1️⃣

Complejidad

Teoría de la complejidad algorítmica

Debemos saber que una solución es un conjunto único, pero no es el único conjunto de pasos que entregan la solución, existen muchas alternativas de solución y estas alternativas pueden diferir por: Número

https://m.monografias.com/trabajos27/complejidad-algoritmica/complejidad-algoritmica.shtml

Complejidad en Tiempo y Espacio

Coste de un algoritmo

➡️ **Complejidad computacional**

Coste requerido para encontrar la solución a un problema, en términos de recursos computacionales: **tiempo** y **espacio** (memoria).

📖 El coste computacional de una instrucción I de un algoritmo se representa como $T(n)$, siendo $\mathbf{n}$ **el valor**

📝 Salvo que se especifique, el *coste computacional* hace referencia al

de la entrada.	tiempo.

Tipo de Instrucción	Coste computacional
I_{simple}	$T_{\text{simple}}(n) = 1$
$I_{selección}$	$T_{\text{selección}}(n) = T_{\text{expresión}}(n) + \max(T_1(n), \ldots, T_n(n))$
$I_{bucle-fijo}$	$T_{bucle-fijo}(n) = k, \quad k \in \mathbb{R}$
Composición (I_1 y I_2)	$T_{1,2}(n) = T_1(n) + T_2(n)$

Ejemplo (diapositiva 13)

Asumiendo que el método `funcion()` tiene un coste T, ¿cuál es el coste computacional de este código?

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        funcion();
    }
}
```

Como hay 2 `for()` que iteran hasta n, cada uno se repite n veces.

El bloque `funcion()` tiene coste T y está dentro del segundo `for()`, por tanto:

$$\text{coste}(n) = n \cdot n \cdot T = \mathbf{Tn^2}$$

Cada `for()` incluye varios costes de instrucciones simples (`int i = 0`, `i < n`, `i++`...), sabiendo que esos costes son constantes ($1 + 1 + 2...$), acaban siendo despreciables, dando lugar a un coste de n por cada bucle.

Finalmente, el bucle interior contiene la función de coste T, este se repite por cada bucle (cuyo coste ya se conoce).

Ejemplo (diapositiva 14)

Asumiendo que el método `funcion()` tiene un coste T, ¿cuál es el coste computacional de este código?

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < i; j++) {
        funcion();
    }
}
```

Hay 2 `for()`: uno itera hasta n y el otro itera hasta i.

El bloque `funcion()` tiene coste T y está dentro del segundo `for()`, por tanto:

$$\text{coste}(n) = n \cdot n \cdot T = \mathbf{Tn^2}$$

Complejidad

➡️ **Función de Complejidad**

Función que asocia cada entrada de un algoritmo a su tiempo de ejecución.

$$T_A : \mathbb{N} \longrightarrow \mathbb{N}$$

💡 Una cuestión importante es **cómo medir la entrada** de un algoritmo.

Ejemplo (Complejidad 1, diapositiva 18)

Problema	Entrada
Búsqueda en una lista	Número de elementos de la lista.
Evaluar un polinomio en un punto	Grado del polinomio.
Multiplicar matrices cuadradas ($n \times n$)	Dimensión n de la matriz.
Multiplicar matrices cuadradas ($n \times n$)	Número de elementos de la matriz.

💡 Generalmente puede usarse el **número de bits de la representación binaria** de la entrada.

❗ Los algoritmos con una **complejidad exponencial** solo son prácticos para resolver problemas de **tamaño pequeño**.

Clases de complejidad y notación asintótica

📖 $\mathrm{O}\Big(g(n)\Big)$ conjunto de funciones acotadas superiormente por un $\dot{g}$.

$$\mathrm{O}\Big(g(n)\Big) = \{f : \mathbb{N} \longrightarrow \mathbb{N} \mid \exists k \geq 0;\ c > 0;\ \forall n \geq k \cdot f(n) \leq c \cdot g(n)\}$$

📖 $\Theta\Big(g(n)\Big)$ conjunto de funciones con el mismo orden de complejidad que g.

