

# Análisis y diseño de algoritmos

## 2. Eficiencia

José Luis Verdú Mas, Jose Oncina, Mikel L. Forcada

Dep. Lenguajes y Sistemas Informáticos  
Universidad de Alicante

30-01-2018 (389)



- Proporcionar la capacidad para analizar con rigor la eficiencia de los algoritmos
  - Distinguir los conceptos de eficiencia en tiempo y en espacio
  - Entender y saber aplicar criterios asintóticos a los conceptos de eficiencia
  - Saber calcular la complejidad temporal o espacial de un algoritmo
  - Saber comparar, en cuanto a su eficiencia, distintas soluciones algorítmicas a un mismo problema



- 1 Noción de Complejidad
- 2 Cotas de complejidad
- 3 Cálculo de Complejidades
  - Algoritmos iterativos
  - Algoritmos recursivos



- 1 Noción de Complejidad
- 2 Cotas de complejidad
- 3 Cálculo de Complejidades
  - Algoritmos iterativos
  - Algoritmos recursivos



# ¿Qué es un algoritmo?

## Definición (Algoritmo)

Un algoritmo es una serie finita de instrucciones no ambiguas que expresa un método de resolución de un problema

Importante:

- la **máquina** sobre la que se define el algoritmo debe estar bien definida
- los **recursos** (usualmente tiempo y memoria) necesarios para cada paso elemental deben estar acotados
- El algoritmo debe **terminar** en un número **finito** de pasos



# Noción de complejidad

## Definición (Complejidad algorítmica)

Es una medida de cómo **crecen** los **recursos** que necesita un algoritmo para resolver un problema cuando el **tamaño** del problema crece.

Los recursos mas usuales son:

**Tiempo:** Complejidad temporal

**Memoria:** Complejidad espacial

Se suele expresar en función de la dificultad *a priori* del problema:

**Tamaño del problema:** lo que ocupa su representación

**Parámetro representativo:** *i.e.* la dimensión de una matriz



# ¿Cuál es el tamaño de un problema?

## Definición (Tamaño de un problema o instancia)

Número de bits que se necesita para codificar una instancia

Problema	tamaño
Sumar uno a un entero (binario de 32 bits)	
Decir cuál es el mayor de 2 números	
Ordenar un vector de $n$ enteros	
Multiplicar dos matrices de enteros de $m \times n$ y $n \times \ell$	

# ¿Cuál es el tamaño de un problema?

## Definición (Tamaño de un problema o instancia)

Número de bits que se necesita para codificar una instancia

Problema	tamaño
Sumar uno a un entero (binario de 32 bits)	32
Decir cuál es el mayor de 2 números	
Ordenar un vector de $n$ enteros	
Multiplicar dos matrices de enteros de $m \times n$ y $n \times \ell$	



# ¿Cuál es el tamaño de un problema?

## Definición (Tamaño de un problema o instancia)

Número de bits que se necesita para codificar una instancia

Problema	tamaño
Sumar uno a un entero (binario de 32 bits)	32
Decir cuál es el mayor de 2 números	$2 \cdot 32$
Ordenar un vector de $n$ enteros	
Multiplicar dos matrices de enteros de $m \times n$ y $n \times \ell$	



# ¿Cuál es el tamaño de un problema?

## Definición (Tamaño de un problema o instancia)

Número de bits que se necesita para codificar una instancia

Problema	tamaño
Sumar uno a un entero (binario de 32 bits)	32
Decir cuál es el mayor de 2 números	$2 \cdot 32$
Ordenar un vector de $n$ enteros	$32n$
Multiplicar dos matrices de enteros de $m \times n$ y $n \times \ell$	

# ¿Cuál es el tamaño de un problema?

## Definición (Tamaño de un problema o instancia)

Número de bits que se necesita para codificar una instancia

Problema	tamaño
Sumar uno a un entero (binario de 32 bits)	32
Decir cuál es el mayor de 2 números	$2 \cdot 32$
Ordenar un vector de $n$ enteros	$32n$
Multiplicar dos matrices de enteros de $m \times n$ y $n \times \ell$	$32(mn + n\ell)$



# ¿Cuál es el tamaño de un problema?

## Definición (Tamaño de un problema o instancia)

Número de bits que se necesita para codificar una instancia

Problema	tamaño
Sumar uno a un entero (binario de 32 bits)	32
Decir cuál es el mayor de 2 números	$2 \cdot 32$
Ordenar un vector de $n$ enteros	$32n$
Multiplicar dos matrices de enteros de $m \times n$ y $n \times \ell$	$32(mn + n\ell)$

- Usualmente se omite el tamaño de enteros, reales, punteros, etc. si se asume que su tamaño está acotado

# ¿Cuál es el tamaño de un problema?

## Definición (Tamaño de un problema o instancia)

Número de bits que se necesita para codificar una instancia

Problema	tamaño
Sumar uno a un entero (binario de 32 bits)	32
Decir cuál es el mayor de 2 números	$2 \cdot 32$
Ordenar un vector de $n$ enteros	$32n$
Multiplicar dos matrices de enteros de $m \times n$ y $n \times \ell$	$32(mn + n\ell)$

- Usualmente se omite el tamaño de enteros, reales, punteros, etc. si se asume que su tamaño está acotado
- ¿Cuántos bits se necesitan para codificar un entero  $n$  arbitrariamente grande?



