

n	200	1000
T(n)	1 día	1 año

- La mejora del hardware no es suficiente.

- Análisis asintótico.

- La elección de una estructura de datos adecuada es fundamental para una buena resolución del problema.

Jerarquía de costes computacionales: consecuencias prácticas (II)

Tiempos de ejecución en una máquina que ejecuta 10^9 *pasos por segundo (~ 1 GHz), en función del coste del algoritmo y del tamaño del problema* n:

Talla	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n
10	3.322 ns	10 ns	33 ns	100 ns	1 μs	1 μs
20	4.322 ns	20 ns	86 ns	400 ns	8 μs	1 ms
30	4.907 ns	30 ns	147 ns	900 ns	27 μs	1 s
40	5.322 ns	40 ns	213 ns	2 μs	64 μs	18.3 min
50	5.644 ns	50 ns	282 ns	3 μs	125 μs	13 días
100	6.644 ns	100 ns	664 ns	10 μs	1 ms	$40 \cdot 10^{12}$ años
1000	10 ns	1 μs	10 μs	1 ms	1 s	
10000	13 ns	10 μs	133 μs	100 ms	16.7 min	
100000	17 ns	100 μs	2 ms	10 s	11.6 días	
1000000	20 ns	1 ms	20 ms	16.7 min	31.7 años	

Tamaño del problema

* El tiempo de ejecución de un algoritmo no es fijo sino que depende del tamaño del problema
  - Ejemplos
    - Asignación de tareas
    - Ordenación

* Describiremos el tiempo de ejecución por medio de una función f que depende del tamaño del problema n (TALLA)

$$\boxed{f(n)}$$

$\downarrow$

> Para comparar la eficiencia de 2 algoritmos compararemos las funciones que describen su tiempo de ejecución

$$f : \mathbb{N} \longrightarrow \mathbb{R}_0^+$$
$$n \longrightarrow f(n)$$

Algoritmos vs implementaciones

Hay que hacer la distinción entre:

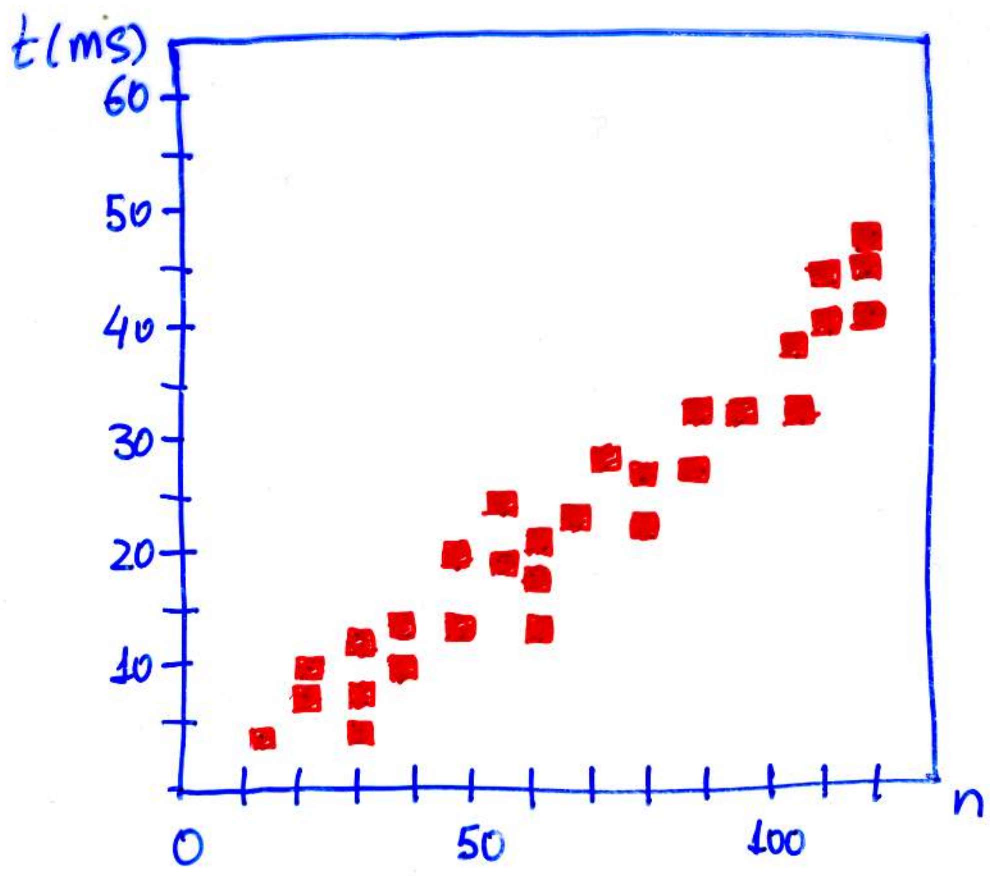
Algoritmos: Conjunto finito de pasos que nos llevan a resolver un problema

Implementaciones: Realización de un algoritmo en un determinado lenguaje de programación

* NUESTRO ANALISIS SE REFERIRÁ A ALGORITMOS Y NO A IMPLEMENTACIONES

MIDIENDO EL TIEMPO DE EJECUCION

- ¿Cómo deberíamos medir el tiempo de ejecución de un algoritmo?
- Estudio experimental:
  - Escribir un programa que implementa el algoritmo
  - Ejecutar el programa con conjuntos de datos de diferente tamaño y composición.
  - Usar algún método para tener una adecuada medida del tiempo de ejecución
  - Los datos resultantes podrían parecerse a algo como esto:



MAS ALLA DE LOS ESTUDIOS EXPERIMENTALES

- Los estudios experimentales tienen varias limitaciones:
  - Es necesario implementar y testear el algoritmo para determinar su tiempo de ejecución.
  - Los experimentos deben hacerse sobre un conjunto limitado de entradas y puede no ser indicativo del tiempo de ejecución en otras entradas no incluidas en el experimento.
  - Para comparar dos algoritmos, deberían usarse los mismos entornos hardware y software.

