

# Algorítmica



**Los Del DGIIM**, [losdeldgiim.github.io](https://losdeldgiim.github.io)

Doble Grado en Ingeniería Informática y Matemáticas  
Universidad de Granada



Esta obra está bajo una Licencia Creative Commons Atribución-NoComercial-SinDerivadas 4.0 Internacional (CC BY-NC-ND 4.0).

Eres libre de compartir y redistribuir el contenido de esta obra en cualquier medio o formato, siempre y cuando des el crédito adecuado a los autores originales y no persigas fines comerciales.

# Algorítmica

Los Del DGIIM, `losdeldgiim.github.io`

José Juan Urrutia Milán

Granada, 2023-2024



# Índice general

<b>1. Eficiencia de Algoritmos</b>	<b>7</b>
1.1. Análisis de algoritmos . . . . .	10

El siguiente documento pdf no es sino el mero resultado proveniente de la amalgación proficiente de un cúmulo de notas, todas ellas tomadas tras el transcurso de consecutivas clases magistrales, primordialmente —empero, no exclusivamente— de teoría, en simbiosis junto con una síntesis de los apuntes originales provenientes de la asignatura en la que se basan los mismos.

Como motivación para la asignatura, introducimos a continuación un par de problemas que sabremos resolver tras la finalización de esta:

**Ejercicio** (Parque de atracciones). Disponemos de un conjunto de atracciones

$$A_1, A_2, \dots, A_n$$

Para cada atracción, conocemos la hora de inicio y la hora de fin. Podemos proponer varios retos de programación acerca de este parque de atracciones:

1. Seleccionar el mayor número de atracciones que un individuo puede visitar.
2. Seleccionar las atracciones que permitan que un visitante esté ocioso el menor tiempo posible.
3. Conocidas las valoraciones de los usuarios,  $val(A_i)$ , seleccionar aquellas que garanticen la máxima valoración conjunta en la estancia.

**Ejercicio.** Una empresa decide comprar un robot que deberá soldar varios puntos ( $n$ ) en un plano. El software del robot está casi terminado pero falta diseñar el algoritmo que se encarga de decidir en qué orden el robot soldará los  $n$  puntos. Se pide diseñar dicho algoritmo, minimizando el tiempo de ejecución del robot (este depende del tiempo de soldadura que es constante más el tiempo de cada desplazamiento entre puntos, que depende de la distancia entre ellos). Por tanto, deberemos ordenar el conjunto de puntos minimizando la distancia total de recorrido.

## Nociones de conceptos

A lo largo de la asignatura, será común ver los siguientes conceptos, los cuales aclararemos antes de empezar la misma:

- Instancia: Ejemplo particular de un problema.
- Caso: Instancia de un problema con una cierta dificultad.

Generalmente, tendremos tres casos:

- El mejor caso: Instancia con menor número de operaciones y/o comparaciones.
- El peor caso: Instancia con mayor número de operaciones y/o comparaciones.
- Caso promedio. Normalmente, será igual al peor caso.

Para notar la eficiencia del peor caso usaremos  $O(\cdot)$ , mientras que para el mejor caso,  $\Omega(\cdot)$ .

Diremos que un algoritmo es *estable* en ordenación si, dado un criterio de ordenación que hace que dos elementos sean iguales en cuanto a orden, el orden de stos vendrá dado por el primero se que introdujo en la entrada.

**Ejemplo.** Dado el criterio de que un número es menor que otro si es par, ante la instancia del problema: 1, 2, 3, 4. La salida de un algoritmo de ordenación estable según este criterio será:

2, 4, 1, 3

Sin embargo, un ejemplo de salida que podría dar un algoritmo no estable sería:

4, 2, 3, 1

Los datos se encuentra ordenados pero no en el orden de la entrada.