# Atención:

La complejidad puede depender de cómo se codifique el problema

## Ejemplo

Sumar uno a un entero arbitrariamente grande

- Complejidad **constante** si el entero se codifica en base uno
- Complejidad **lineal** si el entero se codifica en base dos

Normalmente se prohíben:

- codificaciones en base uno
- codificaciones no compactas



# El tiempo de ejecución

El tiempo de ejecución de un algoritmo depende de:

## Factores externos

- La máquina en la que se va a ejecutar
- El compilador
- Los datos de entrada suministrados en cada ejecución

## Factores internos

- El número de instrucciones que ejecuta el algoritmo y su duración



# ¿Cómo estudiamos el tiempo de ejecución?

## Definición (Análisis empírico o *a posteriori*)

Ejecutar el algoritmo para distintos valores de entrada y **cronometrar** el tiempo de ejecución

- ▲ Es una medida del comportamiento del algoritmo en su entorno
- ▼ El resultado depende de los factores externos e internos

## Definición (Análisis teórico o *a priori*)

Obtener una función que represente el tiempo de ejecución (en operaciones elementales) del algoritmo para cualquier valor de entrada

- ▲ El resultado depende sólo de los factores internos
- ▲ No es necesario implementar y ejecutar los algoritmos
- ▼ No obtiene una medida real del comportamiento del algoritmo en el entorno de aplicación





# Tiempo de ejecución de un algoritmo

## Definición (Operaciones elementales)

Son aquellas operaciones que realiza el ordenador en un tiempo acotado por una constante

## Ejemplo (Operaciones elementales)

- Operaciones aritméticas básicas
- Asignaciones a variables de tipo predefinido por el compilador
- Los saltos (llamadas a funciones, retorno desde ellos ...)
- Comparaciones lógicas
- Acceso a estructuras indexadas básicas (vectores o matrices)



# Tiempo de ejecución de un algoritmo

Para simplificar, se suele considerar que el coste temporal de las operaciones elementales es unitario

## Definición (Tiempo de ejecución de un algoritmo)

Una función ( $T(n)$ ) que mide el número de operaciones elementales que realiza el algoritmo para un tamaño de problema  $n$



# Ejemplo: Suma de los elementos de un vector

## Ejemplo (sintaxis de la STL)

```
1 int sumar( const vector<int> &v){  
2     int s = 0;  
3  
4     for(int i = 0; i < v.size(); i++)  
5         s += v[i];  
6  
7     return s;  
8 }
```

Si estudiamos el bucle ( $n = v.size()$ ):

$n$	asign.	comp.	inc.	total
0	1	1	0	2
1	1	2	1	4
2	1	3	2	6
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$n$	1	$n + 1$	$n$	$2 + 2n$

La complejidad del algoritmo será:

$$T(n) = \underbrace{1}_{\text{primera asignación}} + \underbrace{2 + 2n}_{\text{bucle}} + \underbrace{n}_{\text{interior del bucle}} = 3 + 3n$$



# Ejercicio: Traspuesta de una matriz cuadrada

## Traspuesta de una matriz $d \times d$

(Sintaxis de la librería `armadillo`)

```
1 void traspuesta( mat& A){ // supongo que A.n_rows == A.n_cols
2     for( int i = 1; i < A.n_rows; i++ )
3         for( int j = 0; j < i ; j++ )
4             swap( A(i,j), A(j,i) );
5 }
```

Como la complejidad del bucle interior es:  $2 + 3i$  veces

$$T_d(d) = \underbrace{2(d-1) + 2}_{\text{bucle exterior}} + \underbrace{\sum_{i=1}^{d-1} (2 + 3i)}_{\text{interior}} = \dots = O(d^2)$$

Si queremos la complejidad con respecto al tamaño del problema ( $s = d^2$ ):

$$T_s(s) = T_d(d) = O(d^2) = O(s)$$



# Ejercicio: Producto de dos matrices cuadradas

## Producto de dos matrices $d \times d$

(Sintaxis de la librería `armadillo`)

```
1 mat producto( const mat &A, const mat &B ){
2     mat R(A.n_rows, B.n_cols);
3     for( int i = 0; i < A.n_rows; i++ )
4         for( int j = 0; j < B.n_cols; j++ ) {
5             double acc = 0.0;
6             for( int k = 0; k < A.n_cols; k++ )
7                 acc += A(i,k) * B(k,j);
8             R(i,j) = acc;
9         }
10 }
11 return R;
12 }
```

- La complejidad de las líneas 6-7 es  $O(d)$
- La complejidad de las líneas 4-9 es  $O(d) + d \cdot O(d) = O(d^2)$
- La complejidad de las líneas 3-10 es  $O(d) + d \cdot O(d^2) = O(d^3)$

La complejidad del algoritmo será:  $T_d(d) = O(d^3)$



# Ejercicio: Producto de dos matrices cuadradas

¿Cual es la complejidad con respecto al tamaño?