- Por tanto

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

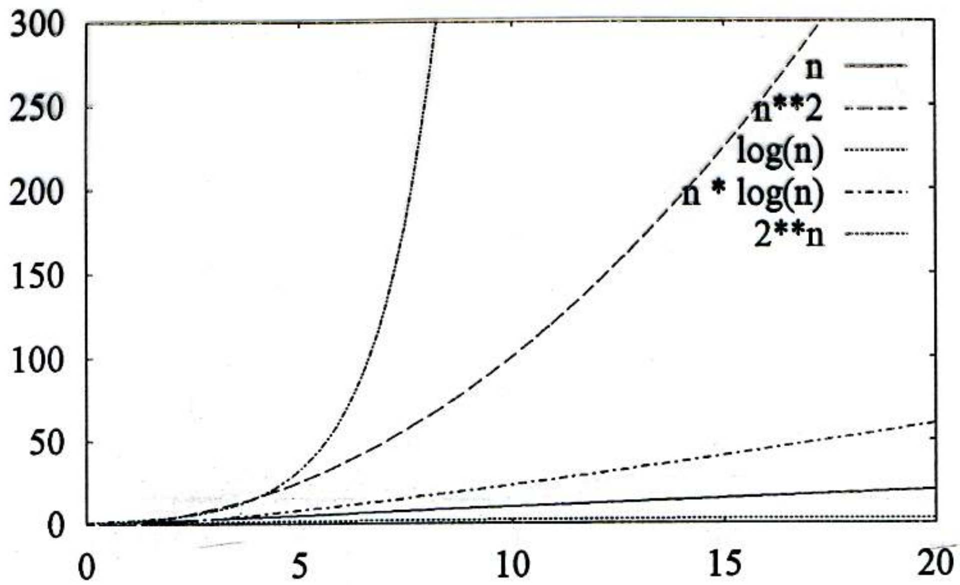
Eficiencia teórica VS experimental o empírica

FAMILIAS DE ÓRDENES DE EFICIENCIA

- **Orden de Eficiencia**: Decimos que un algoritmo tiene un tiempo de ejecución de orden T(n), para una función dada T, si existe una constante positiva c y una implementación del algoritmo capaz de resolver cada ejemplo del problema en un tiempo acotado superiormente por cT(n), donde n es el tamaño del problema considerado.

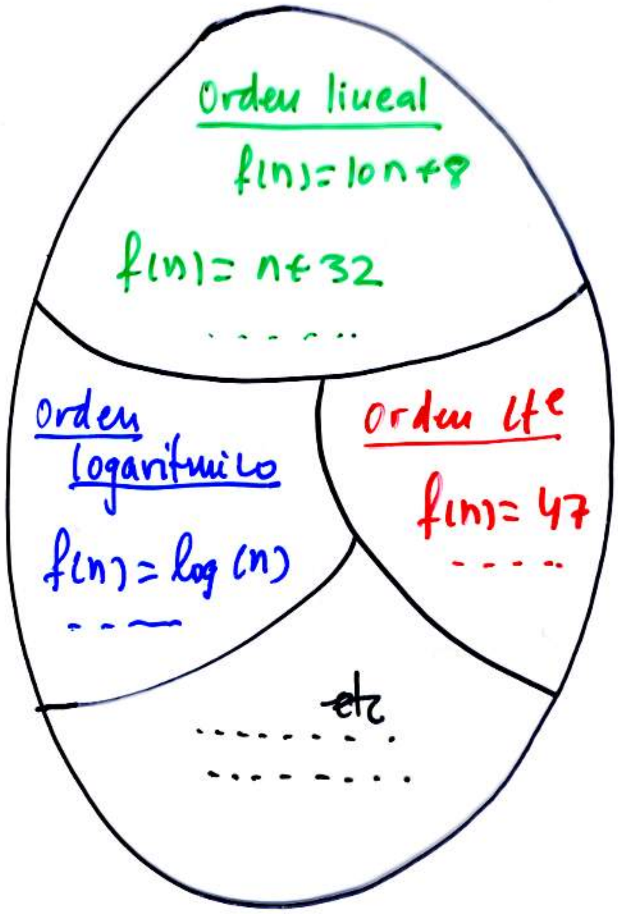
- Algunos de los órdenes más habituales son:
  - n: lineal
  - n^2: cuadrático
  - n^k: (con k natural) polinómico.
  - log$n$: logarítmico
  - c^n: exponencial

- Comparación gráfica de algunos órdenes de eficiencia



- Eficiencia en uso de recursos: memoria, ...

- Buscar un compromiso entre tiempo y recursos.