$$\Theta\Big(g(n)\Big) = \{f : \mathbb{N} \longrightarrow \mathbb{N} \mid \exists k \geq 0;\ c_1, c_2 > 0;\ \forall n \geq k \cdot c_1 \cdot f(n) \leq f(n) \leq c_2 \cdot g(n)\}$$

📖 $\Omega\Big(g(n)\Big)$ conjunto de funciones acotadas inferiormente por un $\dot{g}$.

$$\Omega\Big(g(n)\Big) = \{f : \mathbb{N} \longrightarrow \mathbb{N} \mid \exists k \geq 0;\ c > 0;\ \forall n \geq k \cdot f(n) \geq c \cdot g(n)\}$$

Ejemplo (personal)

Complejidades	$\Omega(\log(n))$	$\Theta(n)$	$\mathrm{O}(n^2)$
$n^2 + 10$	✅	❌	✅
$5 \cdot \log(n)$	✅	❌	✅
$256n^3 + n^2$	✅	❌	❌
$1024 + 64n$	✅	✅	✅
$n^2(n^2 - 1) + 1$	✅	❌	❌
$\frac{1}{2}n(n-1)$	✅	❌	✅

Principio de invarianza

📖 Dos implementaciones del mismo algoritmo **solo diferirán en una constante multiplicativa**.

Sean $T_1(n)$ y $T_2(n)$ los tiempos de dos implementaciones del algoritmo A, entonces:

$$\exists a, b \in \mathbb{R} \quad | \quad T_1(n) \leq a \cdot T_2(n) \quad \wedge \quad T_2(n) \leq b \cdot T_1(n)$$

Otra forma de verlo es:

$$T_1(n) \in \Theta\Big(T_2(n)\Big) \quad \wedge \quad T_2(n) \in \Theta\Big(T_1(n)\Big)$$

💡 Gracias a este principio, **los costes de las instrucciones pasan a ser poco relevantes** de cara a la complejidad final de un algoritmo.

> 📝 Esto significa que al establecer los costes constantes de las instrucciones al analizar un código, los valores asignados no afectan a la solución final.

Propiedades

$$f_1(n) \in \mathrm{O}\Big(g_1(n)\Big), \quad f_2(n) \in \mathrm{O}\Big(g_2(n)\Big) \quad \longrightarrow \quad f_1(n) + f_2(n) \in \mathrm{O}\Bigg(\max\Big(g_1(n), g_2(n)\Big)\Bigg)$$

$$l = \lim_{n \to \infty} \frac{f(n)}{g(n)} = \begin{cases} l < \infty & \Longleftrightarrow \quad f(n) \in \mathrm{O}\Big(g(n)\Big) \\ 0 < l < \infty & \Longleftrightarrow \quad f(n) \in \Theta\Big(g(n)\Big) \\ 0 < l & \Longleftrightarrow \quad f(n) \in \Omega\Big(g(n)\Big) \end{cases}$$

$$\mathrm{O}\Big(\log_a(n)\Big) = \mathrm{O}\Big(\log_b(n)\Big), \quad a, b \in \mathbb{R}$$

🧠 Propiedad de los logaritmos

$$a^{\log_x(b)} \equiv b^{\log_x(a)}$$

> 💡 Por esto **no es necesario especificar la base** de un $\log(x)$ en $\mathrm{O}(\ldots)$.

Comparativa de complejidades

Los comportamientos asintóticos de aparición más frecuente pueden ordenarse de menor a mayor crecimiento de la siguiente forma:

$$1 < \log(n) < n < n \cdot \log(n) < n^k < k^n < n!$$

$\mathrm{O}(1)$	Constante	*No depende del tamaño del problema.*
$\mathrm{O}(\log(n))$	Logarítmica	Búsqueda binaria.