El tamaño del problema es  $s = 2d^2$  por lo que  $d = \sqrt{s/2}$

$$T_s(s) = T_d(d) = O(d^3) = O(\sqrt{s/2}^3) = O(s^{3/2})$$

¿Cual es la complejidad espacial?

- En la complejidad espacial no se tiene en cuenta lo que ocupa la codificación del problema.
- Solo se tiene en cuenta lo que es imputable al algoritmo.

$$T_d(d) = d^2$$

$$T_s(s) = T_d(\sqrt{s/2}) = O(s)$$



- Dado un vector de enteros  $v$  y el entero  $z$ 
  - Devuelve el primer índice  $i$  tal que  $v[i] == z$
  - Devuelve  $-1$  en caso de no encontrarlo

## Búsqueda de un elemento

```
1 int buscar( const vector<int> &v, int z ){  
2     for( int i = 0; i < v.size(); i++ )  
3         if( v[i] == z )  
4             return i;  
5     return -1;  
6 }
```



# Problema

- No podemos contar el número de pasos porque para diferentes entradas de un mismo tamaño de problema se obtienen diferentes complejidades
- En el ejemplo de la transparencia anterior:

v	z	Instrucciones
(1, 0, 2, 4)	1	3
(1, 0, 2, 4)	0	6
(1, 0, 2, 4)	2	9
(1, 0, 2, 4)	4	12
(1, 0, 2, 4)	5	14

- ¿Qué podemos hacer?
  - Acotar el coste mediante dos funciones que expresen respectivamente, el **coste máximo** y el **coste mínimo** del algoritmo (cotas de complejidad)





1 Noción de Complejidad

2 Cotas de complejidad

3 Cálculo de Complejidades

- Algoritmos iterativos
- Algoritmos recursivos



- Cuando aparecen diferentes casos para una misma talla  $n$ , se introducen las siguientes medidas de la **complejidad**
  - Caso peor: **cota superior** del algoritmo  $\rightarrow C_s(n)$
  - Caso mejor: **cota inferior** del algoritmo  $\rightarrow C_i(n)$
  - Caso promedio: **coste promedio**  $\rightarrow C_m(n)$
- Todas son funciones del **tamaño** del problema
- El coste promedio es difícil de evaluar a **priori**
  - Es necesario conocer la **distribución de probabilidad** de la entrada
  - ¡No es la media de la cota inferior y de la cota superior!

## Buscar elemento

```
1 int buscar( const vector<int> &v, int z ) {  
2     for( int i = 0; i < v.size(); i++ )  
3         if( v[i] == z )  
4             return i;  
5     return -1;  
6 }  
7
```