Así:

Cuando queramos estudiar el tiempo de ejecución de un algoritmo <u>independientemente de la implementación</u>, tendremos que calcular la <u>clase de equivalencia</u> a la que corresponde la función del tiempo de ejecución, es decir, determinar <u>el ORDEN DE EFICIENCIA</u>

Así:

La decisión sobre la mayor o menor eficiencia de un algoritmo dependerá del comportamiento del orden de eficiencia cuando n crece, es decir, cuando $\boxed{n \to \infty}$, de forma que comparamos PERFILES DE CRECIMIENTO

"Un algoritmo es mas eficiente que otro si su perfil de crecimiento crece mas lentamente"

Comparación de ordenes de eficiencia

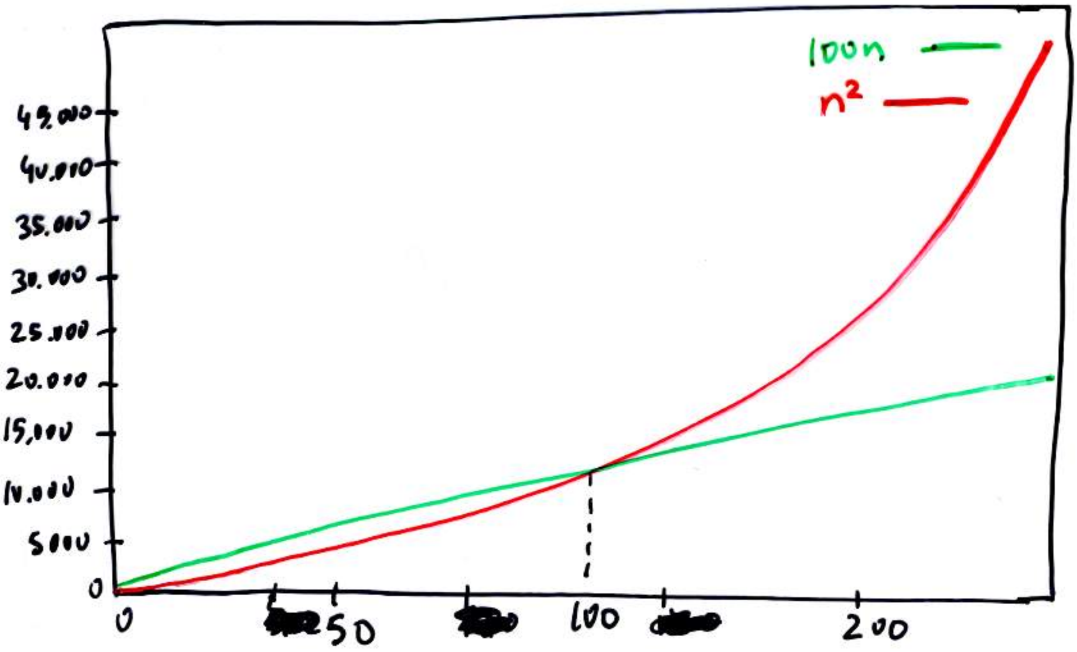
2 precondiciones; para decir que un orden es mayor que otro:

* El resultado no puede depender de lo que ocurre en un intervalo finito de valores de la función
* El resultado no puede depender de la elección de las funciones concretas que representen las correspondientes clases

Definición de orden entre funciones

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

$$\boxed{\exists\, c \in \mathbb{R}^+,\ n_0 \in \mathbb{N} \ /\ \forall n \geq n_0 \quad g(n) \leq c f(n)}$$



$g(n) = 100n$ es menor que $f(n) = n^2$ porque:

$\exists\ c \in \mathbb{R}^+ \quad [c = 100]$

$n_0 \in \mathbb{N} \quad [n_0 = 1]$ tales que

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

$\downarrow$

$n = 1$	$100 \leq 100$
$n = 2$	$200 \leq 400$
$n = 3$	$300 \leq 900$

$\vdots$

Importante: $\boxed{\text{EL ORDEN ES SOLO PARCIAL}}$

$\downarrow$

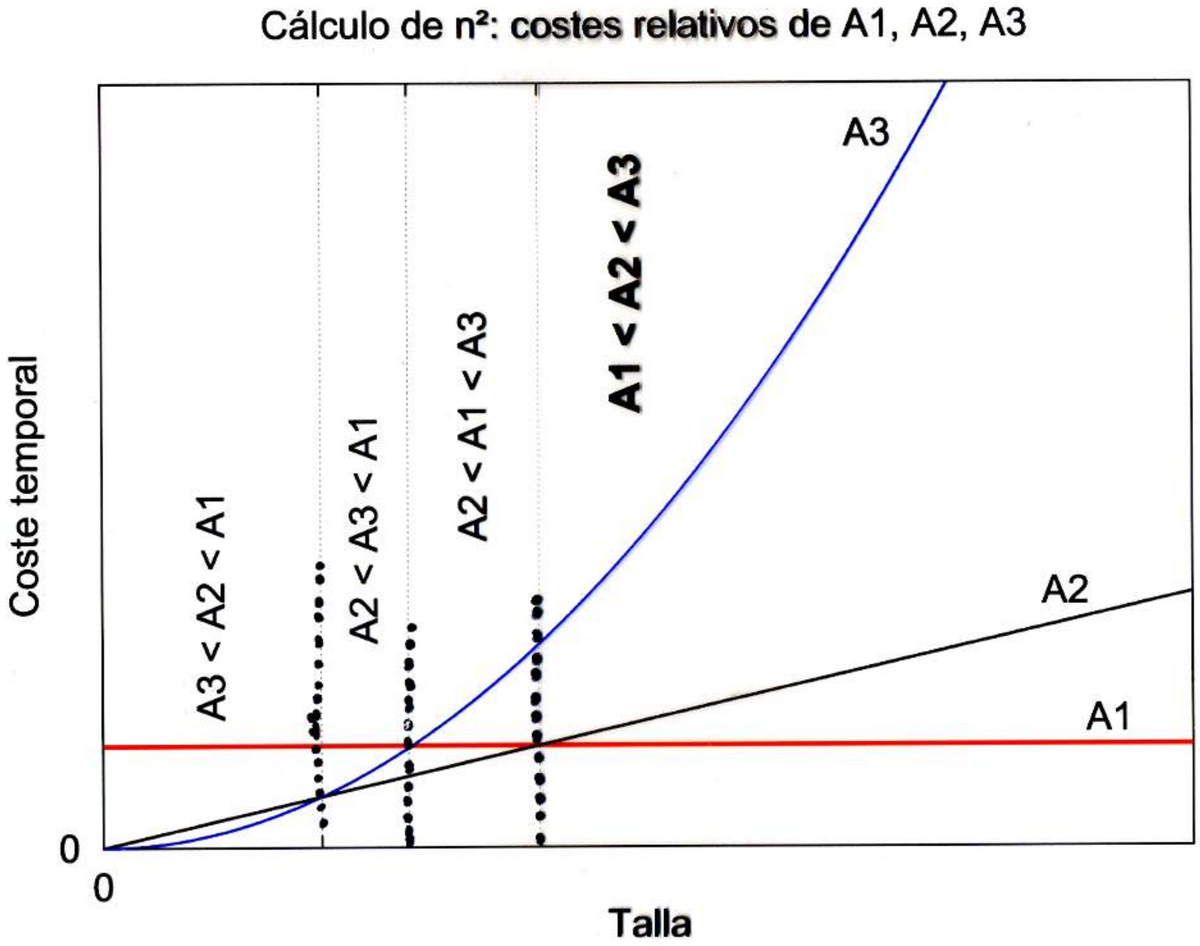
Podemos encontrar 2 funciones que no puedan ser ordenadas

Ej:

$$f(n) = \begin{cases} n^2 & \text{si } n \text{ es par} \\ n^4 & \text{si } n \text{ es impar} \end{cases} \qquad g(n) = n^3$$

Coste asintótico

En general, tendríamos un comportamiento relativo de $A1$, $A2$, $A3$ tal como:



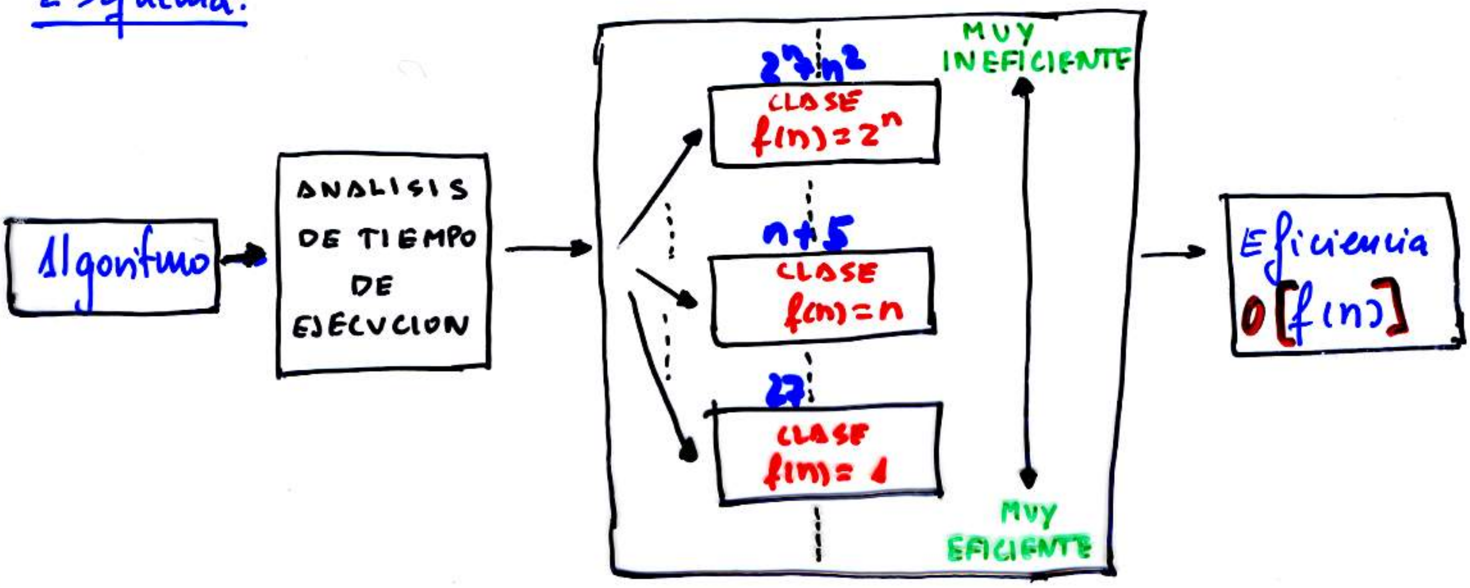
Una buena caracterización computacional de un programa:

Dependencia funcional del coste con la talla – ¡para tallas grandes!

