# ANÁLISIS DE COMPLEJIDAD

Análisis y Diseño de Datos y Algoritmos
Apuntes creados por David Romero
26/08/2019 – 1.0.1

**Atribución-NoComercial-CompartirIgual**
**4.0 Internacional (CC BY-NC-SA 4.0)**

# ANÁLISIS DE COMPLEJIDAD

1. ¿Qué es la complejidad de un algoritmo?
2. Órdenes de complejidad
3. Problemas P, NP y NP-completos
4. Resolución de ecuaciones en recurrencia
5. Resolución de sumatorios
6. Estudio de la complejidad de un algoritmo
7. Ejercicios resueltos



Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

# ANÁLISIS DE COMPLEJIDAD

1. ¿Qué es la complejidad de un algoritmo?
2. Órdenes de complejidad
3. Problemas P, NP y NP-completos
4. Resolución de ecuaciones en recurrencia
5. Resolución de sumatorios
6. Estudio de la complejidad de un algoritmo
7. Ejercicios resueltos

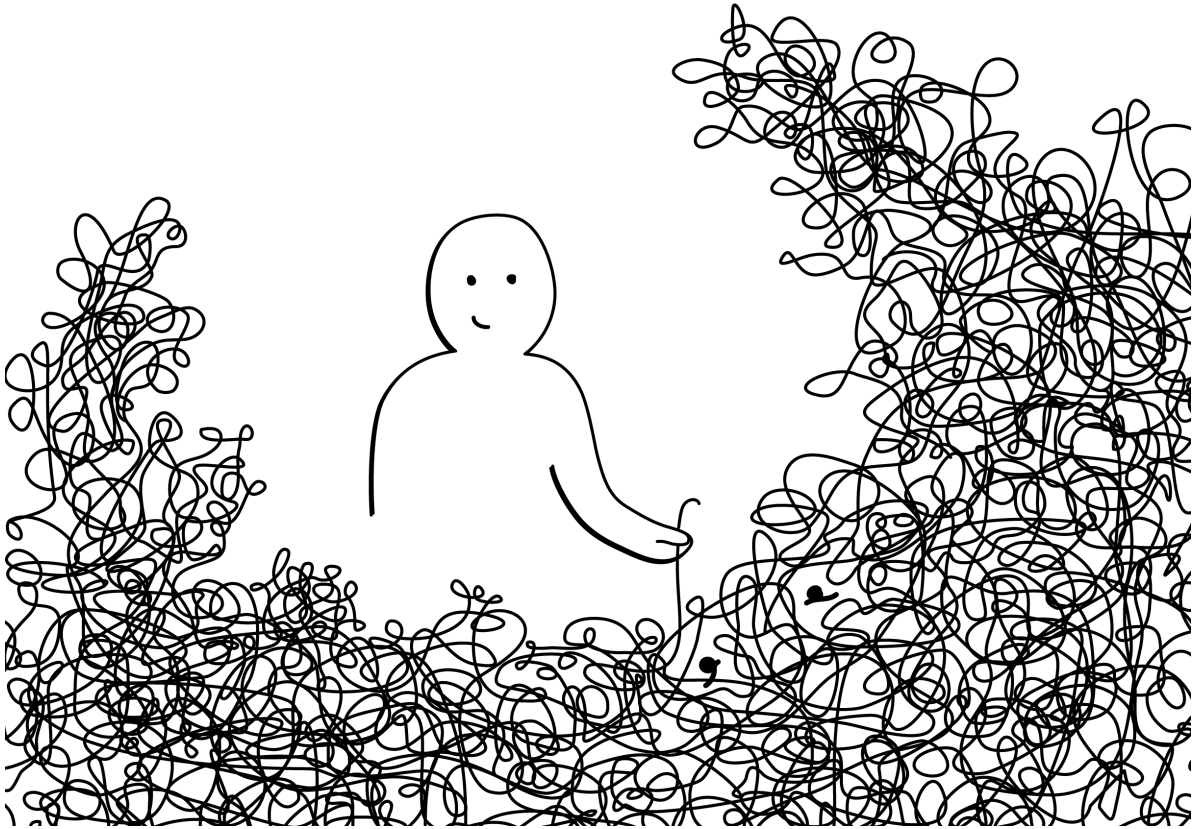


Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

# 1.1. Motivación

¿Todos los algoritmos son igual de eficientes? ¿Cuánto tardará el algoritmo que acabo de implementar? Y si diseño una alternativa, ¿será mejor o peor?

Es más, ¿todos los problemas tienen solución? ¿O lo tienen en un tiempo razonable? Vamos a intentar responder a estas preguntas.





Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

# 1.2. Definición de complejidad

La **complejidad de un algoritmo** es una **métrica** expresada en función del tamaño del problema.

En dicha métrica se mide la cantidad de recursos (tiempo y espacio) que se necesita para llevar a cabo dicho algoritmo.

Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

# ANÁLISIS DE COMPLEJIDAD

1. ¿Qué es la complejidad de un algoritmo?
2. Órdenes de complejidad
3. Problemas P, NP y NP-completos
4. Resolución de ecuaciones en recurrencia
5. Resolución de sumatorios
6. Estudio de la complejidad de un algoritmo
7. Ejercicios resueltos



Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

# 2.1. Tiempo de ejecución

Se define $T(n)$ como la función que nos da el tiempo de ejecución en función del tamaño $n$.

$\boldsymbol{T(n)}$ **= tiempo de ejecución en función de n**

Pero, ¿cuándo una función es mayor o menor que otra?



Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

## 2.2. Relación entre funciones

Sean $h(n)$ y $g(n)$ dos funciones de tiempo de ejecución, monótonas y crecientes.

## 2.3. Equivalencias (I)

Equivalencias **en el infinito**

- $<_\infty$ menor
- $>_\infty$ mayor
- $\leq_\infty$ menor o igual
- $\geq_\infty$ mayor o igual
- $\approx_\infty$ equivalente
- $=_\infty$ igual



Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

## 2.2. Relación entre funciones

Sean $h(n)$ y $g(n)$ dos funciones de tiempo de ejecución, monótonas y crecientes.

## 2.3. Equivalencias (II)

Se dice que $h(n)$ es **menor** que $g(n)$ cuando el límite de su cociente es 0.

$$h(n) <_{\infty} g(n) \leftrightarrow \lim_{n \to \infty} \frac{h(n)}{g(n)} = 0$$



Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

## 2.2. Relación entre funciones

Sean $h(n)$ y $g(n)$ dos funciones de tiempo de ejecución, monótonas y crecientes.