- En este caso el tamaño del problema es  $n = v.size()$

	Mejor caso	Peor caso
	$1 + 1 + 1$	$1 + 3n + 1$
Suma	3	$3n + 2$

## Cotas:

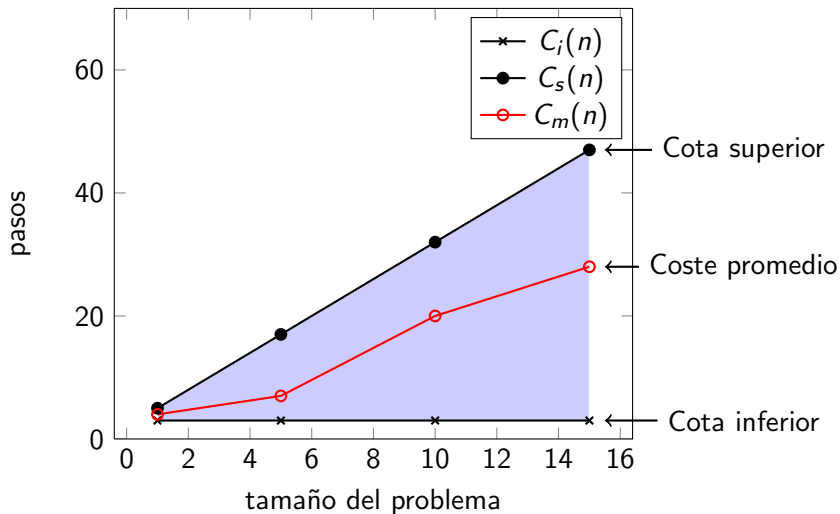
$$C_s(n) = 3n + 2 = O(n)$$

$$C_i(n) = 3 = O(1)$$



# Cotas superior e inferior

- Coste de la función buscar



- El estudio de la complejidad resulta realmente interesante **para tamaños grandes de problema** por varios motivos:
  - Las diferencias “reales” en tiempo de ejecución de algoritmos con diferente coste para tamaños pequeños del problema no suelen ser muy significativas
  - Es lógico invertir tiempo en el desarrollo de un buen algoritmo sólo si se prevé que éste realizará un gran volumen de operaciones
- Al estudio de la complejidad para tamaños grandes de problema se le denomina **análisis asintótico**
  - Permite clasificar las funciones de complejidad de forma que podamos compararlas entre si fácilmente
  - Para ello, se definen clases de equivalencia que engloban a las funciones que “crecen de la misma forma”.
- Se emplea la notación asintótica



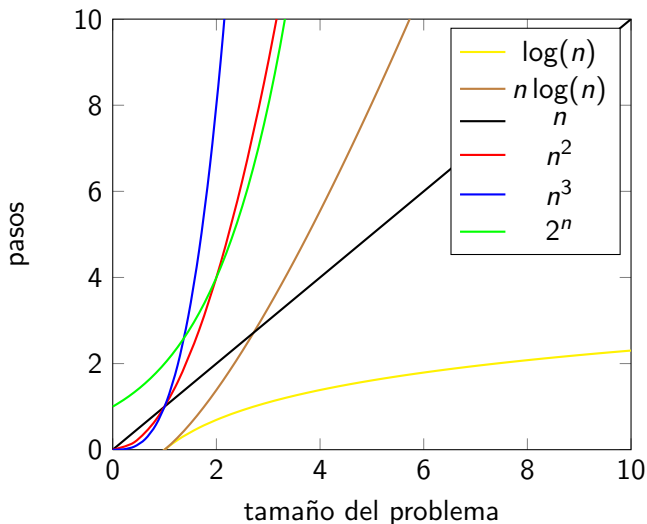
## Notación asintótica:

- Notación matemática utilizada para representar la complejidad cuando el tamaño de problema ( $n$ ) crece ( $n \rightarrow \infty$ )
- Se definen tres tipos de notación:
  - Notación  $O$  (ómicron mayúscula o *big omicron*)  $\Rightarrow$  cota superior
  - Notación  $\Omega$  (omega mayúscula o *big omega*)  $\Rightarrow$  cota inferior
  - Notación  $\Theta$  (zeta mayúscula o *big theta*)  $\Rightarrow$  coste exacto



# ¿Para qué sirven?

- Permite agrupar en clases funciones con el mismo crecimiento



- Sea  $f : \mathbb{N} \rightarrow \mathbb{R}^+$ ; se define el conjunto  $O(f)$  como el conjunto de funciones acotadas superiormente por un múltiplo de  $f$  :

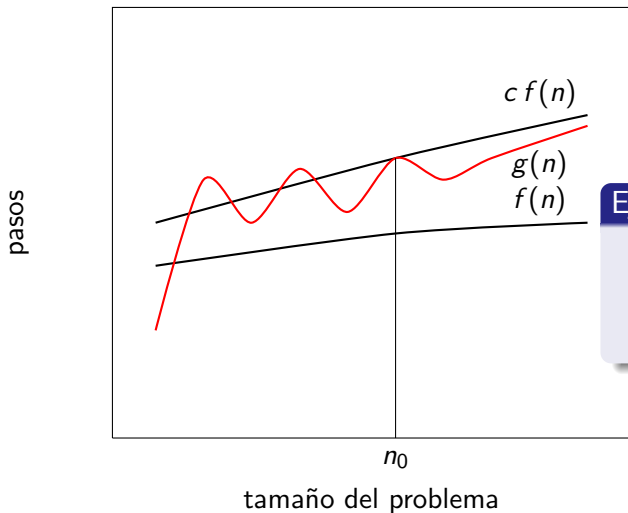
$$O(f) = \{g : \mathbb{N} \rightarrow \mathbb{R}^+ \mid \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, g(n) \leq cf(n)\}$$

- Dada una función  $t : \mathbb{N} \rightarrow \mathbb{R}^+$  se dice que  $t \in O(f)$  si existe un múltiplo de  $f$  que es cota superior de  $t$  para valores grandes de  $n$





# Cota superior. Notación $O$



## Ejemplos:

- ¿ $3n + 1 \in O(n)$ ?
- ¿ $3n^2 + 1 \in O(n)$ ?
- ¿ $3n^2 + 2 \in O(n^2)$ ?

