

## Introducción

### ¿Qué es la eficiencia de los algoritmos?

- ❖ Estudio **teórico comparativo**
- ❖ **Independiente** de la **implementación**
- ❖ **Clasificar** algoritmos en **familias**
- ❖ Comportamiento según **crece el tamaño de su entrada** → **asintótico**

### ¿Cómo medimos la eficiencia?

- ❖ *¿Qué nos interesa medir?*

- **Gasto en memoria**
- **Tiempo empleado**
- **Otros recursos**

Podemos medir la eficiencia por cualquiera de estos puntos.

Sin embargo, nos interesa medir el punto 2.

- ❖ *¿Cómo nos interesa medirlo?*

- **Caso peor** → Aquella ejecución del algoritmo en la que los datos de entrada hacen que el algoritmo vaya a emplear más tiempo en su ejecución, es decir, que estaríamos estudiando cual es la entrada más difícil de procesar por el algoritmo, o lo que es lo mismo, cuánto tarda el algoritmo en procesar esa entrada.
- **Caso medio** → Nos tendremos que apoyar en una distribución estadística de los datos de entrada y estudiamos aquel que es más probable. Estaríamos en este caso estudiando las entradas sobre las que más veces se ejecutaría el algoritmo.
- **Caso mejor** → Va a estudiar el tiempo empleado en la ejecución cuando los datos de entrada son óptimos, es decir, estaremos ante la entrada más sencilla de procesar que se pueda encontrar el algoritmo

Podemos medir cualquiera de los 3 datos.

Sin embargo, nos interesa el primer caso.

En conclusión, mediremos la eficiencia mediante su **coste** **asintótico** **temporal** **en el caso peor**.

Cuando el tamaño crezca

Tiempo empleado por el algoritmo en la ejecución

Estimaremos una cota superior midiendo cuánto tarda el algoritmo en ejecutarse ante la entrada más costosa de procesar

### ¿Cómo vamos a analizar el coste?

- ❖ **Métricas de análisis**
  - **Clasificación de funciones en familias**
- ❖ **Reglas prácticas para el análisis**
  - Programas **iterativos**
  - Programas **recursivos**

# Métricas de análisis y Órdenes de complejidad

## Notación de Landau. Cota superior

La **notación de Landau** permite “capturar” cuál es el término dominante de una función, es decir, cual es el término que más influye en el valor de dicha función. De este modo, vamos a conseguir clasificar las funciones de coste en familias según como sea su crecimiento. Por ejemplo:

Orden de  $n^2$  ←  $O(n^2) = \{g / g \text{ no crece más deprisa que } n^2\}$

Representa a todas las funciones  $g$  tales que no crecen más deprisa que  $n^2$ , es decir, su tasa de crecimiento no supera la tasa de crecimiento de  $n^2$

Formalmente:

$$O(n^2) = \{g: \mathbb{N} \rightarrow \mathbb{R}^+ \cup \{0\} | \exists n_0 \in \mathbb{N}, c \in \mathbb{R}^+ \text{ tal que } \forall n \geq n_0, g(n) \leq c \cdot n^2\}$$

Se puede definir como el conjunto de todas las funciones  $g$  que van de  $\mathbb{N}$  (naturales) en  $\mathbb{R}^+ \cup 0$  (reales positivos y 0) para las cuales existe un  $n_0 \in \mathbb{N}$  y una constante  $c \in \mathbb{R}^+$  tales que para cualquier  $n \geq n_0$ , es decir, para un tamaño del problema mayor que  $n_0$ , el valor de  $g(n) \leq c \cdot n^2$ , es decir, que a partir de  $n_0$  el crecimiento de  $g$  es, a lo sumo, el de  $n^2$ , obviando una posible constante multiplicativa gracias a esa constante  $c$