### Algoritmos de ordenación

A continuación, un breve repaso de algoritmos de ordenación:

- Burbuja es el peor algoritmo de ordenación.
- Si tenemos pocos elementos, suele ser más rápido un algoritmos simple como selección o inserción. Entre estos, selección hace muchas comparaciones y pocos intercambios, mientras que inserción hace menos comparaciones y más intercambios. Por tanto, ante datos pesados con varios registros, selección será mejor que inserción.
- Cuando se tienen muchos elementos, es mejor emplear un algoritmo de ordenación del orden  $n \log(n)$ .





# 1. Eficiencia de Algoritmos

La asignatura se centrará en eficiencia basada en el tiempo de ejecución (no en la eficiencia en cuanto espacio, memoria usada por el programa).

Para calcular la eficiencia de un algoritmo, tenemos tres métodos:

- Método empírico: donde se mide el tiempo real.
- Método teórico: donde se mide el tiempo esperado.
- Método híbrido: tiempo teórico evitando las constantes mediante resultados empíricos.

**Proposición 1.1** (Principio de Invarianza). *Dadas dos implementaciones  $I_1, I_2$  de un algoritmo, el tiempo de ejecución para una misma instancia de tamaño  $n$ ,  $T_{I_1}(n)$  y  $T_{I_2}(n)$ , no diferirá en más de una constante multiplicativa. Es decir,  $\exists K > 0$  que verifica:*

$$T_{I_1}(n) \leq K \cdot T_{I_2}(n)$$

principio Por lo que podremos despreciar las constantes. En un principio, se asumirá que operaciones básicas como sumas, multiplicaciones, ... serán de tiempo constante, salvo excepciones (por ejemplo, multiplicaciones de números de 100000 dígitos).

**Definición 1.1** (Notación  $O$ ). Se dice que un algoritmo  $A$  es de orden  $O(f(n))$ , donde  $f$  es una función  $f : \mathbb{N} \rightarrow \mathbb{R}^+$ , cuando existe una implementación del mismo tamaño cuyo tiempo de ejecución  $T_A(n)$  es menor igual que  $K \cdot f(n)$ , donde  $K$  es una constante real positiva a partir de un tamaño grande  $n_0$ . Formalmente:

$$A \text{ es } O(f(n)) \Leftrightarrow \exists K \in \mathbb{R}^+, \exists n_0 \in \mathbb{N} \mid T_A(n) \leq K \cdot f(n) \quad \forall n \geq n_0$$

La notación  $O$  nos permite conocer cómo se comportará el algoritmo en términos de eficiencia en instancias del caso pero del problema, como mucho, sabemos que el algoritmo no tardará más de  $K \cdot f(n)$  en ejecutarse, en el peor de los casos.

Al decir que el algoritmo  $A$  es de orden  $O(f(n))$ , decimos que siempre podemos encontrar una constante positiva  $K$  que para valores muy grandes del caso  $n$  (a partir de un  $n_0$ ), el tiempo de ejecución del algoritmo siempre será inferior a  $K \cdot f(n)$ :

$$T_A(n) \leq K \cdot f(n)$$

Ejemplos de órdenes de eficiencia son:

- Constante,  $O(1)$ .

- Logarítmico,  $O(\log(n))$ .
- Lineal,  $O(n)$ .
- Cuadrático,  $O(n^2)$ .
- Exponencial,  $O(a^n)$ .
- $\vdots$

**Proposición 1.2** (Principio de comparación). *Para saber si dos órdenes  $O(f(n))$  y  $O(g(n))$  son equivalentes o no, aplicamos las siguientes reglas:*

$$O(f(n)) \equiv O(g(n)) \Leftrightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \rightarrow K \in \mathbb{R}^+$$

$$O(f(n)) > O(g(n)) \Leftrightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \rightarrow \infty$$

$$O(f(n)) < O(g(n)) \Leftrightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \rightarrow 0$$

*Entendiendo que un orden es menor que otro si es mejor, es decir, más rápido en el caso asintótico.*