NOTACIONES O, Ω, Θ

Objetivo: Indicar de forma clara y sin ambigüedad el grado de eficiencia que hemos obtenido en el análisis de un algoritmo.

Esquema:



2 PASOS:

1. ANALISIS DEL TIEMPO DE EJECUCION
   * Estudiar la función que indica el tiempo de ejecución necesario para cada n
2. ANALISIS DE EFICIENCIA
   * Clasificar el tiempo de ejecución en una familia de funciones

Así, si tenemos ordenadas las familias, podremos comparar los algoritmos

NOTACION ASINTOTICA

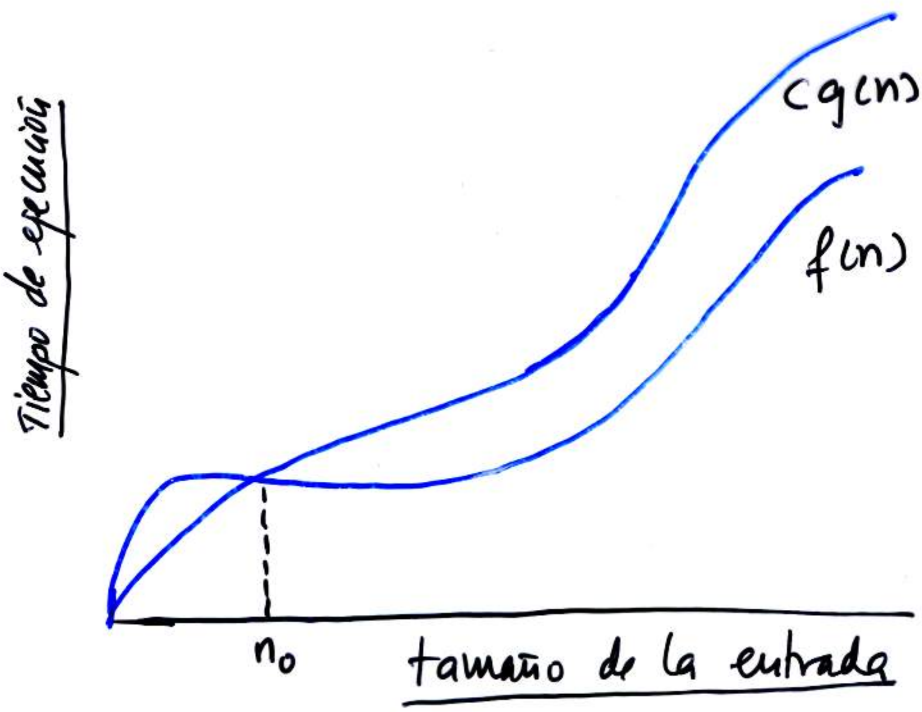
- Meta: Simplificar el análisis eliminando información no necesaria.

$$3n^2 + 3 \simeq n^2$$

- LA NOTACION "O GRANDE"
  - Dadas 2 funciones $f(n)$ y $g(n)$ decimos que: $f(n)$ es $O(g(n))$ si y solo si

$$f(n) \leq c\, g(n) \text{ para } n \geq n_0$$

  - c y n_0 son constantes, $f(n)$ y $g(n)$ son funciones sobre enteros no negativos



$(n+1)^2$ es $O(n^2)$ $[n_0 = 1,\ c = 4]$

$(3n^3 + 2n^2)$ es $O(n^3)$ $[n_0 = 0,\ c = 5]$

3^n no es $O(2^n)$

Aunque según la definición se verifica p. ej. que

$7n+3$ es $O(n^5)$

La idea es buscar una aproximación al menor orden posible

$\boxed{\text{Una regla simple}}$

↓

Eliminar los términos de orden menor y los factores ctes

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 $O(n^2)$
- $5n+3$ es $O(n)$
- $n/2 + \log_2 n$ es $O(n)$
- 250 es $O(1)$

Casos especiales de algoritmos:

- logarítmico : $O(\log_2 n)$
- lineal : $O(n)$
- cuadrático : $O(n^2)$
- polinomial : $O(n^k)$ $k>1$
- exponencial : $O(a^n)$ $n>1$

En resumen:

* La notación "O" caracteriza el tiempo de ejecución de una forma que es independiente de la implementación específica (ordenador, lenguaje de programación, estilo de programación ...)

- El signo $\leq$ implícito en la notación O recoge la idea de eficiencia en el caso peor. Hay que tener en cuenta que:

$$\begin{cases} 2n^2+3 \text{ es } O(n^2) \text{ y} \\ 2n^2+3 \text{ es } O(n^3) \end{cases}$$

Ambas son correctas, pero es preferible la primera porque da una caracterización más fina.