$$f \in O(f)$$

$$f \in O(g) \Rightarrow O(f) \subseteq O(g)$$

$$O(f) = O(g) \Leftrightarrow f \in O(g) \wedge g \in O(f)$$

$$f \in O(g) \wedge g \in O(h) \Rightarrow f \in O(h)$$

$$f \in O(g) \wedge f \in O(h) \Rightarrow f \in O(\min\{g, h\})$$

$$f_1 \in O(g_1) \wedge f_2 \in O(g_2) \Rightarrow f_1 + f_2 \in O(\max\{g_1, g_2\})$$

$$f_1 \in O(g_1) \wedge f_2 \in O(g_2) \Rightarrow f_1 + f_2 \in O(g_1 + g_2)$$

$$f_1 \in O(g_1) \wedge f_2 \in O(g_2) \Rightarrow f_1 f_2 \in O(g_1 g_2)$$

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f \in O(g)$$

$$f(n) = a_m n^m + \dots + a_1 n + a_0 \text{ con } a_m > 0 \Rightarrow f(n) \in O(n^m)$$

$$O(f) \subset O(g) \Rightarrow f \in O(g) \wedge g \notin O(f)$$

- Sea  $f : \mathbb{N} \rightarrow \mathbb{R}^+$ ; se define el conjunto  $\Omega(f)$  como el conjunto de funciones acotadas inferiormente por un múltiplo de  $f$ :

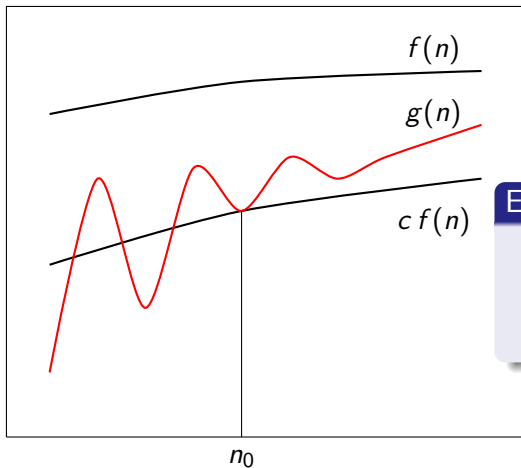
$$\Omega(f) = \{g : \mathbb{N} \rightarrow \mathbb{R}^+ \mid \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N} : \forall n \geq n_0, g(n) \geq cf(n)\}$$

- Dada una función  $t : \mathbb{N} \rightarrow \mathbb{R}^+$  se dice que  $t \in \Omega(f)$  si existe un múltiplo de  $f$  que es cota inferior de  $t$  para valores grandes de  $n$



# Cota inferior. Notación $\Omega$

pasos



tamaño del problema

## Ejemplos:

- ¿ $3n + 1 \in \Omega(n)$ ?
- ¿ $3n^2 + 1 \in \Omega(n)$ ?
- ¿ $3n^2 + 2 \in \Omega(n^2)$ ?



$$f \in \Omega(f)$$

$$f \in \Omega(g) \Rightarrow \Omega(f) \subseteq \Omega(g)$$

$$\Omega(f) = \Omega(g) \Leftrightarrow f \in \Omega(g) \wedge g \in \Omega(f)$$

$$f \in \Omega(g) \wedge g \in \Omega(h) \Rightarrow f \in \Omega(h)$$

$$f \in \Omega(g) \wedge f \in \Omega(h) \Rightarrow f \in \Omega(\max\{g, h\})$$

$$f_1 \in \Omega(g_1) \wedge f_2 \in \Omega(g_2) \Rightarrow f_1 + f_2 \in \Omega(\min(f_1, f_2))$$

$$f_1 \in \Omega(g_1) \wedge f_2 \in \Omega(g_2) \Rightarrow f_1 + f_2 \in \Omega(f_1 + f_2)$$

$$f_1 \in \Omega(g_1) \wedge f_2 \in \Omega(g_2) \Rightarrow f_1 f_2 \in \Omega(g_1 g_2)$$

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \Rightarrow g \in \Omega(f)$$

$$f(n) = a_m n^m + \dots + a_1 n + a_0 \text{ con } a_m > 0$$

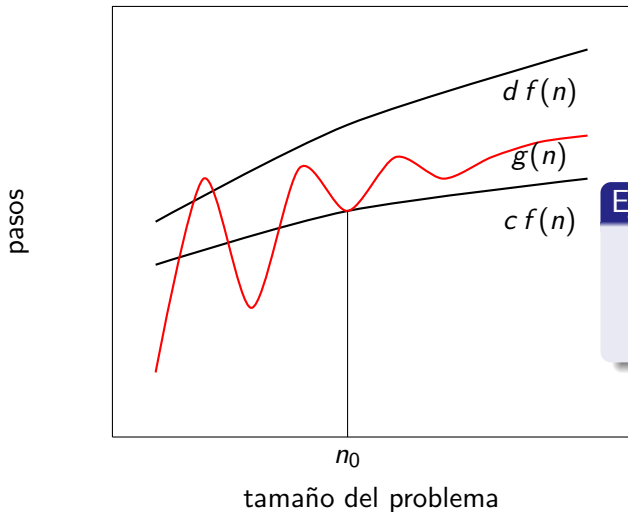
$$\Rightarrow f(n) \in \Omega(n^m)$$

- Sea  $f : \mathbb{N} \rightarrow \mathbb{R}^+$ ; se define el conjunto  $\Theta(f)$  como el conjunto de funciones acotadas superior e inferiormente por un múltiplo de  $f$ :