## 2.3. Equivalencias (III)

Se dice que $h(n)$ es **mayor** que $g(n)$ cuando el límite de su cociente es $\infty$.

$$h(n) >_{\infty} g(n) \leftrightarrow \lim_{n \to \infty} \frac{h(n)}{g(n)} = \infty$$



Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

## 2.2. Relación entre funciones

Sean $h(n)$ y $g(n)$ dos funciones de tiempo de ejecución, monótonas y crecientes.

## 2.3. Equivalencias (IV)

Se dice que $h(n)$ es **equivalente** a $g(n)$ cuando el límite de su cociente es menor que ∞.

$$h(n) \approx_{\infty} g(n) \leftrightarrow \lim_{n \to \infty} \frac{h(n)}{g(n)} < \infty$$



Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

## 2.2. Relación entre funciones

Sean $h(n)$ y $g(n)$ dos funciones de tiempo de ejecución, monótonas y crecientes.

## 2.3. Equivalencias (V)

Se dice que $h(n)$ es **igual** a $g(n)$ cuando el límite de su cociente es 1

$$h(n) =_{\infty} g(n) \leftrightarrow \lim_{n \to \infty} \frac{h(n)}{g(n)} = 1$$



Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

## 2.2. Relación entre funciones

Sean $h(n)$ y $g(n)$ dos funciones de tiempo de ejecución, monótonas y crecientes.

## 2.3. Equivalencias (VI)

Se dice que $h(n)$ es **menor o igual** que $g(n)$ cuando el límite de su cociente es menor que ∞

$$h(n) \leq_{\infty} g(n) \leftrightarrow 0 \leq \lim_{n \to \infty} \frac{h(n)}{g(n)} < \infty$$



Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

## 2.2. Relación entre funciones

Sean $h(n)$ y $g(n)$ dos funciones de tiempo de ejecución, monótonas y crecientes.

## 2.3. Equivalencias (VII)

Se dice que $h(n)$ es **mayor o igual** que $g(n)$ cuando

$$h(n) \geq_{\infty} g(n) \leftrightarrow \lim_{n \to \infty} \frac{h(n)}{g(n)} > 0$$



Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

# 2.4. Orden de Complejidad Exacto

Las relaciones anteriores definen un conjunto de clases de equivalencia. A cada una de esas clases de equivalencia se le llama **Orden de Complejidad Exacto** ($\Theta$).

El orden de complejidad exacto de $g(n)$ es el **conjunto de funciones que son equivalentes a ella en el infinito**.

$$\Theta\big(\boldsymbol{g(n)}\big) = \{\boldsymbol{f(n)} \mid \lim_{\boldsymbol{n}\to\infty} \frac{\boldsymbol{f(n)}}{\boldsymbol{g(n)}} = \boldsymbol{k}, \boldsymbol{0} < \boldsymbol{k} < \infty\}$$



Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

## 2.5. Jerarquía de órdenes de complejidad

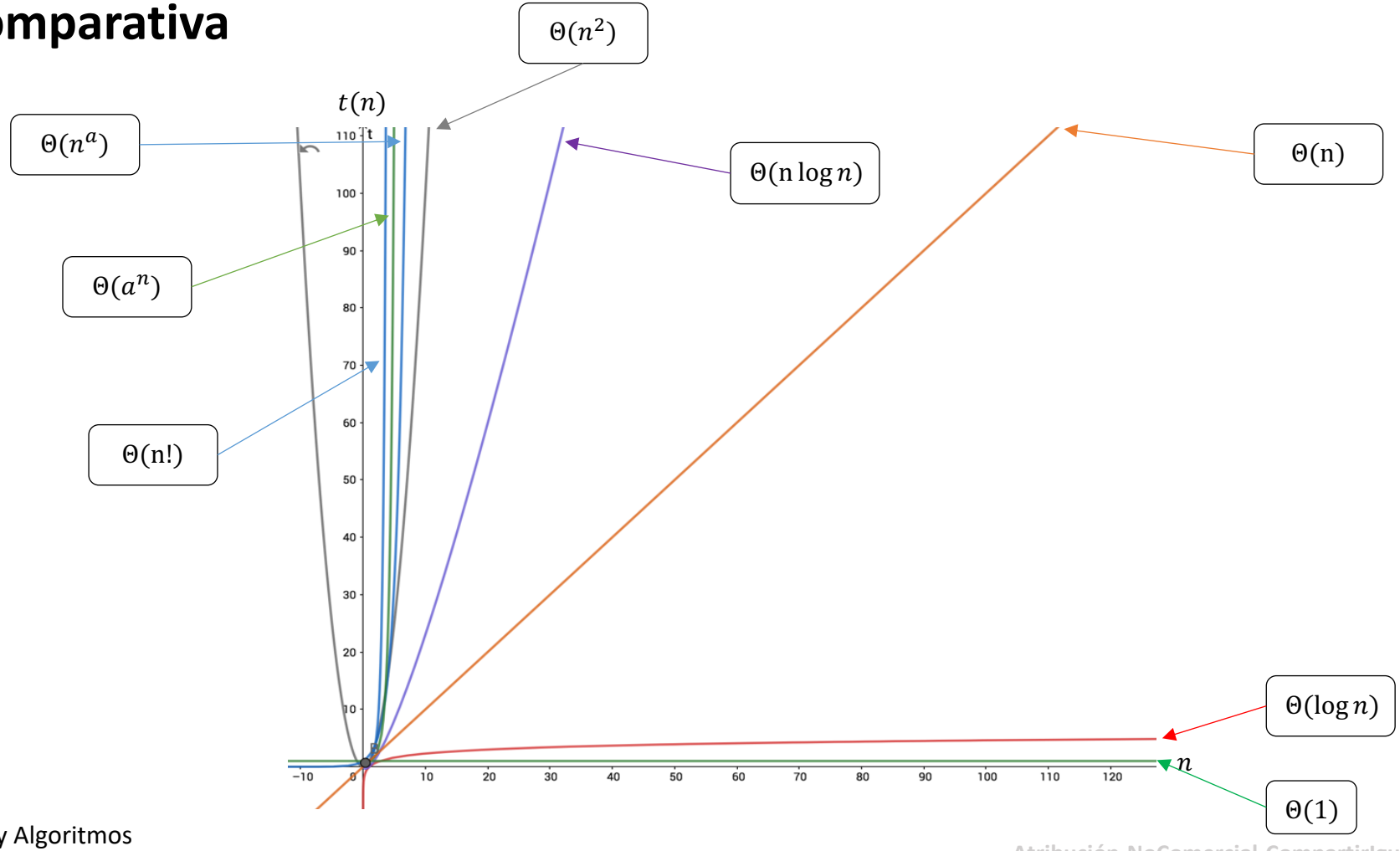
| | |
|---|---|
| $\Theta(1)$ | Orden constante |
| $\Theta(\log n)$ | Orden logarítmico |
| $\Theta(\mathrm{n})$ | Orden lineal |
| $\Theta(\mathrm{n} \log n)$ | Orden cuasi-lineal |
| $\Theta(n^2)$ | Orden cuadrático |
| $\Theta(n^3)$ | Orden cúbico |
| $\Theta(n^a)$ | Orden polinómico ($a > 3$) |
| $\Theta(a^n)$ | Orden exponencial ($a > 1$) |
| $\Theta(\mathrm{n}!)$ | Orden factorial |

$$\boldsymbol{\Theta(1) < \Theta(log\, n) < \Theta(n) < \Theta(\mathrm{n}\, log\, n) < \Theta(n^2) < \Theta(n^3) < \Theta(n^a) < \Theta(a^n) < \Theta(\mathrm{n}!) < \Theta(n^n)}$$



Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

# 2.6. Gráfica comparativa






Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

## 2.7. Cotas

A veces, no podemos obtener un orden de complejidad exacto.

Sean $f(n)$ y $g(n)$ dos funciones de tiempo de ejecución, monótonas y crecientes.

### Cota superior $O$

Se define una **cota superior** $O(g(n))$ cuando

$$f(n) \in O(g(n)) \; si \; \lim_{n \to \infty} \frac{f(n)}{g(n)} < \infty$$

### Cota inferior $\Omega$

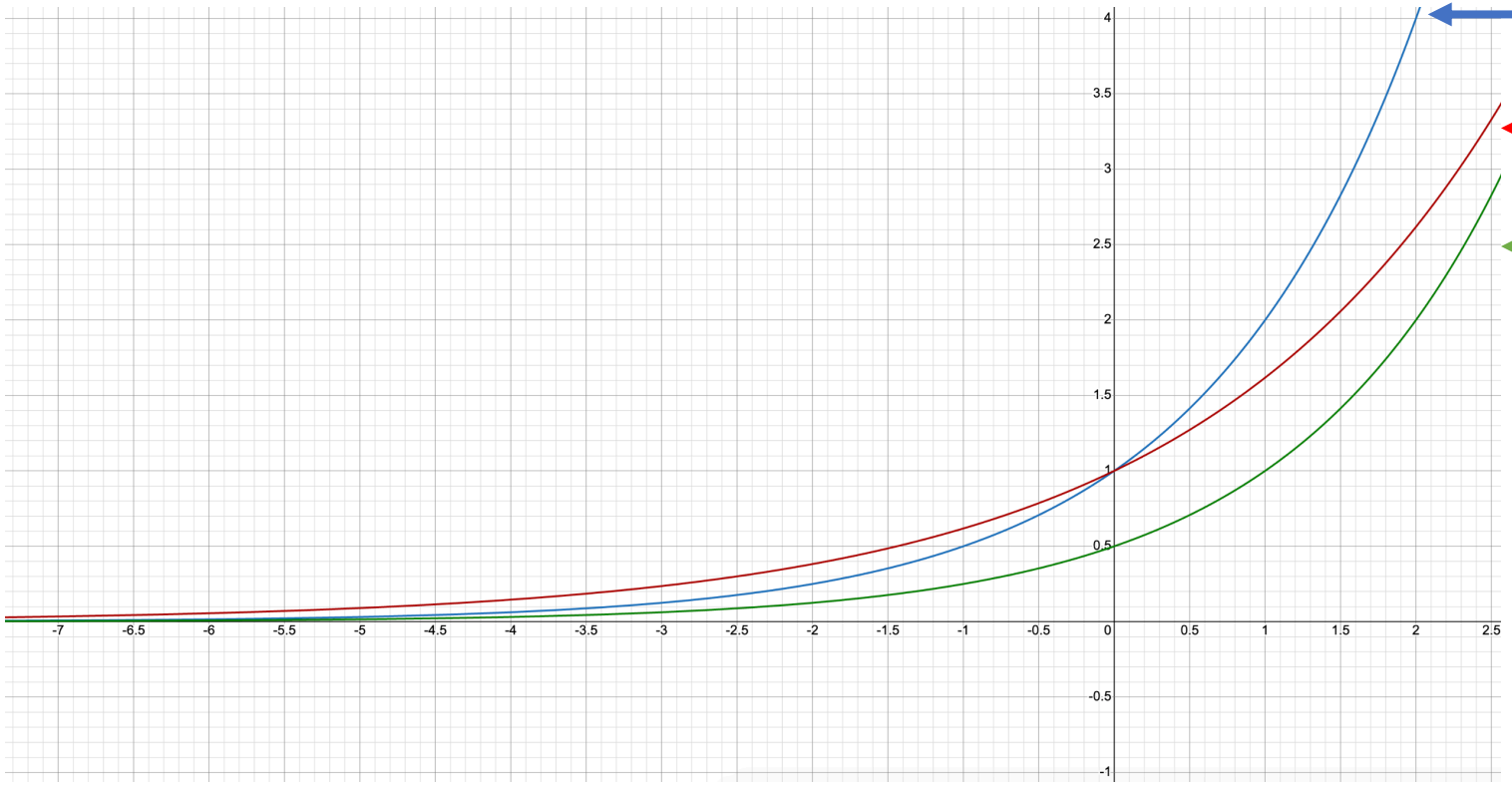
Se define una **cota inferior** $\Omega(g(n))$ cuando

$$f(n) \in \Omega(g(n)) \; si \; \lim_{n \to \infty} \frac{f(n)}{g(n)} > 0$$



Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

# 2.7. Cotas




$O(\mathrm{g(n)})$ — *Cota superior*

$\Theta(\mathrm{g(n)})$ — *Orden exacto*

$\Omega(\mathrm{g(n)})$ — *Cota inferior*

Complejidad de Fibonacci:

$$O\big(\mathrm{g(n)}\big) = 2^n$$

$$\theta\ \big(\mathrm{g(n)}\big) = \left(\frac{1+\sqrt{5}}{2}\right)^n$$

$$\Omega\ \big(\mathrm{g(n)}\big) = 2^{n/2}$$



Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

# ANÁLISIS DE COMPLEJIDAD

1. ¿Qué es la complejidad de un algoritmo?
2. Órdenes de complejidad
3. Problemas P, NP y NP-completos
4. Resolución de ecuaciones en recurrencia
5. Resolución de sumatorios
6. Estudio de la complejidad de un algoritmo
7. Ejercicios resueltos



Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

# 3.1. Problemas P

Los algoritmos de complejidad polinómica se dicen que son **tratables** en el sentido de que suelen ser abordables en la práctica **en un tiempo polinómico (P).**