- Se usa la notación "O" para expresar el número de operaciones primitivas que se ejecutan como una función del tamaño de la entrada

- Los tiempos de ejecución en una notación 'O' se pueden comparar:
  - Un algoritmo que se ejecuta en $O(n)$ es mejor que uno que se ejecuta en $O(n^2)$
  - $O(\log_2 n)$ es mejor que $O(n)$

Jerarquía de funciones:

$$\log_2 n \ll n \ll n^2 \ll n^3 \ll 2^n$$

Notación asintótica: jerarquía de costes computacionales

Algunas relaciones entre órdenes usuales:

Exponenciales

$\cdots$ Polinómicas

Superlineales $\cdots$

Lineales

$$O(1) \subset O(\log n) \subset O(\sqrt{n}) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset O(n^3) \subset O(2^n) \subset O(n^n)$$

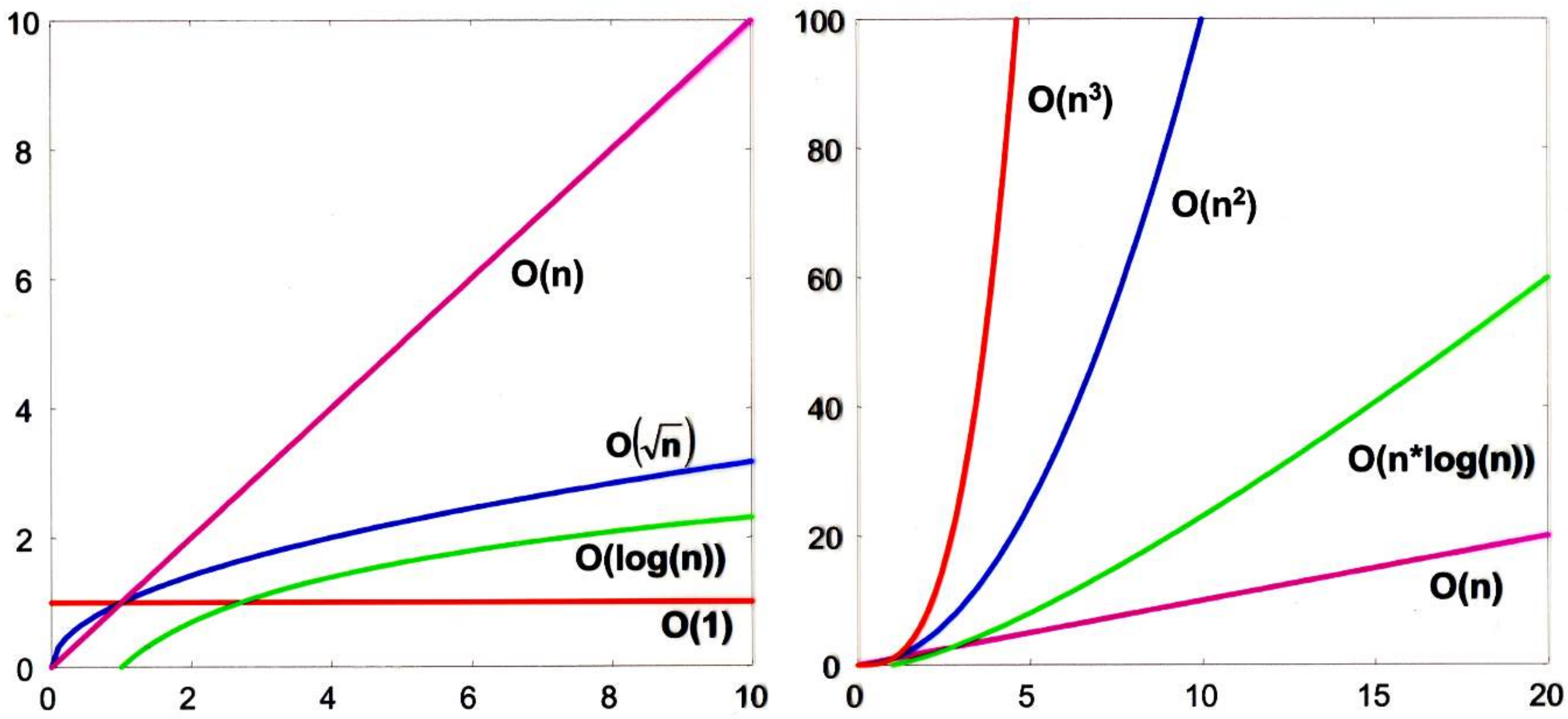
Lineales

$\cdots$ Sublineales

Logarítmicas

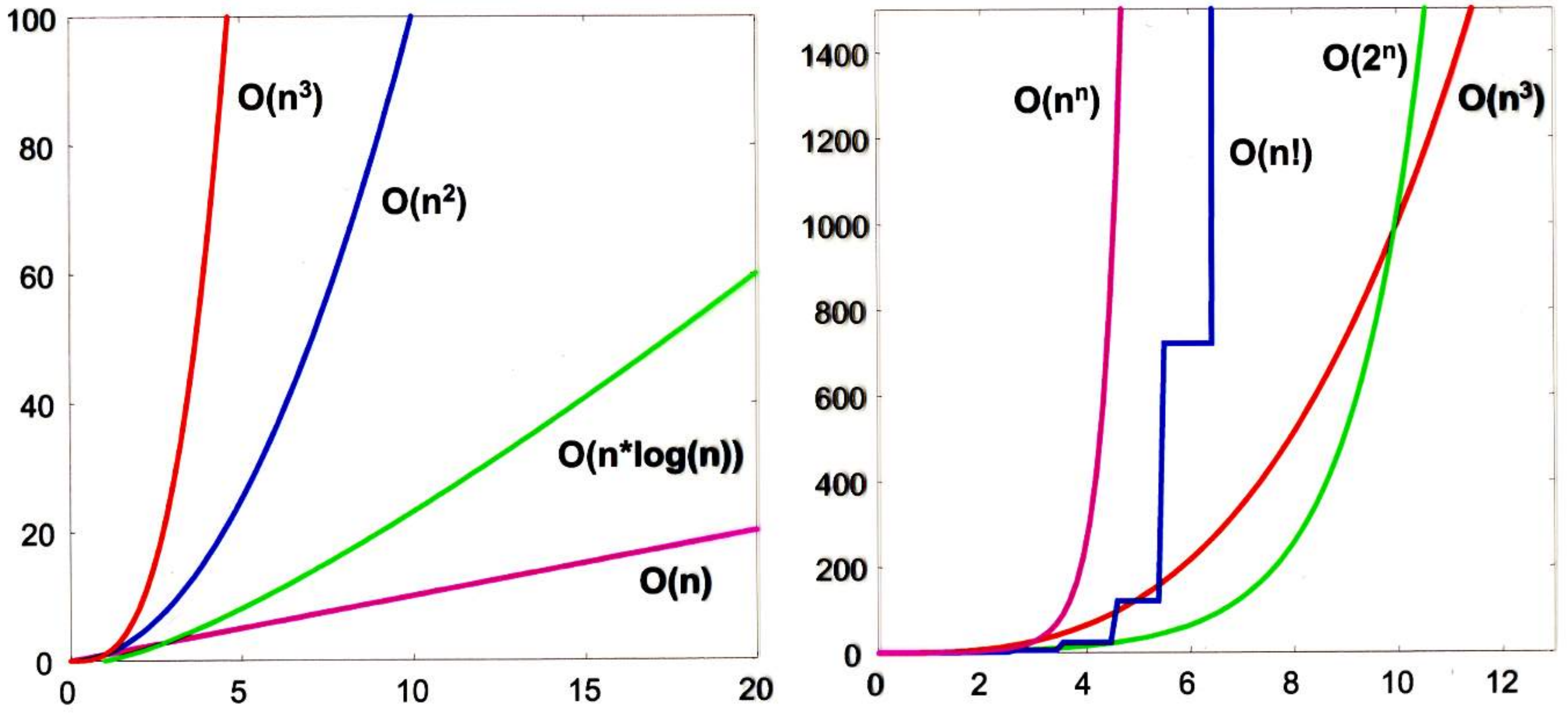
Constantes

Notación asintótica: jerarquía de costes computacionales



sublineales, lineales y superlineales

Notación asintótica: jerarquía de costes computacionales



superlineales, polinómicas y exponenciales

- Cuidado con los factores constantes muy grandes:

  Un algoritmo con tiempo de ejecución $1.000.000\, n$ aun es $O(n)$ pero podría ser menos eficiente sobre nuestros datos que uno con tiempo de ejecución $2n^2$ que es $O(n^2)$