$$\Theta(f) = \{g : \mathbb{N} \rightarrow \mathbb{R}^+ \mid \exists c, d \in \mathbb{R}^+, \exists n_0 \in \mathbb{N} : \\ \forall n \geq n_0, cf(n) \leq g(n) \leq df(n)\}$$

- O lo que es lo mismo:  $\Theta(f) = O(f) \cap \Omega(f)$
- Dada una función  $t : \mathbb{N} \rightarrow \mathbb{R}^+$  se dice que  $t \in \Theta(f)$  si existen múltiplos de  $f$  que son a la vez cota superior y cota inferior de  $t$  para valores grandes de  $n$

# Coste exacto. Notación $\Theta$



## Ejemplos:

- ¿ $3n + 1 \in \Theta(n)$ ?
- ¿ $3n^2 + 1 \in \Theta(n)$ ?
- ¿ $3n^2 + 2 \in \Theta(n^2)$ ?

$$f \in \Theta(f)$$

$$f \in \Theta(g) \Rightarrow \Theta(g) = \Theta(f)$$

$$\Theta(f) = \Theta(g) \Leftrightarrow f \in \Theta(g) \wedge g \in \Theta(f)$$

$$f \in \Theta(g) \wedge g \in \Theta(h) \Rightarrow f \in \Theta(h)$$

$$f \in \Theta(g) \wedge f \in \Theta(h) \Rightarrow f \in \Theta(\max\{g, h\})$$

$$f_1 \in \Theta(g_1) \wedge f_2 \in \Theta(g_2) \Rightarrow f_1 + f_2 \in \Theta(f_1 + f_2)$$

$$f_1 \in \Theta(g_1) \wedge f_2 \in \Theta(g_2) \Rightarrow f_1 f_2 \in \Theta(g_1 g_2)$$

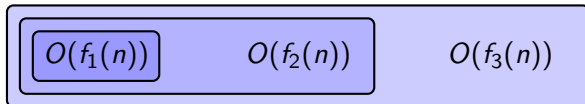
$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = k, k \neq 0, k \neq \infty \Rightarrow \Theta(f) = \Theta(g)$$

$$f(n) = a_m n^m + \dots + a_1 n + a_0 \text{ con } a_m > 0 \\ \Rightarrow f(n) \in \Theta(n^m)$$



# Jerarquías de funciones

- Los conjuntos de funciones están incluidos unos en otros generando una ordenación de las diferentes funciones. Por ejemplo, para  $O(\cdot)$ ,



- Las clases más utilizadas en la expresión de complejidades son:

$$\begin{array}{ccccccc} \underbrace{O(1)}_{\text{constantes}} & \subset & \underbrace{O(\log \log n)}_{\text{sublogarítmicas}} & \subset & \underbrace{O(\log n) \subset O(\log^{a(>1)} n)}_{\text{logarítmicas}} \\ & & \subset & \underbrace{O(\sqrt{n})}_{\text{sublineales}} & \subset & \underbrace{O(n)}_{\text{lineales}} & \subset & \underbrace{O(n \log n)}_{\text{lineal-logarítmicas}} \\ & & & & & \subset & \underbrace{O(n^2) \subset O(n^{a(>2)})}_{\text{polinómicas}} & \subset & \underbrace{O(2^n) \subset O(a(>2)^n)}_{\text{exponenciales}} & \subset & \underbrace{O(n!) \subset O(n^n)}_{\text{superexponenciales}} \end{array}$$



1 Noción de Complejidad

2 Cotas de complejidad

3 Cálculo de Complejidades

- Algoritmos iterativos
- Algoritmos recursivos



- Pasos para obtener las cotas de complejidad
  - ① Determinar la **talla** o tamaño del problema
  - ② Determinar los **casos mejor** y **peor** (instancias para las que el algoritmo tarda más o menos)
    - Para algunos algoritmos, el caso mejor y el caso peor son el mismo ya que se comportan igualmente para cualquier instancia del mismo tamaño
  - ③ Obtención de las cotas **para cada caso**
    - Algoritmos iterativos
    - Algoritmos recursivos