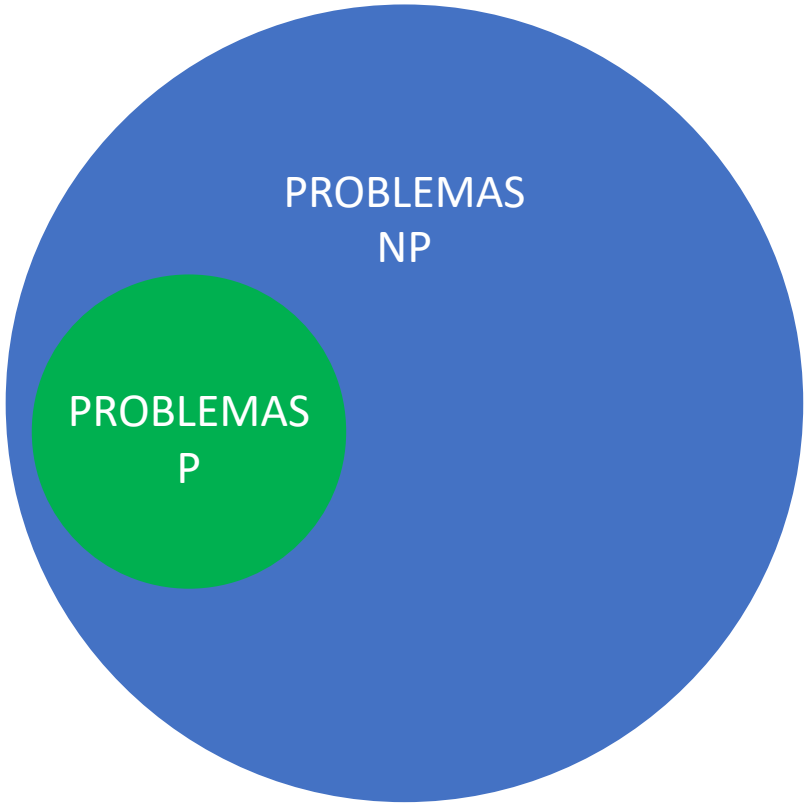
Ejemplos: algoritmos de programación lineal, elevar al cuadrado, funciones simples, suma y productos de números naturales, etc.

Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

## 3.2. Problemas NP

Los problemas que **no se pueden resolver en tiempo polinomial (NP)** se dicen que son **intratables**, aunque se puede aplicar un algoritmo polinómico para comprobar si la solución obtenida es o no válida.

Ejemplos: calcular una raíz cuadrada, camino hamiltoniano, etc.





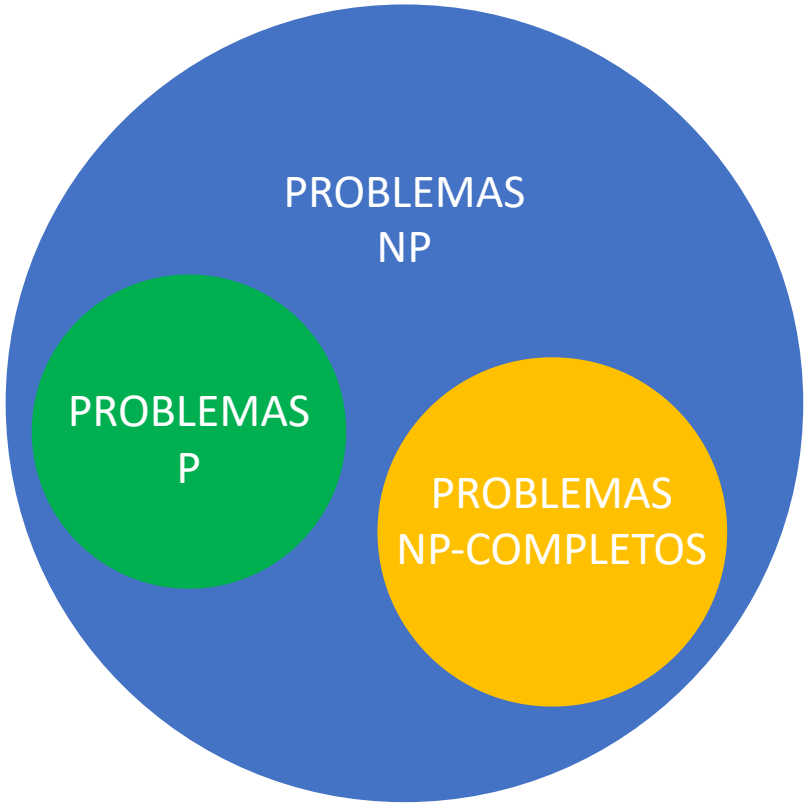

Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

# 3.3. Problemas NP-completos

Son algoritmos de **extrema complejidad**.

Se llaman así por encontrarse en la frontera externa de la clase NP. Estos problemas son todos “iguales”; si se encontrara una solución para uno de ellos, se encontraría una solución para el resto.

Ejemplos: problema de la suma de subconjuntos, problema del viajante, etc.






Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

# ANÁLISIS DE COMPLEJIDAD

1. ¿Qué es la complejidad de un algoritmo?
2. Órdenes de complejidad
3. Problemas P, NP y NP-completos
4. Resolución de ecuaciones en recurrencia
5. Resolución de sumatorios
6. Estudio de la complejidad de un algoritmo
7. Ejercicios resueltos



Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

# 4.1. Definiciones

**Recurrencia lineal:** el tamaño del problema disminuye en cada iteración en una razón aritmética.

**Recurrencia no lineal:** el tamaño del problema disminuye en cada iteración en una razón geométrica.



Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

# 4.2. Recurrencias lineales

Sea una ecuación de recurrencia lineal del tipo:

$$T(n) = aT(n-b) + n^d log^p n$$

Su **orden de complejidad exacto** es:

$$T(n) = \begin{cases} \Theta\left(a^{\frac{n}{b}} log^p n\right), & \text{si a} > 1 \\ \Theta\left(n^{d+1} log^p n\right), & \text{si a} = 1 \\ \Theta\left(n^d log^p n\right), & \text{si a} < 1 \end{cases}$$



Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

# 4.2. Recurrencias lineales. Cotas

Sea una ecuación de recurrencia lineal del tipo:

$$T(n) = a_1 T(n - b_1) + \cdots + a_k T(n - b_k) + c_1^n g_1(n) + \cdots + c_j^n g_j(n);\ b_i < b_{i+1};\ c_i < c_{i+1}$$