- Algunas reglas simples:

  Transitividad

  $$\left.\begin{array}{l} f(n) = O(g(n)) \\ g(n) = O(h(n)) \end{array}\right\} \Rightarrow f(n) \text{ es } O(h(n))$$

  Polinomios

  $$a_0 + a_1 n + \cdots + a_d n^d \text{ es } O(n^d)$$

  Jerarquía de funciones

  $$n + \log_2 n \text{ es } O(n) \;;\; 2^n + n^3 \text{ es } O(2^n)$$

  La base de los logaritmos puede ignorarse

  $$\log_a n \text{ es } O(\log_b n)$$

  Las potencias dentro de los logaritmos pueden ignorarse

  $$\log(n^2) \text{ es } O(\log n)$$

  La base y potencias en los exponentes no pueden ignorarse

  $$3^n \text{ no es } O(2^n)$$

  $$a^{(n^2)} \text{ no es } O(a^n)$$

A menudo el coste NO es (sólo) función de la talla

En los ejemplos vistos hasta ahora, todas las "instancias" de una talla dada tenían el mismo coste computacional. Pero esto *no* es siempre así. En el siguiente ejemplo, ¿cuál es el coste de la función `busca(n)`?:

```
#include <stdio.h>
#define MAXN 1000000

int busca(int *v, int n, int x) /* busca x en v[0...n-1] */
{
    int i;
    for (i=0; i<n; i++)
        if (v[i] == x)
            return i;
    return -1;
}
```