**Ejemplo.** Si tenemos dos algoritmos  $A$  y  $B$  con órdenes de eficiencia  $O(n^2)$  y  $O((4n+1)^2 + n)$  respectivamente, tratamos de ver qué algoritmos es más eficiente:

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \lim_{n \rightarrow \infty} \frac{n^2}{(4n+1)^2 + n} = \lim_{n \rightarrow \infty} \frac{n^2}{(16n^2 + 1 + 2 \cdot 4n \cdot 1) + n} = \lim_{n \rightarrow \infty} \frac{1}{16}$$

Gracias a la Proposición 1.2, tenemos que los algoritmos  $A$  y  $B$  son equivalentes.

**Ejemplo.** En esta ocasión, tenemos a dos algoritmos  $A$  y  $B$  con órdenes de eficiencia de  $O(2^n)$  y  $O(3^n)$ , respectivamente.

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \lim_{n \rightarrow \infty} \frac{2^n}{3^n} = \lim_{n \rightarrow \infty} \left(\frac{2}{3}\right)^n = 0$$

Por la Proposición 1.2,  $A$  es más eficiente que  $B$ .

**Ejemplo.** El algoritmo  $A$  tiene una eficiencia  $O(n)$  y el algoritmo  $B$  tiene una eficiencia de  $O(n \log(n))$ . Buscamos cuál es más eficiente.

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \lim_{n \rightarrow \infty} \frac{n}{n \log(n)} = \lim_{n \rightarrow \infty} \frac{1}{\log(n)} = 0$$

Por lo que  $A$  es más eficiente que  $B$ , por la Proposición 1.2.

**Ejemplo.** Disponemos de dos algoritmos,  $A$  y  $B$  con órdenes de eficiencia  $O((n^2 + 29)^2)$  y  $O(n^3)$  respectivamente. Intuimos que  $B$  es más eficiente que  $A$  pero queremos probarlo.

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \lim_{n \rightarrow \infty} \frac{(n^2 + 29)^2}{n^3} = \infty$$

Gracias a la Proposición 1.2, hemos probado lo que esperábamos;  $B$  es más eficiente que  $A$ .

**Ejemplo.** Se quiere probar que  $O(\log(n))$  es más eficiente que  $O(n)$ .

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \lim_{n \rightarrow \infty} \frac{n}{\log(n)} = \lim_{n \rightarrow \infty} \frac{10^n}{n} = \infty$$

Por la Proposición 1.2, lo acabamos de probar.

**Ejemplo.** Se quiere dar un ejemplo de que el orden de eficiencia de los logaritmos es equivalente sin importar la base de este. Podemos ver qué sucede con  $O(\log_2(n))$  y con  $O(\log_3(n))$ :

$$\lim_{n \rightarrow \infty} \frac{\log_3(n)}{\log_2(n)} = \lim_{n \rightarrow \infty} \frac{\ln(2)}{\ln(3)}$$

Por lo que ambos algoritmos tienen el mismo orden de eficiencia.

**Definición 1.2** (Notación Omega). Se dice que un algoritmo  $A$  es de orden  $\Omega(f(n))$ , donde  $f$  es una función  $f : \mathbb{N} \rightarrow \mathbb{R}^+$ , cuando existe una implementación del mismo tamaño cuyo tiempo de ejecución  $T_A(n)$  es mayor igual que  $K \cdot f(n)$ , donde  $K$  es una constante real positiva a partir de un tamaño grande  $n_0$ . Formalmente:

$$A \text{ es } \Omega(f(n)) \Leftrightarrow \exists K \in \mathbb{R}^+, \exists n_0 \in \mathbb{N} \mid T_A(n) \geq K \cdot f(n) \quad \forall n > n_0$$

La notación  $\Omega$  nos permite conocer cómo se comportará el algoritmo en términos de eficiencia en instancias del caso mejor del problema. Como poco, sabemos que el algoritmo no tardará menos de  $K \cdot f(n)$  en ejecutarse, en el mejor de los casos.