$\mathrm{O}(n)$	Lineal	Búsqueda lineal.
$\mathrm{O}(n \cdot \log(n))$	Casi lineal	Quick-sort.
$\mathrm{O}(n^2)$	Cuadrática	Algoritmo de la burbuja.
$\mathrm{O}(n^3)$	Cúbica	Producto de matrices.
$\mathrm{O}(n^k), \quad k > 3$	Polinómica	
$\mathrm{O}(k^n), \quad k > 1$	Exponencial	Algunos algoritmos de grafos.
$\mathrm{O}(n!)$	Factorial	

Algoritmos de ordenación y su complejidad

Estrategias y fórmulas útiles

Cómo analizar la eficiencia en tiempo de algoritmos

1. Decidir cómo medir el tamaño de entrada.
2. Identificar las operaciones básicas.
3. Comprobar si el número de veces que se ejecutan las instrucciones simples dependen del tamaño de la entrada.
    - Si hay otras condiciones a tener en cuenta, habrá que estudiar las complejidades en el mejor y peor caso.
4. Establecer una suma que defina el número de veces que se ejecutan las operaciones básicas.
5. Reducir las sumas y establecer la complejidad asintótica.

🧠 Propiedades de los sumatorioste

$$\sum_i (kx_i) = k \sum_i (x_i)$$

$$\sum_i (x_i + y_i) = \sum_i (x_i) + \sum_i (y_i)$$

$$\sum_i^n (x) = x(n - i)$$

$$\sum_{i=0}^n (x^i) = \frac{x^{n+1} - 1}{x - 1}$$

$$\sum_{i=0}^n (i^k) = \sum_{i=1}^n (i^k) = \frac{n(n+1)}{2} = \frac{n^{k+1}}{k+1}$$

Análisis de algoritmos no-recursivos

Ordenación por inserción

Ejemplo (Complejidad 2, diapositiva 2)

Un algoritmo que ordena un array de números de menor a mayor, por inserción.

```
/**
 * @param a     Array de enteros a ordenar
 *
 */
public static void insercion(int[] a) {
    for (int i = 1; i < a.length(); i++) {

      // a[0 .. (i-1)]
      int j = i-1;
      int x = a[i];

      // Insertar a[i] en a[0 .. (i-1)] para
      // que a[0 .. i] quede ordenado
      while (j >= 0 && a[j] > x) {
        a[j+1] = a[j];
        j--;
      }

      a[j+1] = x;
```

Vector de entrada:

`6 2 8 3 5 7 9`

Verde: ordenado.

Rojo: desordenado.

Amarillo: iteración actual.

1. `6` **`2`** `8 3 5 7 9`
2. `2 6` **`8`** `3 5 7 9`
3. `2 6 8` **`3`** `5 7 9`
4. `2 3 6 8` **`5`** `7 9`

```
    }
}
```

5. `2 3 5 6 8 7 9`
6. `2 3 5 6 7 8 9`
7. `2 3 5 6 7 8 9`

Tamaño de la entrada

Tamaño del array: n

Operaciones elementales

Asignación y comparación de elementos del array.

Función $T(n)$ para el código presentado:

$$T(n) = \underbrace{1}_{\texttt{int i = 1}} + \underbrace{\sum_{i=1}^{n-1}\left(\underbrace{1}_{\texttt{i < n}} + \underbrace{2}_{\texttt{i++}} + \underbrace{\sum_{j=i-1}^{n}\left(\underbrace{1}_{\texttt{a[j+1] = a[j]}} + \underbrace{2}_{\texttt{j--}}\right)}_{\texttt{while()}}\right)}_{\texttt{for()}} + \underbrace{1}_{\texttt{i < n}}$$

📝 He tratado de ser lo más atómico posible al asignar los valores. No obstante:

> 📝 Esto significa que al establecer los costes constantes de las instrucciones al analizar un código, los valores asignados no afectan a la solución final.

▼ **Mejor caso**

El array empieza ordenado.

- El bucle `while()` no se ejecuta nunca.
- El bucle `for()` se ejecuta una vez (iteración #0).

$$\begin{aligned} T(n) &= 1 + \sum_{i=1}^{n-1}\left(1 + 2 + \sum_{j=i-1}^{(0)}(1+2)\right) + 1 &= \\ &= 2 + \sum_{i=1}^{n-1}(3+0) &= \\ &= 2 + 3(n-1) \end{aligned}$$

$$\mathbf{T(n) = 3(n-1) + 2}, \quad \mathbf{T(n) \in \Theta(n)}$$

▼ **Peor caso**

El array está invertido.