A menudo el coste NO es (sólo) función de la talla

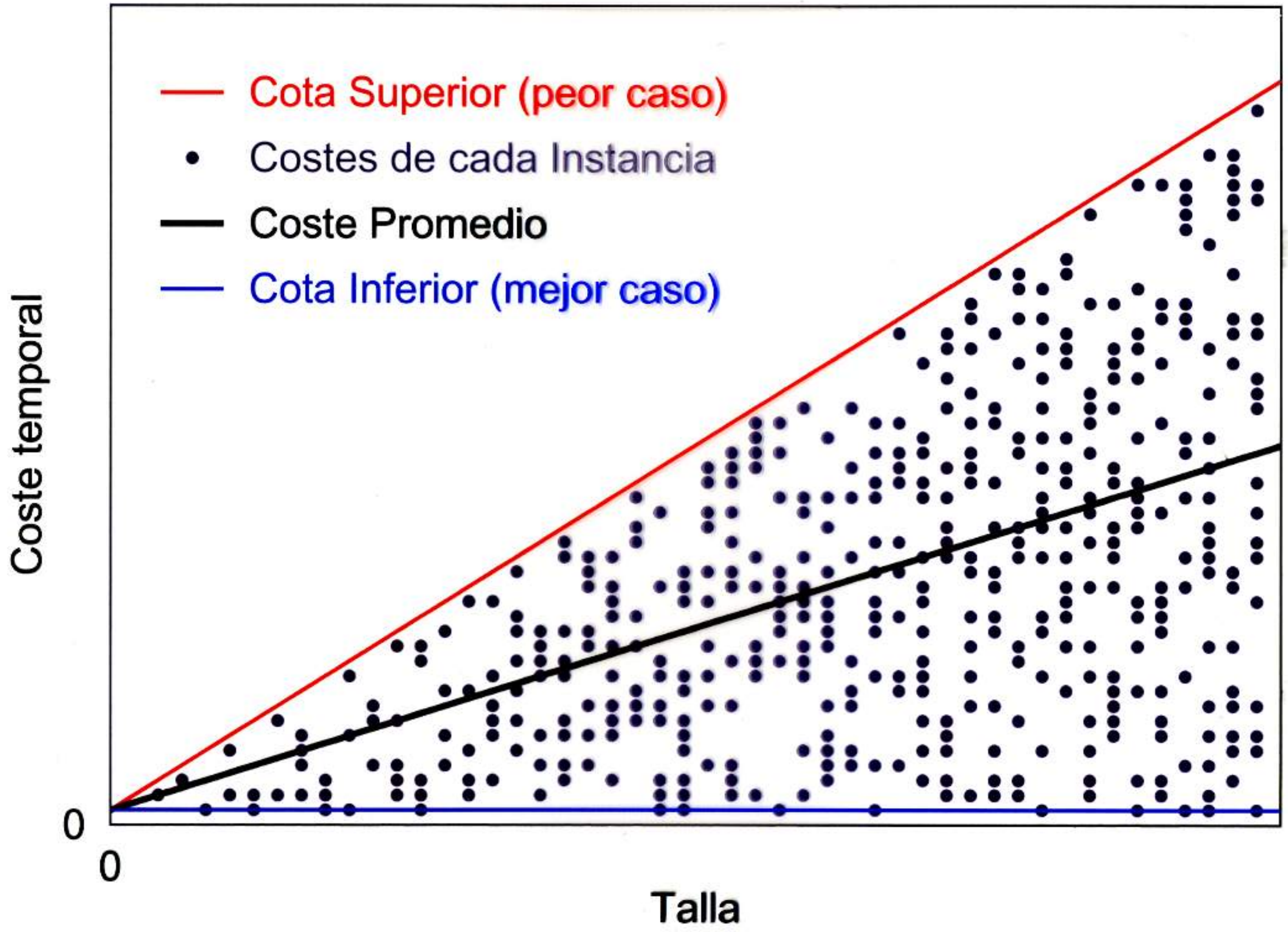
```
int main() /* busca.c */
{
    int i, x, aux;
    int v[MAXN], n = 0;     /* Vector donde buscar y su talla */

    printf("Teclear datos, fin con ^D)\n");
    while ((n < MAXN) && (scanf("%d", &aux) != EOF)) { /* lee v[] */
        v[n] = aux;
        n++;
    }
    printf("Dato a buscar: ");
    while (scanf("%d", &x) == 1) {
        i = busca(v, n, x);
        printf("Posición de %d: %d\n", x, i);
        printf("Dato a buscar: ");
    }
    return 0;
}
```

PASO: comparación en if; *Talla* $= n$; $T_{busca}(n) = ???$

¡Depende del contenido del vector y del valor concreto del elemento a buscar!

Extremos del coste: casos mejor, peor y promedio

Número de PASOS requeridos por 'busca'



$$T_{busca}^{b}(n) = 1 \quad T_{busca}^{w}(n) = n \quad T_{busca}^{m}(n) = n/2$$

ELECCION DEL MEJOR ALGORITMO

Aunque la elección de un algoritmo se basa en su eficiencia en la práctica debemos considerar varios factores:

* Tamaño de los problemas que nuestro software va a resolver
* Requisitos de tiempo y espacio de nuestro sistema
* Complejidad de implementación y mantenimiento de los algoritmos

Ejemplo

Algoritmo 1 con $t_1(n) = 100n$ → lineal $[\Theta(n)]$

Algoritmo 2 con $t_2(n) = n^2/5$ → cuadrático $[\Theta(n^2)]$

En la práctica el segundo algoritmo puede ser mas recomendable que el primero si:

- El tamaño de los problemas a resolver no va a pasar de 100 [Constante multiplicativa !!]

- El algoritmo 1 requiere una cantidad de espacio muy superior. [p. ej. que requiera una cantidad de espacio cuadrática respecto al tamaño, mientras que el segundo requiera una cantidad cte] si en nuestra aplicación es muy importante el espacio escogeremos el Algoritmo 2.

- El algoritmo 1 requiere un coste de implementación y mantenimiento muy altos respecto al segundo.