Su **cota superior** es:

$$T(n) <_\infty (a_1 + \cdots + a_k) T(n - b_1) + c_j^n g(n)$$

Su **cota inferior** es:

$$T(n) >_\infty (a_1 + \cdots + a_k) T(n - b_k) + c_j^n g(n)$$



Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

# 4.2. Recurrencias lineales. Ejemplos

- $T(n) = T(n-1)$

  $a = 1, b = 1, d = 0, p = 0$

  $T(n) = 1 * T(n-1) + n^0 log^0 n = \boldsymbol{\Theta(n)}$

- $T(n) = 2T(n-1)$

  $a = 2, b = 1, d = 0, p = 0$

  $T(n) = 2 * T(n-1) + n^0 log^0 n = \boldsymbol{\Theta(2^n)}$

- $T(n) = 2T(n-2) + n$

  $a = 2, b = 2, d = 1, p = 0$

  $T(n) = 2 * T(n-2) + n^1 log^0 n = \boldsymbol{\Theta(2^{n/2})}$

Orden de complejidad exacto

$T(n) = aT(n-b) + n^d log^p n$

$$T(n) = \begin{cases} \Theta\left(a^{\frac{n}{b}} log^p n\right), & \text{si a} > 1 \\ \Theta\left(n^{d+1} log^p n\right), & \text{si a} = 1 \\ \Theta\left(n^d log^p n\right), & \text{si a} < 1 \end{cases}$$

Cotas

$T(n) = a_1 T(n-b_1) + \cdots + a_k T(n-b_k) + c_1^n g_1(n) + \cdots + c_j^n g_j(n)$

$T(n) <_\infty (a_1 + \cdots + a_k) T(n-b_1) + c_j^n g(n)$

$T(n) >_\infty (a_1 + \cdots + a_k) T(n-b_k) + c_j^n g(n)$



Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

# 4.2. Recurrencias lineales. Ejemplos

- $T(n) = T(n-1) + T(n-2)$

  $b_1 = 1, b_k = 2, c_j^n = 0$

  $T(n) <_\infty 2T(n-1) = \boldsymbol{\Theta(2^n)}$

  $T(n) >_\infty 2T(n-2) = \boldsymbol{\Theta(2^{n/2})}$

- $T(n) = 2T(n-1) + 3T(n-2) + 3n^2 + 2n$

  $b_1 = 1, b_k = 2, c_j^n = 3$

  $T(n) <_\infty 5T(n-1) + 3n^2 = \boldsymbol{\Theta(5^n)}$

  $T(n) >_\infty 5T(n-2) + 3n^2 = \boldsymbol{\Theta(5^{n/2})}$

Orden de complejidad exacto

$T(n) = aT(n-b) + n^d log^p n$

$$T(n) = \begin{cases} \Theta\left(a^{\frac{n}{b}} log^p n\right), & \text{si a} > 1 \\ \Theta\left(n^{d+1} log^p n\right), & \text{si a} = 1 \\ \Theta\left(n^d log^p n\right), & \text{si a} < 1 \end{cases}$$

Cotas

$T(n) = a_1 T(n-b_1) + \cdots + a_k T(n-b_k) + c_1^n g_1(n) + \cdots + c_j^n g_j(n)$

$T(n) <_\infty (a_1 + \cdots + a_k) T(n-b_1) + c_j^n g(n)$

$T(n) >_\infty (a_1 + \cdots + a_k) T(n-b_k) + c_j^n g(n)$



Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

# 4.3. Recurrencias no lineales

Sea una ecuación de recurrencia no lineal del tipo:

$$T(n) = aT\left(\frac{n}{b}\right) + n^d log^p n$$

Su **orden de complejidad exacto** definido por el Teorema Maestro es:

$$T(n) = \begin{cases} \Theta\left(n^{\log_b a}\right), & \text{si a} > b^d \\ \Theta\left(n^d log^{p+1} n\right), & \text{si a} = b^d \\ \Theta\left(n^d log^p n\right), & \text{si a} < b^d \end{cases}$$



Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

# 4.3. Recurrencias no lineales. Cotas

Sea una ecuación de recurrencia no lineal del tipo:

$$T(n) = a_1 T\left(\frac{n}{b_1}\right) + \cdots + a_k T\left(\frac{n}{b_k}\right) + g(n); b_i < b_{i+1}$$

Su **cota superior** es:

$$T(n) <_\infty (a_1 + \cdots + a_k) T\left(\frac{n}{b_1}\right) + g(n)$$

Su **cota inferior** es:

$$T(n) >_\infty (a_1 + \cdots + a_k) T\left(\frac{n}{b_k}\right) + g(n)$$



Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

# 4.3. Recurrencias no lineales. Ejemplos

- $T(n) = T(n/2)$

  $a = 1, b = 2, d = 0, p = 0$

  $T(n) = 1 * T\left(\frac{n}{2}\right) + n^0 log^0 n = \boldsymbol{\Theta(log\ n)}$

- $T(n) = 2T(n/2)$

  $a = 2, b = 2, d = 0, p = 0$

  $T(n) = 2 * T\left(\frac{n}{2}\right) = \Theta\left(n^{log_2 2}\right) = \boldsymbol{\Theta(n)}$

- $T(n) = 2T(n/3) + n$

  $a = 2, b = 3, d = 1, p = 0$

  $T(n) = 2 * T\left(\frac{n}{3}\right) + n^1 log^0 n = \boldsymbol{\Theta(n)}$

Orden de complejidad exacto

$$T(n) = aT\left(\frac{n}{b}\right) + n^d log^p n$$

$$T(n) = \begin{cases} \Theta\left(n^{\log_b a}\right), & \text{si a} > b^d \\ \Theta\left(n^d log^{p+1} n\right), & \text{si a} = b^d \\ \Theta\left(n^d log^p n\right), & \text{si a} < b^d \end{cases}$$

Cotas

$$T(n) = a_1 T\left(\frac{n}{b_1}\right) + \cdots + a_k T\left(\frac{n}{b_k}\right) + g(n); b_i < b_{i+1}$$

$$T(n) <_\infty (a_1 + \cdots + a_k) T\left(\frac{n}{b_1}\right) + g(n)$$

$$T(n) >_\infty (a_1 + \cdots + a_k) T\left(\frac{n}{b_k}\right) + g(n)$$



Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

# 4.3. Recurrencias no lineales. Ejemplos

- $T(n) = T(n/2) + T(n/2)$

  $b_1 = 2, b_k = 2, c_j^n = 0$

  $T(n) <_\infty 2T(n/2) = \boldsymbol{\Theta\left(n^{log_2 2}\right) = \Theta(n)}$

  $T(n) >_\infty 2T(n/2) = \boldsymbol{\Theta\left(n^{log_2 2}\right) = \Theta(n)}$