**Generalización (cota superior):**

Orden de  $f$  ←  $O(f) = \{g / g \text{ no crece más que } f\}$

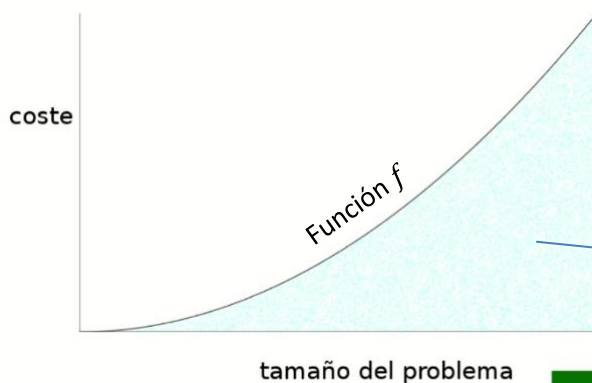
Representa a todas las funciones  $g$  cuya tasa de crecimiento no supera la tasa de crecimiento de la función  $f$

Formalmente:

$$O(f) = \{g: \mathbb{N} \rightarrow \mathbb{R}^+ \cup \{0\} | \exists n_0 \in \mathbb{N}, c \in \mathbb{R}^+ \text{ tal que } \forall n \geq n_0, g(n) \leq c \cdot f(n)\}$$

Se puede definir como el conjunto de todas las funciones  $g$  que van de  $\mathbb{N}$  (naturales) en  $\mathbb{R}^+ \cup 0$  (reales positivos y 0) para las cuales existe un  $n_0 \in \mathbb{N}$  y una constante  $c \in \mathbb{R}^+$  tales que para cualquier  $n \geq n_0$ , es decir, para un tamaño del problema mayor que  $n_0$ , el valor de  $g(n) \leq c \cdot f(n)$ , es decir, que a partir de  $n_0$  el crecimiento de  $g$  es, a lo sumo, el de la función  $f$ , obviando una posible constante multiplicativa gracias a esa constante  $c$

Es equivalente a decir que la función  $g$  esta **acotada superiormente** por la función  $f$



Orden de  $f$

### Cota inferior:

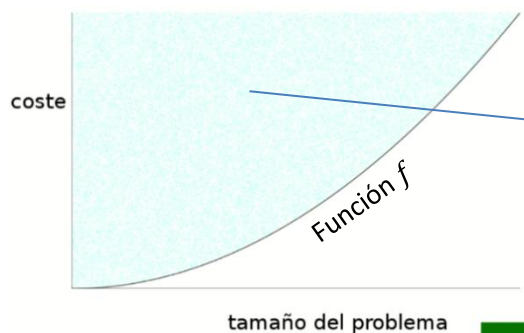
$$\Omega(f) = \{g / g \text{ crece al menos como } f\}$$

Representa a todas las funciones  $g$  que crecen al menos como la función  $f$  a partir de cierto punto

Formalmente:

$$\Omega(f) = \{g: \mathbb{N} \rightarrow \mathbb{R}^+ \cup \{0\} | \exists n_0 \in \mathbb{N}, c \in \mathbb{R}^+ \text{ tal que } \forall n \geq n_0, g(n) \geq c \cdot f(n)\}$$

Se puede definir como el conjunto de todas las funciones  $g$  que van de  $\mathbb{N}$  (naturales) en  $\mathbb{R}^+ \cup 0$  (reales positivos y 0) para las cuales existe un  $n_0 \in \mathbb{N}$  y una constante  $c \in \mathbb{R}^+$  tales que para cualquier  $n \geq n_0$ , es decir, para un tamaño del problema mayor que  $n_0$ , el valor de  $g(n) \geq c \cdot f(n)$ , es decir, que a partir de  $n_0$  el crecimiento de  $g$  es, al menos, el de la función  $f$ , obviando una posible constante multiplicativa gracias a esa constante  $c$



Es equivalente a decir que la función  $g$  está **acotada inferiormente** por la función  $f$

Orden de  $f$

### Cota exacta:

$$\Theta(f) = \{g / g \text{ crece exactamente como } f\}$$

Representa a todas las funciones  $g$  que crecen exactamente como la función  $f$  a partir de cierto punto

Formalmente:

Coloquialmente diríamos que las funciones  $g$  y  $f$  son exactamente iguales. Formalmente se dice que  $g$  está **acotada superior e inferiormente** por la función  $f$

$$\Theta(f) = O(f) \cap \Omega(f)$$

### Órdenes de complejidad

Con la medida **orden** podemos clasificar las funciones en una serie de familias según podamos acotar su crecimiento.

$$f(n) = n^2$$

$$g(n) = 3n^2$$

$$h(n) = 35n^2 + 4n + 7$$

$$i(n) = n$$

$$f, g, h \in O(n^2)$$

$$i \in O(n)$$

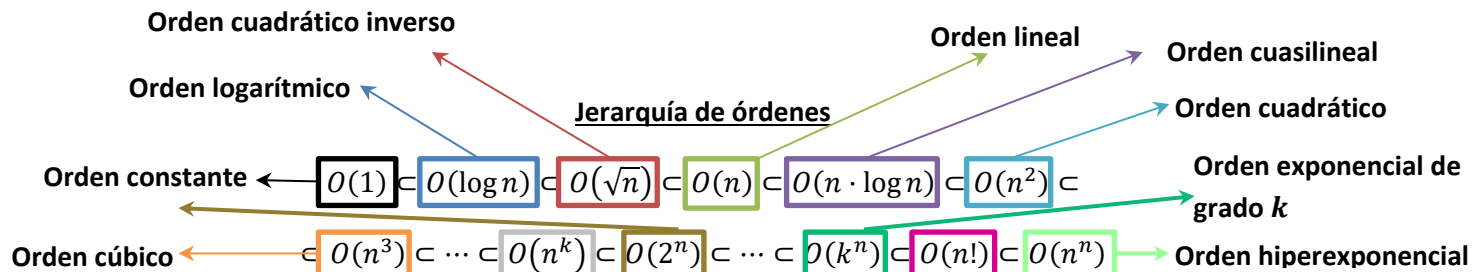
Las funciones  $f, g, h$  pertenecen al orden de  $n^2$  ( $O(n^2)$ ), ya que los términos que realmente están diciendo como crece la función, es decir, los términos que más influyen en el crecimiento de la función según crece  $n$  son del orden de  $n^2$  ( $O(n^2)$ )

La función  $i$  pertenece al orden de  $n$  ( $O(n)$ ). Sin embargo, como estamos hablando de cotas superiores, es fácil ver que el crecimiento de  $f$  es superior al crecimiento de la función  $i$ . Es decir,  $f$  también es una cota superior de  $i$ , o lo que es lo mismo, la función  $i$  también pertenece al orden de  $n^2$  ( $O(n^2)$ ).

Esto se va a cumplir para cualquier otra función que pertenezca al  $O(n)$ , lo que significa que el  $O(n)$  está incluido, y además de forma estricta, en el  $O(n^2)$

$$O(n) \subset O(n^2)$$

## Jerarquía de órdenes



**Orden polinómico de grado  $k$**

- ❖ **Orden constante** → A este orden pertenecen todas las funciones cuyo crecimiento es igual al de una función constante, es decir, todas aquellas funciones que no crecen
- ❖ **Orden logarítmico** → A él pertenecen las funciones cuyo crecimiento no supera a la función logaritmo
- ❖ **Orden cuadrático inverso o sublineal** → A este orden pertenecen las funciones cuyo crecimiento no supera a la función  $\sqrt{n}$
- ❖ **Orden lineal** → Agrupa a todas aquellas funciones cuyo crecimiento está acotado por la función  $f(n) = n$
- ❖ **Orden cuasilineal** → Contiene las funciones cuyo crecimiento está acotado por la función  $n \cdot \log n$ . Es un crecimiento algo mayor que el de la función  $n$ , de hay el nombre de cuasilineal
- ❖ **Orden cuadrático** → Agrupa a todas las funciones cuyo crecimiento no supera al de la función  $n^2$
- ❖ **Orden cúbico** → Cualquier función polinómica de cualquier grado  $k$  crece menos que el orden exponencial
- ❖ ...
- ❖ **Orden polinómico de grado  $k$**  → La tasa de crecimiento de estas funciones cada vez es mayor a medida que el grado del polinomio que define el orden crece
- ❖ **Orden exponencial** } Todas ellas, a partir de cierto punto,
- ❖ **Orden exponencial de grado  $k$**  } crecen menos que la función factorial
- ❖ **Orden factorial** → Crece menos aún que la función hiperexponencial
- ❖ **Orden hiperexponencial**