## Sumar elementos

```
1 int sumar( const vector<int> &v ) {  
2     int s = 0;  
3     for( int i = 0; i < v.size(); i++ )  
4         s += v[i];  
5     return s;  
6 }
```

Línea	Pasos	C. Asintótica
2	1	$\Theta(1)$
3,4	$n$	$\Theta(n)$
5	1	$\Theta(1)$
Suma	$n+2$	$\Theta(n)$

## Buscar elemento

```
1 int buscar( const vector<int> &v, int z ) {  
2     for( int i = 0; i < v.size(); i++ )  
3         if( v[i] == z )  
4             return i;  
5     return -1;  
6 }
```

Línea	Cuenta Pasos		C. Asintótica	
	Mejor caso	Peor caso	Mejor caso	Peor caso
2	1	$n$	$\Omega(1)$	$O(n)$
3	1	$n$	$\Omega(1)$	$O(n)$
4	1	0	$\Omega(1)$	–
5	0	1	–	$O(1)$
Suma	3	$2n + 1$	$\Omega(1)$	$O(n)$

$$C_s(n) = 2n + 1$$

$$C_i(n) = 3$$

$$C_s(n) \in O(n)$$

$$C_i(n) \in \Omega(1)$$



## Elemento máximo de un vector

```
1 int maximo( const vector<int> &v ) {  
2     int max = v[0];  
3     for( int i = 1; i < v.size(); i++ )  
4         if( v[i] > max )  
5             max = v[i];  
6     return max;  
7 }
```



## Búsqueda en un vector ordenado

```
1 int buscar( const vector<int> &v, int x ) {  
2     int pri = 0;  
3     int ult = v.size() - 1;  
4     while( pri < ult ) {  
5         int m = ( pri + ult ) / 2;  
6         if ( v[m] < x )  
7             pri = m + 1;  
8         else  
9             ult = m;  
10    }  
11    if( v[pri] == x )  
12        return pri;  
13    else  
14        return -1;  
15 }
```



- Dado un algoritmo recursivo:

## Búsqueda binaria

```
1 int buscar( const vector<int> &v, int pri, int ult, int x){  
2     if( pri == ult )  
3         return (v[pri] == x) ? pri : -1;  
4     int m = ( pri + ult ) / 2;  
5     if( v[m] < x )  
6         return buscar( v, m+1, ult, x );  
7     else  
8         return buscar( v, pri, m, x );  
9 }
```

- El coste depende de las llamadas recursivas, y, por tanto, debe definirse recursivamente:

$$T(n) \in \begin{cases} \Theta(1) & n = 1 \\ \Theta(1) + T(n/2) & n > 1 \end{cases} \quad (n = \text{ult} - \text{pri} + 1)$$





- Una **relación de recurrencia** es una expresión que relaciona el valor de una función  $f$  definida para un entero  $n$  con uno o más valores de la misma función para valores menores que  $n$

$$f(n) = \begin{cases} a f(F(n)) + P(n) & n > n_0 \\ P'(n) & n \leq n_0 \end{cases}$$

Donde:

- $a \in \mathbb{N}$  es una constante
- $P(n), P'(n)$  son funciones de  $n$
- $F(n) < n$  (normalmente  $n - b$  con  $b > 0$ , o  $n/b$  con  $b > 1$ )

- Las relaciones de recurrencia se usan para expresar la complejidad de un algoritmo recursivo aunque también son aplicables a los iterativos
- Si el algoritmo dispone de mejor y peor caso, puede haber una relación de recurrencia para cada caso
- La complejidad de un algoritmo se obtiene en tres pasos:
  - 1 Determinación de la talla del problema
  - 2 Obtención de las relaciones de recurrencia del algoritmo
  - 3 Resolución de las relaciones
- Para resolverlas, usaremos el método de **sustitución**:
  - Es el método más sencillo
  - Sólo para funciones lineales (sólo una vez en función de sí mismas)
  - Consiste en sustituir cada  $f(n)$  por su valor al aplicarle de nuevo la función hasta obtener un término general



# Ordenación por selección

- Ejemplo: Ordenar un vector a partir del elemento `pri`:

## Ordenación por selección (recursivo)

```
1 void ordenar( vector<int> &v, int pri) {  
2     if( pri == v.size() )  
3         return;  
4     int m = pri;  
5     for( int i = pri + 1; i < v.size(); i++ )  
6         if( v[i] < v[m] )  
7             m = i;  
8     swap( v[m], v[pri]);  
9     ordenar(v, pri + 1);  
10 }
```

- Obtener ecuación de recurrencia a partir del algoritmo:

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ \Theta(n) + T(n-1) & n > 1 \end{cases}$$

donde  $n = v.size() - pri$ .