- $T(n) = 2T(n/2) + T(n/3) + n^2 \log n$

  $b_1 = 2, b_k = 3, g(n) = n^2 \log n$

  $T(n) <_\infty 3T(n/2) + n^2 \log n = \boldsymbol{\Theta(n^2 \log n)}$

  $T(n) >_\infty 3T(n/3) + n^2 \log n = \boldsymbol{\Theta(n^2 \log n)}$

Orden de complejidad exacto

$$T(n) = aT\left(\frac{n}{b}\right) + n^d log^p n$$

$$T(n) = \begin{cases} \Theta\left(n^{\log_b a}\right), & \text{si a} > b^d \\ \Theta\left(n^d log^{p+1} n\right), & \text{si a} = b^d \\ \Theta\left(n^d log^p n\right), & \text{si a} < b^d \end{cases}$$

Cotas

$$T(n) = a_1 T\left(\frac{n}{b_1}\right) + \cdots + a_k T\left(\frac{n}{b_k}\right) + g(n); b_i < b_{i+1}$$

$$T(n) <_\infty (a_1 + \cdots + a_k) T\left(\frac{n}{b_1}\right) + g(n)$$

$$T(n) >_\infty (a_1 + \cdots + a_k) T\left(\frac{n}{b_k}\right) + g(n)$$



Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

# ANÁLISIS DE COMPLEJIDAD

1. ¿Qué es la complejidad de un algoritmo?
2. Órdenes de complejidad
3. Problemas P, NP y NP-completos
4. Resolución de ecuaciones en recurrencia
5. Resolución de sumatorios
6. Estudio de la complejidad de un algoritmo
7. Ejercicios resueltos



Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

# 5.1. Propiedades básicas de los sumatorios

$$\sum_{i=a}^{b} f_i = f_a + f_{a+1} + \cdots + f_b$$

$$\sum_{i=a}^{b} (f_i + g_i) = \sum_{i=a}^{b} f_i + \sum_{i=a}^{b} g_i$$

$$\sum_{i=a}^{b} \sum_{j=c}^{d} f(i)g(j) = \sum_{i=a}^{b} f(i) \sum_{j=c}^{d} g(j)$$

$$\sum_{i=a}^{b} g(x)f(i) = g(x) \sum_{i=a}^{b} f(i), \qquad si\ g(x)\ no\ depende\ de\ i$$

$$\sum_{i=a}^{b} f(i) = \sum_{i=a}^{c} f(i) - \sum_{i=a}^{b-1} f(i), \qquad con\ c > b$$

$$\sum_{i=n}^{a} f(i) = \sum_{i=a}^{n} f(i)$$



Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

## 5.2. Sumatorios usuales

$$\sum_{i=a}^{n} 1 \cong n$$

$$\sum_{i=a}^{n} i = \frac{n(n+1)}{2} \approx \frac{1}{2}n^2 + \cdots$$

$$\sum_{i=a}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \approx \frac{1}{3}n^3$$

$$\sum_{i=a}^{n} i^k = \frac{1}{k+1}n^{k+1} + \cdots$$



Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

## 5.3. Resolución de un sumatorio aritmético

```
i = 0;
while(i<=n)
    i = i+2;
    (aquí va una operación constante k0)
}
```

La variable $i$ siempre toma valores **múltiplos** de 2

$$\sum_{i=0+2u}^{n} k_0 \longrightarrow \sum_{\boldsymbol{i=a+r*u}}^{\boldsymbol{n}} \boldsymbol{k_0 f(i)}; \boldsymbol{T(n) = k_0 * \frac{1}{r} * n}$$

$$u = \begin{cases} 1 & si\ la\ actualización\ es\ positiva \\ -1 & si\ la\ actualización\ es\ negativa \end{cases}$$



Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

## 5.3. Resolución de un sumatorio geométrico

```
i = n;
while(i>=1){
    i = i/3;
    (aquí va una operación constante k0)
}
```

La variable $i$ tomará valores que sean **potencias** de 3

$$u = \begin{cases} 1 & si\ es\ un\ producto \\ -1 & si\ es\ una\ división \end{cases}$$

$$\sum_{i=1*3^u}^{n} k_0 \longrightarrow \sum_{i=a*r^u}^{n} \boldsymbol{k_0 f(i)}; \boldsymbol{T(n) = k_0 * \frac{1}{\ln r} * \begin{cases} \ln n & si\ f(i) = 1 \\ n & e.c.o.c. \end{cases}}$$



Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

# ANÁLISIS DE COMPLEJIDAD

1. ¿Qué es la complejidad de un algoritmo?
2. Órdenes de complejidad
3. Problemas P, NP y NP-completos
4. Resolución de ecuaciones en recurrencia
5. Resolución de sumatorios
6. Estudio de la complejidad de un algoritmo
7. Ejercicios resueltos



Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

## 6.1. Tamaño de un problema

Se define el tamaño $n$ de un problema $p$ como el tiempo necesario para su ejecución, $\mathrm{T}(p)$

$$n = \mathrm{t}(p)$$



Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

## 6.2. Casos de ejecución

*Ejemplo: dada una lista de números enteros, averiguar si al menos uno de ellos es múltiplo de 3*

```
public static boolean hayMultiplo3
        (List<Integer> lista){
        boolean r = false;
        for(Integer e: lista){
                r = e%3==0;
                if(r) break;
        }
        return r;
}
```

Tenemos, pues, **tres casos diferentes de ejecución**, a los que denominamos:

- **Caso Peor, $T^p(n)$**: aquel problema que tarda **más tiempo** en resolverse. En el ejemplo, que no haya ningún número múltiplo de 3 o que sea el última
- **Caso Mejor, $T^m(n)$**: aquel problema que tarda **menos tiempo** en resolverse. En el ejemplo, que el primer número sea múltiplo de 3.
- **Caso Medio, $T^{md}(n)$**: aquel problema que tiene una **distribución de probabilidad** de los problemas, $f(p)$. Viene dado por:

$$T^{md}(n) = \sum_{p \in P | t(p)=n} T(p) f(p)$$



Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

# 6.3. Complejidad de un algoritmo iterativo

**Bloque básico**
Un bloque básico es una secuencia de instrucciones sin ningún salto de control (*if, while*).

Dada la siguiente secuencia de bloques, el tiempo total será la suma de cada uno de dichos bloques:

$$\boldsymbol{s1};$$
$$\boldsymbol{s2};$$
$$\ldots$$
$$\boldsymbol{sk};$$

$$\boldsymbol{T_s(n) = T_{s1}(n) + T_{s2}(n) + \cdots + T_{sk}(n)}$$



Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

# 6.3. Complejidad de un algoritmo iterativo

**Bloque *if***

La estructura de un bloque *if* es de la forma:

```
if(g){
  s1;
}else{
  s2;
}
```

$$T_{if}^{p} = T_{g}^{p} + max(T_{s1}^{p}, T_{s2}^{p})$$

$$T_{if}^{m} = T_{g}^{m} + min(T_{s1}^{m}, T_{s2}^{m})$$

$$T_{if}^{md} = T_{g}^{md} + f_1 T_{s1}^{md} + f_2 T_{s2}^{md})$$

Donde ***g*** es una expresión lógica y ***s*** un bloque de código. Representamos por $T_g$ el tiempo de evaluación de una expresión lógica.



Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

# 6.3. Complejidad de un algoritmo iterativo

**Bloque *while***

La estructura de un bloque *while* es de la forma:

```
while(g){
    s;
}
```

Siendo **g** una expresión lógica, $T_g$ el tiempo de evaluación de dicha expresión, **s** un bloque de código y $T_s$ el tiempo de evaluación de dicho bloque:

$$\boldsymbol{T_w(n) = T_g + \sum_{i \in I} (T_g + T_s(i))}$$



Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

# ANÁLISIS DE COMPLEJIDAD

1. ¿Qué es la complejidad de un algoritmo?
2. Órdenes de complejidad
3. Problemas P, NP y NP-completos
4. Resolución de ecuaciones en recurrencia
5. Resolución de sumatorios
6. Estudio de la complejidad de un algoritmo
7. Ejercicios resueltos



Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

# 7.1. Calentando motores

Calcular los siguientes sumatorios:

$$\sum_{i=1}^{n} i(i+3)$$

$$\sum_{i=1}^{n} (i+1)^3$$

$$\sum_{k=1}^{n} k^2(2k+3)$$



Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

# 7.1. Calentando motores

Calcular los siguientes sumatorios:

$$\sum_{i=1}^{n} i(i+3) = \sum_{i=1}^{n} i^2 + \sum_{i=1}^{n} 3i = \frac{1}{3}n^3 + 3\sum_{i=1}^{n} i = \frac{1}{3}n^3 + 3\frac{1}{2}n^2$$

$$\sum_{i=1}^{n} (i+1)^3$$

$$\sum_{k=1}^{n} k^2(2k+3)$$



Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

# 7.1. Calentando motores

Calcular los siguientes sumatorios:

$$\sum_{i=1}^{n} i(i+3) = \sum_{i=1}^{n} i^2 + \sum_{i=1}^{n} 3i = \frac{1}{3}n^3 + 3\sum_{i=1}^{n} i = \frac{1}{3}n^3 + 3\frac{1}{2}n^2$$

$$\sum_{i=1}^{n} (i+1)^3 = \sum_{i=1}^{n} i^3 + 3i + 3i^2 + 1 = \sum_{i=1}^{n} i^3 + \sum_{i=1}^{n} 3i + \sum_{i=1}^{n} 3i^2 + \sum_{i=1}^{n} 1 = \frac{1}{4}n^4 + 3n + 3\frac{1}{3}n^3 + n = \frac{1}{4}n^4 + n^3 + 4n$$

$$\sum_{k=1}^{n} k^2(2k+3)$$



Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

# 7.1. Calentando motores

Calcular los siguientes sumatorios:

$$\sum_{i=1}^{n} i(i+3) = \sum_{i=1}^{n} i^2 + \sum_{i=1}^{n} 3i = \frac{1}{3}n^3 + 3\sum_{i=1}^{n} i = \frac{1}{3}n^3 + 3\frac{1}{2}n^2$$

$$\sum_{i=1}^{n} (i+1)^3 = \sum_{i=1}^{n} i^3 + 3i + 3i^2 + 1 = \sum_{i=1}^{n} i^3 + \sum_{i=1}^{n} 3i + \sum_{i=1}^{n} 3i^2 + \sum_{i=1}^{n} 1 = \frac{1}{4}n^4 + 3n + 3\frac{1}{3}n^3 + n = \frac{1}{4}n^4 + n^3 + 4n$$

$$\sum_{k=1}^{n} k^2(2k+3) = \sum_{k=1}^{n} 2k^3 + 3k^2 = 2\sum_{k=1}^{n} k^3 + 3\sum_{k=1}^{n} k^2 = 2\frac{1}{4}n^4 + 3\frac{1}{3}n^3 = \frac{1}{2}n^4 + n^3$$



Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

# 7.2. Problemas de bucles iterativos (I)

Dado el siguiente bucle

```
for (int i = 1; i <= n - 1; i++) {
      for (int j = i + 1; j <= n; j++) {
            for (int k = 1; k <= j; k++) {
                  r = r + 2;
            }
      }
}
```

Calcular su complejidad



Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

# 7.2. Problemas de bucles iterativos (I)

Dado el siguiente bucle

```
for (int i = 1; i <= n - 1; i++) {
      for (int j = i + 1; j <= n; j++) {
            for (int k = 1; k <= j; k++) {
                  r = r + 2;
            }
      }
}
```

Calcular su complejidad

$$\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\sum_{k=1}^{j} c_0 = c_0 \sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\sum_{k=1}^{j} 1 = c_0 \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} j =$$

$$= c_0 \sum_{i=1}^{n-1} \frac{1}{2} n^2 = c_0 \frac{1}{2} \sum_{i=1}^{n-1} n^2 = c_0 \frac{1}{2} n^2 \sum_{i=1}^{n-1} 1 = c_0 \frac{1}{2} n^2 n = \frac{1}{2} n^3$$

$$\boldsymbol{T(n) \in \Theta(n^3)}$$



Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

# 7.2. Problemas de bucles iterativos (II)

Dado el siguiente bucle

```
for (i = 1; i < n; i++) {
      for (j = 1; j < i * i; j++) {
            z++;
      }
}
```

Calcular su complejidad



Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

# 7.2. Problemas de bucles iterativos (II)

Dado el siguiente bucle

```
for (i = 1; i < n; i++) {
      for (j = 1; j < i * i; j++) {
            z++;
      }
}
```

Calcular su complejidad

$$\sum_{i=1}^{n-1}\sum_{j=1}^{i^2} c_0 = c_0 \sum_{i=1}^{n-1}\sum_{j=1}^{i^2} 1 = c_0 \sum_{i=1}^{n-1} i^2 = \frac{1}{3}n^3$$

$$\boldsymbol{T(n) \in \Theta(n^3)}$$



Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

# 7.2. Problemas de bucles iterativos (III)

Dado el siguiente bucle

```
i = n;
while (i >= 1) {
      j = 1;
      while (j <= i ) {
            s = s + j;
            j = j + 2;
      }
      i = i / 3;
}
```