- El bucle `while()` se ejecuta siempre.
- El bucle `for()` se ejecuta $n-1$ veces (por `i < a.length()`).

$$\begin{aligned} T(n) &= 1 + \sum_{i=1}^{n-1}\left(1 + 2 + \sum_{j=i-1}^{0}(1+2)\right) + 1 &= \\ &= \dots &= \\ &= n^2 + 2n - 3 \end{aligned}$$

$$\mathbf{T(n) = n^2 + 2n - 3}, \quad \mathbf{T(n) \in \Theta(n^2)}$$

▼ **Caso medio**

La mitad está ordenado.

- El bucle `while()` se ejecutaría $\frac{i-1}{2}$ de veces.
- El bucle `for()` se ejecutaría siempre.

$$\begin{aligned}
T(n) &= & 1+\sum_{i=1}^{n-1}\left(1+2+\sum_{j=i-1}^{\frac{i-1}{2}}(1+2)\right)+1 &= \\
&= & 2+\sum_{i=1}^{n-1}\left(3+\sum_{j=i-1}^{\frac{i-1}{2}}(3)\right) &= \\
&= & 2+\sum_{i=1}^{n-1}\left(3+3\left(\frac{i-1}{2}-(i-1)\right)\right) &= \\
&= & 2+\sum_{i=1}^{n-1}(3)+3\sum_{i=1}^{n-1}\left(\frac{i-1}{2}-(i-1)\right) &= \\
&= & 2+3(n-1)+\tfrac{3}{2}\sum_{i=1}^{n-1}(-i+1) &= \\
&= & 2+3(n-1)-\tfrac{3}{2}\sum_{i=1}^{n-1}(i)+\sum_{i=1}^{n-1}(1) &= \\
&= & 2+3(n-1)-\tfrac{3}{2}i(n-1)+\left(\tfrac{(n-1)((n-1)-1)}{2}\right) &= \\
&= & [\text{constante}]+[n]+[n^2+n]
\end{aligned}$$

$$\mathbf{T(n)=\frac{1}{2}(n^2+3n-4),\quad T(n)\in\Theta(n)}$$

Ordenación por selección

Ejemplo (Complejidad 2, diapositiva 7)

Un algoritmo que ordena un array de números de menor a mayor, por selección.

```
/**
 * @param a     Array de enteros a ordenar
 *
 */
public static void seleccion(int[] a) {
    for (int i = 0; i < a.length()-1; i++) {

        // a[0 .. (i-1)]
        int min = i;

        for (int j= i + 1; j < a.length(); j++) {

            // min es el menor elemento
            if (a[j] < a[min]) {
                min = j;
            }
        }

        // Intercambiar el menor
        int temp = a[i];

        a[i] = a[min];
        a[min] = temp;
    }
}
```

Vector de entrada:

`6 2 8 3 5 7 9`

Verde: ordenado.

Rojo: desordenado.

Amarillo: iteración actual.

1. `6 2 8 3 5 7 9`
2. `2 6 8 3 5 7 9`
3. `2 3 8 6 5 7 9`
4. `2 3 5 6 8 7 9`
5. `2 3 5 6 8 9 7`
6. `2 3 5 6 7 9 8`
7. `2 3 5 6 7 8 9`

Tamaño de la entrada

Tamaño del array: n

Operaciones elementales

Asignación y comparación de elementos del array.

Función $T(n)$ para el código presentado:

$$T(n) = \underbrace{1}_{\texttt{int i = 0}} + \underbrace{\sum_{i=0}^{n-2}\left(\underbrace{1}_{\texttt{i < n-1}} + \underbrace{2}_{\texttt{i++}} + \underbrace{1}_{\texttt{int min = i}} + \underbrace{\sum_{j=i+1}^{n}\left(\underbrace{1}_{\texttt{a[j+1] = a[j]}} + \underbrace{2}_{\texttt{j--}}\right)}_{\texttt{while()}}\right)}_{\texttt{for()}} + \underbrace{1}_{\texttt{i < n}}$$

▼ **Mejor caso**

El array empieza ordenado.

- El bucle `while()` no se ejecuta nunca.
- El bucle `for()` se ejecuta una vez (iteración #0).

$$\begin{aligned} T(n) &= 1 + \sum_{i=1}^{(6)-1}\left(1 + 2 + \sum_{j=i-1}^{n}(1+2)\right) + 1 &= \\ &= \ldots &= \\ &= 3(n-1) \end{aligned}$$

$$\mathbf{T(n) = 3(n-1)}, \quad \mathbf{T(n) \in \Theta(n)}$$

▼ **Peor caso**

El array está invertido.