**Definición 1.3** (Notación Theta). Se dice que un algoritmo  $A$  es de orden exacto  $\theta(f(n))$ , donde  $f$  es una función  $f : \mathbb{N} \rightarrow \mathbb{R}^+$ , cuando existe una implementación del mismo tamaño cuyo tiempo de ejecución  $T_A(n)$  es igual que  $K \cdot f(n)$ , donde  $K$  es una constante real positiva a partir de un tamaño grande  $n_0$ . En este caso, el algoritmo es simultáneamente de orden  $O(f(n))$  y  $\Omega(g(n))$ .

$$A \text{ es } \theta(f(n)) \Leftrightarrow \exists K \in \mathbb{R}^+, \exists n_0 \in \mathbb{N} \mid T_A(n) = K \cdot f(n) \quad \forall n > n_0$$

## Propiedades

A continuación, vemos algunas propiedades de las notaciones anteriormente vistas:

Reflexiva.

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

También para las notaciones  $\Omega$  y  $\theta$ .

Simétrica.

$$f(n) \in \theta(g(n)) \Leftrightarrow g(n) \in \theta(f(n))$$

Suma.

Si  $T_1(n) \in O(f(n))$  y  $T_2(n) \in O(g(n))$ . Entonces:

$$T_1(n) + T_2(n) \in O(\max(f(n), g(n)))$$

Producto.

Si  $T_1(n) \in O(f(n))$  y  $T_2(n) \in O(g(n))$ . Entonces:

$$T_1(n) \cdot T_2(n) \in O(f(n) \cdot g(n))$$

Regla del máximo.

$$O(f(n) + g(n)) = \max(O(f(n)), O(g(n)))$$

Regla de la suma.

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

Regla del producto.

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

Puede suceder que el tamaño del problema no depende de una única variable  $n$ , sino de varias. En estos casos, se analiza de igual forma que en el caso de una variable, pero con una función de varias variables. Conocida una función  $f : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{R}^+$ :

$$A \text{ es } O(f(n, m)) \Leftrightarrow \exists K \in \mathbb{R}^+ \mid T_A(n, m) \leq K \cdot f(n, m) \quad \forall n, m \in \mathbb{N}$$

**Ejemplo.** El orden de eficiencia del algoritmo canónico (el que todos conocemos) de suma de matrices  $n \times m$  es de orden  $O(n \cdot m)$ .

## 1.1. Análisis de algoritmos

El primer paso a la hora de determinar la eficiencia de un algoritmo es identificar qué parámetro determina el tamaño del problema ( $n$ ). Posteriormente, tenemos que tener claro como se analiza cada estructura del código:

1. Operaciones elementales.
2. Secuencias de sentencias.
3. Sentencias condicionales.
4. Sentencias repetitivas.
5. Llamadas a funciones no recursivas.
6. Llamadas a funciones recursivas.

### Sentencias simples u operaciones elementales

Son aquellas instrucciones cuya ejecución no depende del tamaño del caso, como por ejemplo:

- Operaciones matemáticas básicas (sumas, multiplicaciones, ...).
- Comparaciones.
- Operaciones booleanas.

Su tiempo de ejecución está acotado superiormente por una constante. Su orden es  $O(1)$ .

**Secuencias de sentencias**

Constan de la ejecución de secuencias de bloques de sentencias:

Suponiendo que cada sentencia  $i$  tiene eficiencia  $O(f_i(n))$ , la eficiencia de la secuencia se obtiene mediante las reglas de la suma y del máximo:

$$O(f_1(n) + f_2(n) + \cdots + f_r(n)) = \text{máx}[O(f_1(n)), O(f_2(n)), \dots, O(f_r(n))]$$

**Ejemplo.** Un ejemplo que puede parecer confuso es el siguiente:

En este caso, se trata de un código de orden  $O(1)$ , ya que es para un valor fijo de  $n$ , 10.