Calcular su complejidad



Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

# 7.2. Problemas de bucles iterativos (III)

Dado el siguiente bucle

```
i = n;
while (i >= 1) {
      j = 1;
      while (j <= i ) {
            s = s + j;
            j = j + 2;
      }
      i = i / 3;
}
```

Calcular su complejidad

$$\sum_{i=1*3^{u}}^{n}\sum_{j=1}^{i} j+2 = \sum_{i=1*3^{-1}}^{n}\frac{1}{2}n = \frac{1}{2}n\sum_{i=1*\frac{1}{3}}^{n}1 = \frac{1}{2}n\frac{1}{\ln {}^{1}/_{3}}\ln n$$

$$\boldsymbol{T(n) \in \Theta(\mathrm{n}\, log\, n)}$$



Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

# 7.3. Problemas de algoritmos recursivos e iterativos (I)

Dado el siguiente algoritmo

```
double f (int n, double a) {
	double r;
	if (n == 1) {
		r = a;
	} else {
		r = f (n/2, a+1) - f (n/2, a-1);
		for (int i = 1; i <= n; i++) {
			r += a * i;
		}
	}
	return r;
}
```

Calcular su complejidad



Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

# 7.3. Problemas de algoritmos recursivos e iterativos (I)

Dado el siguiente algoritmo

```
double f (int n, double a) {
        double r;
        if (n == 1) {
                r = a;
        } else {
                r = f (n/2, a+1) - f (n/2, a-1);
                for (int i = 1; i <= n; i++) {
                        r += a * i;
                }
        }
        return r;
}
```

Calcular su complejidad

Tamaño del problema = $n$

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{2}\right)$$

$$\sum_{i=1}^{n} 1 = n$$

$$\boldsymbol{T_{total}}\,(\boldsymbol{n}) = 2T\left(\frac{n}{2}\right) + n$$

$\boldsymbol{T^{m}}\,(\boldsymbol{n}) \in \boldsymbol{\Theta(1)}$ **(cuando n == 1 en la inicialización)**
$\boldsymbol{T^{p}}\,(\boldsymbol{n}) \in \boldsymbol{\Theta(\mathrm{n}\,log\,n)}$ **e.c.o.c**



Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

# 7.3. Problemas de algoritmos recursivos e iterativos (II)

Dado el siguiente algoritmo

```
int f (int n) {
      int i, j, z, r;
      if (n < 1) {
            r = 1;
      } else {
            z = 0;
            for (i = 1; i < n; i++) {
                  for (j = 1; j < i * i; j++) {
                        z++;
                  }
            }
            r = z * (( f (n – 2)) ^ 2);
      }
      return r;
}
```

Calcular su complejidad



Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

# 7.3. Problemas de algoritmos recursivos e iterativos (II)

Dado el siguiente algoritmo

```
int f (int n) {
        int i, j, z, r;
        if (n < 1) {
                r = 1;
        } else {
                z = 0;
                for (i = 1; i < n; i++) {
                        for (j = 1; j < i * i; j++) {
                                z++;
                        }
                }
                r = z * (( f (n - 2)) ^ 2);
        }
        return r;
}
```

Calcular su complejidad

Tamaño del problema = $n$

$$\sum_{i=1}^{n-1} \sum_{j=1}^{(i*i)-1} c_0 = c_0 \sum_{i=1}^{n-1} i^2 \approx n^3$$

$$T(n) = T(n-2)$$

$$\boldsymbol{T_{total}(n)} = T(n-2) + n^3$$

$\boldsymbol{T^m(n) \in \Theta(1)}$ **(cuando n < 1 en la inicialización)**

$\boldsymbol{T^p(n) \in \Theta(n^4)}$



Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

# 7.3. Problemas de algoritmos recursivos e iterativos (III)

Dado el siguiente algoritmo

```
int f (int a, b) {
          int r;
          if (a == 0 || b == 0) {
                    r = 1;
          } else {
                    r = 2 * f (a-1, b-1);
          }
          return r;
}
```

Calcular su complejidad



Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

# 7.3. Problemas de algoritmos recursivos e iterativos (III)

Dado el siguiente algoritmo

```
int f (int a, b) {
        int r;
        if (a == 0 || b == 0) {
                r = 1;
        } else {
                r = 2 * f (a-1, b-1);
        }
        return r;
}
```

Calcular su complejidad

Tamaño del problema = $n = \min(a, b)$

$T(n) = T(n-1)$

$\boldsymbol{T_{total}(n) = n}$

$\boldsymbol{T^m(n) \in \Theta(1)}$ **(cuando a == 0 o b == 0 en la inicialización)**
$\boldsymbol{T^p(n) \in \Theta(n)}$



Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

# 7.3. Problemas de algoritmos recursivos e iterativos (IV)

Dado el siguiente algoritmo

```
int f (int [] a, int n) {
          return g (a, n, 0);
}

int g (int [] a, int n, int i) {
          int s;
          if (n-i==0) {
                    s = r;
          } else {
                    r += a[i];
                    s = g (a,i+1);
          }
          return s;
}
```

Calcular su complejidad



Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

# 7.3. Problemas de algoritmos recursivos e iterativos (IV)

Dado el siguiente algoritmo

```
int f (int [] a, int n) {
        return g (a, n, 0);
}

int g (int [] a, int n, int i) {
        int s;
        if (n-i==0) {
                s = r;
        } else {
                r += a[i];
                s = g (a,i+1);
        }
        return s;
}
```

Calcular su complejidad

Tamaño del problema = $m = n - i$

$T(m) = T(m-1)$

$\boldsymbol{T_{total}(m) = m}$

$\boldsymbol{T^m(m) \in \Theta(1)}$ **(cuando n-i== 0 en la inicialización)**
$\boldsymbol{T^p(m) \in \Theta(m)}$



Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)

# BIBLIOGRAFÍA



*José A. Mañas*
*Análisis de Algoritmos – Complejidad*
*Departamento de Ingeniería de Sistemas Telemáticos*
*Universidad Politécnica de Madrid, febrero de 2017*

*Miguel Toro*
*“Tema 11. Complejidad”, del libro “Introducción a la programación”*
*Departamento de Lenguajes y Sistemas Informáticos*
*Universidad de Sevilla*

*“Tema 2. Análisis de la eficiencia de algoritmos”, de la asignatura “Análisis y diseño de datos y algoritmos”*
*Departamento de Lenguajes y Sistemas Informáticos*
*Escuela Técnica Superior de Ingeniería Informática*
*Universidad de Sevilla, 2015*



Atribución-NoComercial-CompartirIgual
4.0 Internacional (CC BY-NC-SA 4.0)