- Resolviendo la recurrencia por sustitución

$$\begin{aligned}T(n) &= n + T(n-1) \\&= n + (n-1) + T(n-2) \\&= n + (n-1) + (n-2) + T(n-3) \\&= n + (n-1) + (n-2) + (n-3) + \cdots + 3 + 2 + T(1) \\&= n + (n-1) + (n-2) + (n-3) + \cdots + 3 + 2 + 1 \\&= \sum_{j=1}^n j = \frac{n(n+1)}{2}\end{aligned}$$

Entonces

$$T(n) \in \Theta(n^2)$$



# Algoritmo de ordenación por partición o *Quicksort*

- Elemento pivote: sirve para dividir en dos partes el vector. Su elección define variantes del algoritmo
  - Al azar
  - Primer elemento (Quicksort primer elemento)
  - Elemento central (Quicksort central)
  - Elemento mediana (Quicksort mediana)
- Pasos:
  - Elección del pivote
  - Se divide el vector en dos partes:
    - parte izquierda del pivote (elementos menores)
    - parte derecha del pivote (elementos mayores)
  - Se hacen dos llamadas recursivas. Una con cada parte del vector



# Quicksort primer elemento

## Quicksort

```
1 void quicksort( int v[], int pri, int ult ) {  
2     if( ult <= pri )  
3         return;  
4     int p = pri;  
5     int j = ult;  
6     while(p < j) {  
7         if( v[p+1] < v[p] ) {  
8             swap( v[p+1], v[p] );  
9             p++;  
10        } else {  
11            swap( v[p+1], v[j] );  
12            j--;  
13        }  
14    }  
15    quicksort(v, pri, p-1);  
16    quicksort(v, p+1, ult);  
17 }
```



- Tamaño del problema:  $n$ 
  - **Mejor caso**: subproblemas  $(n/2, n/2)$

$$T(n) \in \begin{cases} \Theta(1) & n \leq 1 \\ \Theta(n) + T(\frac{n}{2}) + T(\frac{n}{2}) & n > 1 \end{cases}$$

- **Peor caso**: subproblemas  $(0, n-1)$  o  $(n-1, 0)$

$$T(n) \in \begin{cases} \Theta(1) & n \leq 1 \\ \Theta(n) + T(0) + T(n-1) & n > 1 \end{cases}$$

- Mejor caso:

$$f(n) = n + 2T\left(\frac{n}{2}\right) \quad \text{Rec. 1}$$

$$= n + 2\left(\frac{n}{2} + 2f\left(\frac{n}{2^2}\right)\right) = 2n + 2^2T\left(\frac{n}{2^2}\right) \quad \text{Rec. 2}$$

$$= 2n + 2^2\left(\frac{n}{2^2} + 2f\left(\frac{n}{2^3}\right)\right) = 3n + 2^3T\left(\frac{n}{2^3}\right) \quad \text{Rec. 3}$$

$$= i n + 2^i T\left(\frac{n}{2^i}\right) \quad \text{Rec. } i$$

La recursión termina cuando  $n/2^i = 1$  por lo que habrá  $i = \log_2 n$  llamadas recursivas

$$= n \log_2 n + nT(1) = n \log_2 n + n$$

Por tanto,

$$T(n) \in \Omega(n \log_2 n)$$



- Peor caso:

$$T(n) = n + T(n-1) \quad \text{Rec. 1}$$

$$= n + (n-1) + T(n-2) \quad \text{Rec. 2}$$

$$= n + (n-1) + (n-2) + T(n-3) \quad \text{Rec. 3}$$

$$= n + (n-1) + (n-2) + \cdots + T(n-i) \quad \text{Rec. } i$$

La recursión termina cuando  $n - i = 1$  por lo que habrá  $i = n - 1$  llamadas recursivas

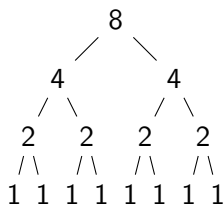
$$= n + (n-1) + (n-2) + \cdots + 3 + 2 + T(1)$$

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

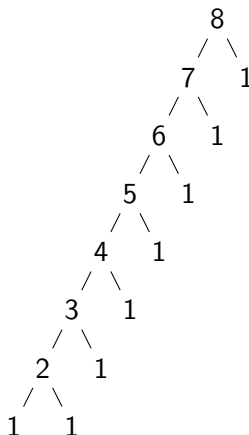
Por tanto,

$$f(n) \in O(n^2)$$

# Quicksort



Caso mejor  
 $\Omega(n \log_2 n)$



Peor caso  
 $O(n^2)$



- En la versión anterior se cumple que el caso mejor es cuando el elemento seleccionado es la mediana
- En este algoritmo estamos forzando el caso mejor
- Obtener la mediana
  - Coste menor que  $O(n \log n)$
  - Se aprovecha el recorrido para reorganizar elementos y para encontrar la mediana en la siguiente subllamada
  - Su complejidad es por tanto de  $\Theta(n \log n)$