## Reglas prácticas para el cálculo del coste: Algoritmos iterativos

### Reglas para el cálculo del coste

Se nos presentan 2 reglas fundamentales:

- 1) Hay que definir cuál es el tamaño del problema
- 2) Hay que calcular el coste de cada una de las instrucciones y luego irlo acumulando


### Operaciones básicas

❖ No dependen del tamaño del problema

- Operaciones de Entrada/Salida
- Asignaciones
- Expresiones

❖ Coste constante  $\rightarrow O(1)$

```
void ordenar (T[] v) {  
    for (int i=2 ; i <= v.length ; i++) {  
        int p=i; int x=v[i]; boolean seguir=True;  
        while (p>1 and seguir) {  
            if (x<v[p-1]) {v[p]=v[p-1]}  
            else {seguir=False}  
            p=p-1;  
        }  
        v[p]=x;  
    }  
}
```

  $\rightarrow$  Operaciones básicas. Coste constante  $\rightarrow O(1)$

### Composición secuencial

❖ Varias sentencias ejecutadas secuencialmente.

- $S_1; S_2; \dots; S_n;$

¿Cómo calculamos su coste?

1) Calcular el coste de cada una:

- $c(S_1) \in O(c_{S_1}); c(S_2) \in O(c_{S_2}); \dots; c(S_n) \in O(c_{S_n})$

2) Sumar los costes:

- $O(c_{S_1}) + O(c_{S_2}) + \dots + O(c_{S_n})$

Debido a la suma de órdenes resulta:

$$O(c_{S_1}) + O(c_{S_2}) + \dots + O(c_{S_n}) = \max_i \{O(c_{S_i})\}$$

Suma de órdenes:

❖ ¿Cómo definimos la suma de órdenes?

$$O(f) + O(g)$$

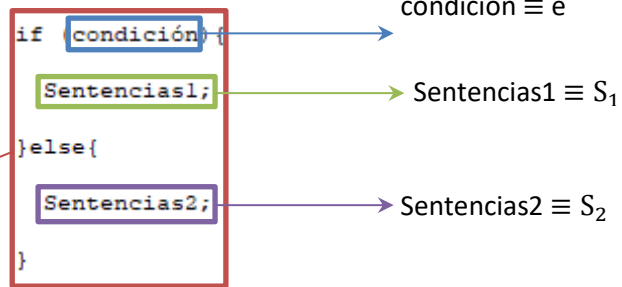
❖ La suma de órdenes es el orden de la suma:

$$O(f) + O(g) = O(f + g)$$

$$O(f) + O(g) = \text{conjunto más grande de ambos}$$

$$O(n^2) + O(n) = O(n^2 + n) = O(n^2)$$

## Sentencias condicionales

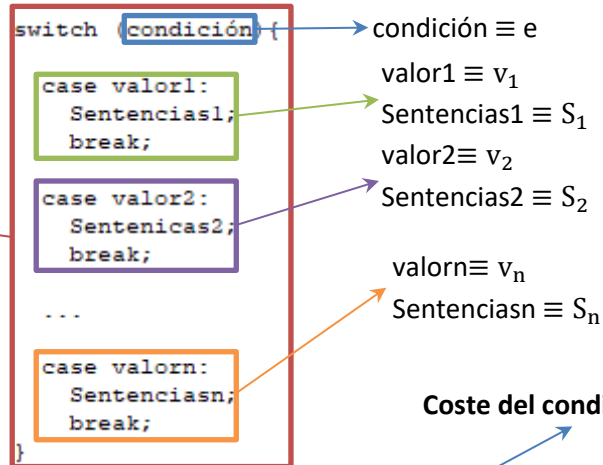


Equivale a:

- Calcular  $e$  y ejecutar  $S_1$  (si  $e$  es cierto)
- Calcular  $e$  y ejecutar  $S_2$  (si  $e$  es falso)
- ❖ Composición secuencial de  $e$ ;  $S_1$ ; ó de  $e$ ;  $S_2$ ;
- ❖ **Coste máximo** de  $e$ ;  $S_1$ ; y  $e$ ;  $S_2$ ;

**Coste del condicional if**

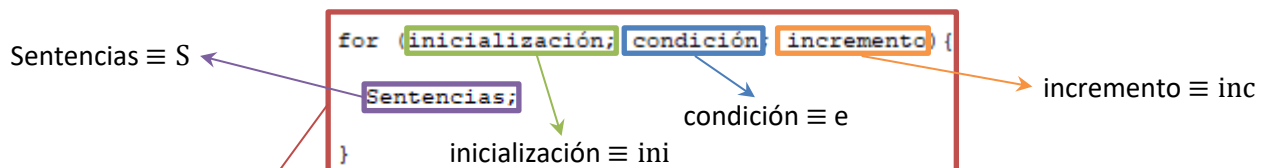
$$\max \{O(c_e), O(c_{S_1}), O(c_{S_2})\}$$



**Coste del condicional switch**

$$\max \{O(c_e), O(c_{S_1}), O(c_{S_2}), \dots, O(c_{S_n})\}$$

## Bucles



❖ Equivale a:

- Ejecutar  $ini$
- Para cada vuelta del bucle ( $v(n)$  vueltas)
  - Calcular  $e$
  - Si no es falsa  $\rightarrow$  Ejecutar  $S$ ;  $inc$ ; y volvemos al punto de calcular  $e$
  - Si es falsa  $\rightarrow$  Salir

**Coste del bucle for**

$$\max \{O(c_{ini}), O(v(n)) \cdot \max \{O(c_e), O(c_S), O(c_{inc})\}\}$$

## Producto de órdenes de complejidad:

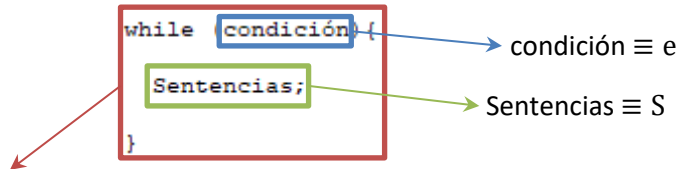
### ❖ ¿Cómo definimos el producto de órdenes?

$$O(f) \cdot O(g)$$

### ❖ El producto de órdenes es el orden del producto

$$O(f) \cdot O(g) = O(f \cdot g)$$

$$O(n) \cdot O(n) = O(n \cdot n) = O(n^2)$$

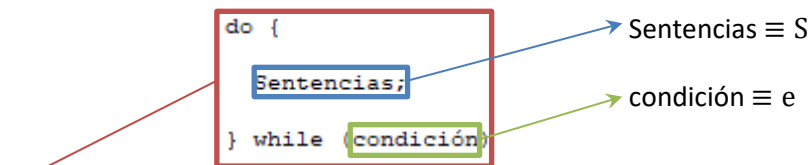


### ❖ Para cada vuelta del bucle ( $v(n)$ vueltas):

- Calcular  $e$
- Si no es falsa  $\rightarrow$  Ejecutar  $S$ ;
- Si es falsa  $\rightarrow$  Salir

**Coste del bucle while**

$$O(v(n)) \cdot \max\{O(c_e), O(c_s)\}$$



### ❖ Ejecutar $S$ ;

### ❖ Para cada vuelta del bucle ( $v(n)$ vueltas):

- Calcular  $e$
- Si no es falsa  $\rightarrow$  Ejecutar  $S$ ;
- Si es falsa  $\rightarrow$  Salir