- El bucle `while()` se ejecuta siempre.
- El bucle `for()` se ejecuta $n-1$ veces (por `i < a.length()`).

$$\begin{aligned} T(n) &= 1 + \sum_{i=1}^{n-1}\left(1 + 2 + \sum_{j=i-1}^{0}(1+2)\right) + 1 &= \\ &= \ldots &= \\ &= n^2 + 2n - 3 \end{aligned}$$

$$\mathbf{T(n) = n^2 + 2n - 3}, \quad \mathbf{T(n) \in \Theta(n^2)}$$

▼ **Caso medio**

La mitad está ordenado.

- El bucle `while()` se ejecutaría $\frac{i-1}{2}$ de veces.
- El bucle `for()` se ejecutaría siempre.

$$
\begin{aligned}
T(n) &= 1+\sum_{i=1}^{n-1}\left(1+2+\sum_{j=i-1}^{\frac{i-1}{2}}(1+2)\right)+1 &=\\
&= 2+\sum_{i=1}^{n-1}\left(3+\sum_{j=i-1}^{\frac{i-1}{2}}(3)\right) &=\\
&= 2+\sum_{i=1}^{n-1}\left(3+3\left(\frac{i-1}{2}-(i-1)\right)\right) &=\\
&= 2+\sum_{i=1}^{n-1}(3)+3\sum_{i=1}^{n-1}\left(\frac{i-1}{2}-(i-1)\right) &=\\
&= 2+3(n-1)+\tfrac{3}{2}\sum_{i=1}^{n-1}(-i+1) &=\\
&= 2+3(n-1)-\tfrac{3}{2}\sum_{i=1}^{n-1}(i)+\sum_{i=1}^{n-1}(1) &=\\
&= 2+3(n-1)-\tfrac{3}{2}i(n-1)+\left(\tfrac{(n-1)((n-1)-1)}{2}\right) &=\\
&= [\text{constante}]+[n]-[n]+[n^2+n]
\end{aligned}
$$

$$
\mathbf{T(n)=\frac{1}{2}(n^2+3n-4),\quad T(n)\in\Theta(n)}
$$

Ordenación por el método de la burbuja

Ejemplo (Complejidad 2, diapositiva 9)

Un algoritmo que ordena un array de números de menor a mayor, por el método de la burbuja.

```
/**
 * Mejora del algoritmo de selección, en el que
 * cada iteración del bucle interno se mueven
 * todos los datos adyacentes desordenados.
 *
 * @param a     Array de enteros a ordenar
 */
public static void burbuja(int[] a) {
    for (int i = 0; i < a.length()-1; i++) {

        // 'a[0 .. (i-1)]' está ordenado
        for (int j= a.lenght()-1; j > i; j--) {

            // 'a[j]' es el menor elemento
            // del array 'a[j .. (N-1)]'
            if (a[j] < a[j-1]) {

                // Intercambiar el menor
                int temp = a[j];

                a[j] = a[j-1];
                a[j-1] = temp;
            }
        }
    }
}
```

Vector de entrada:

6 2 8 3 5 9 7

Verde: ordenado.

Rojo: desordenado.

Amarillo: iteración actual.

1. 6 **2** 8 3 5 7 9
2. 2 6 8 **3** 5 7 9
3. 2 3 8 6 **5** 7 9
4. 2 3 5 **6** 8 7 9
5. 2 3 5 6 8 9 **7**
6. 2 3 5 6 7 9 **8**
7. 2 3 5 6 7 8 9

Tamaño de la entrada

Tamaño del array: n

Operaciones elementales

Asignación y comparación de elementos del array.

Función $T(n)$ para el código presentado:

$$
T(n)=\underbrace{1}_{\texttt{int i = 0}}+\underbrace{\sum_{i=0}^{n-2}\left(\underbrace{1}_{\texttt{i < n-1}}+\underbrace{2}_{\texttt{i++}}+\underbrace{1}_{\texttt{int min = i}}+\underbrace{\sum_{j=i+1}^{n}\left(\underbrace{1}_{\texttt{a[j+1] = a[j]}}+\underbrace{2}_{\texttt{j--}}\right)}_{\texttt{while()}}\right)}_{\texttt{for()}}+\underbrace{1}_{\texttt{i < n}}
$$

▼ Mejor caso

El array empieza ordenado.