**Coste del bucle do – while**

$$O(v(n)) \cdot \max\{O(c_e), O(c_s)\}$$

### Llamada a subprogramas

$f(e_1, e_2, \dots, e_n)$

Para ser más exactos, deberíamos haberle sumado a la expresión el  $O(c_s)$ , pero no nos aporta nada ya que siempre se cumple (para estos bucles):

$$O(c_s) \leq O(v(n)) \cdot \max\{O(c_e), O(c_s)\}$$

### ❖ Calcular $e_1, e_2, \dots, e_n$

### ❖ Ejecutar el cuerpo del subprograma

**Coste de un subprograma**

$$\max\{O(c_{e_1}), O(c_{e_2}), \dots, O(c_{e_n}), O(c_f)\}$$

## Reglas prácticas para el cálculo del coste: Algoritmos recursivos

### Recurrencias

En estos casos, en los que queremos calcular el coste de una función recursiva, **la función de coste también es recursiva**, y tiene el siguiente aspecto:

Nº de llamadas recursivas  $\leftarrow$

$$T(n) = \begin{cases} k_1(n) & \text{si } n \text{ es caso base} \\ a \cdot T(n') + k_2(n) & \text{si } n \text{ no es caso base} \end{cases}$$

Coste de cierto  $n' < n$

Coste de las operaciones no recursivas que se realizan en cada llamada

Esto es lo que denominamos **recurrencia** y resolverla va a consistir en dar una expresión no recursiva para  $T(n)$ , pero este proceso puede ser muy, muy complejo en general.

### Resolución de recurrencias

- ❖ No resolveremos completamente una recurrencia
- ❖ Bastará con calcular a qué orden de complejidad pertenece  $T(n)$
- ❖ Vamos a tratar 2 familias de recurrencia:
  - **Reducción mediante sustracción**
  - **Reducción mediante división**

#### Reducción mediante sustracción:

El aspecto general de la recurrencia es:

$$T(n) = \begin{cases} c_b(n) & \text{si } 0 \leq n < b \\ a \cdot T(n-b) + c_{nr}(n) & \text{si } n \geq b \end{cases}$$

- ❖  $c_b(n)$  → Coste del caso base → Aquellos en los que  $n < b$
- ❖  $c_{nr}(n)$  → Coste de las operaciones no recursivas que se realizan en cada llamada
- ❖  $a$  → Nº de llamadas recursivas
- ❖  $n - b$  → Tamaño de los subproblemas generados, es decir, cada uno de ellos reducirá en  $b$  unidades el problema original

No vamos a detallar todos los pasos dados para resolver la recurrencia. Simplemente nos vamos a quedar con el resultado final en el que vamos a tener estas 2 reglas prácticas que distinguen entre **recursión simple** y **recursión múltiple**.

$$T(n) \in \begin{cases} O(n \cdot c_{nr}(n) + c_b(n)) & \text{si } a = 1 \\ O(a^{n \text{ div } b} \cdot (c_{nr}(n) + c_b(n))) & \text{si } a > 1 \end{cases}$$

Recursión simple

Recursión múltiple



### Reducción mediante división:

$$T(n) = \begin{cases} c_b(n) & \text{si } 1 \leq n < b \\ a \cdot T(n \text{ div } b) + c_{nr}(n) & \text{si } n \geq b \end{cases}$$

- ❖  $c_b(n)$  → Coste del caso base
- ❖  $c_{nr}(n)$  → Coste de las operaciones no recursivas en cada llamada
- ❖  $a$  → Nº de llamadas recursivas
- ❖  $n \text{ div } b$  → Tamaño de los subproblemas

Dividir  $n$  entre  $b$  ←

Tras resolver la recurrencia nos encontramos nuevamente con 2 casos:

$$T(n) \in \begin{cases} O(\log_b(n) \cdot c_{nr}(n) + c_b(n)) & \text{si } a = 1 \\ O(n^{\log_b(a)} \cdot (c_{nr}(n) + c_b(n))) & \text{si } a > 1 \end{cases}$$

→ Recursión lineal ( $a = 1$ )

→ Recursión múltiple ( $a > 1$ )