- El bucle `while()` no se ejecuta nunca.
- El bucle `for()` se ejecuta una vez (iteración #0).

$$\begin{aligned} T(n) &= 1 + \sum_{i=1}^{(6)-1} \left(1 + 2 + \sum_{j=i-1}^{n} (1+2) \right) + 1 &= \\ &= \dots &= \\ &= 3(n-1) \end{aligned}$$

$$\mathbf{T(n) = 3(n-1)}, \quad \mathbf{T(n) \in \Theta(n)}$$

▼ Peor caso

El array está invertido.

- El bucle `while()` se ejecuta siempre.
- El bucle `for()` se ejecuta $n-1$ veces (por `i < a.length()`).

$$\begin{aligned} T(n) &= 1 + \sum_{i=1}^{n-1} \left(1 + 2 + \sum_{j=i-1}^{0} (1+2) \right) + 1 &= \\ &= \dots &= \\ &= n^2 + 2n - 3 \end{aligned}$$

$$\mathbf{T(n) = n^2 + 2n - 3}, \quad \mathbf{T(n) \in \Theta(n^2)}$$

▼ Caso medio

La mitad está ordenado.

- El bucle `while()` se ejecutaría $\frac{i-1}{2}$ de veces.
- El bucle `for()` se ejecutaría siempre.

$$\begin{aligned} T(n) &= 1 + \sum_{i=1}^{n-1} \left(1 + 2 + \sum_{j=i-1}^{\frac{i-1}{2}} (1+2) \right) + 1 &= \\ &= 2 + \sum_{i=1}^{n-1} \left(3 + \sum_{j=i-1}^{\frac{i-1}{2}} (3) \right) &= \\ &= 2 + \sum_{i=1}^{n-1} \left(3 + 3 \left(\frac{i-1}{2} - (i-1) \right) \right) &= \\ &= 2 + \sum_{i=1}^{n-1} (3) + 3 \sum_{i=1}^{n-1} \left(\frac{i-1}{2} - (i-1) \right) &= \\ &= 2 + 3(n-1) + \tfrac{3}{2} \sum_{i=1}^{n-1} (-i+1) &= \\ &= 2 + 3(n-1) - \tfrac{3}{2} \sum_{i=1}^{n-1} (i) + \sum_{i=1}^{n-1} (1) &= \\ &= 2 + 3(n-1) - \tfrac{3}{2} i(n-1) + \left(\tfrac{(n-1)((n-1)-1)}{2} \right) &= \\ &= [\text{constante}] + [n] - [n] + [n^2 + n] \end{aligned}$$

$$\mathbf{T(n) = \frac{1}{2}(n^2 + 3n - 4)}, \quad \mathbf{T(n) \in \Theta(n)}$$

Algoritmos recursivos básicos y su complejidad

Factorial

Ejemplo (Complejidad 2, diapositiva 14)

```
/**
 * @param n     Número natural
 * @return n!   1·2·(...)·n
 */
public static int factorial(int n) {
    if (n < 0) {
        throw new IllegalArgumentException("Número no válido.");
    }

    if (n == 0) {
        return 1;

    } else {
        return n * factorial(n - 1);
    }
}
```

Tamaño de la entrada

Un número natural n.

Operaciones elementales

Multiplicación de números.

Función de recurrencia $T(n)$ para el código presentado:

$$T(n) = \begin{cases} \underbrace{1}_{\texttt{return 1}} & \text{si } \; n = 0 \\ \underbrace{T(n-1)}_{\texttt{factorial(n-1)}} \quad + \quad \underbrace{1}_{\texttt{return n * factorial(n-1)}} & \text{si } \; n \geq 1 \end{cases}$$

Torres de Hanoi

Ejemplo (Complejidad 2, diapositiva 19)

```
/**
 * @param n     Número de discos
 * @param a     Varilla origen
 * @param b     Varilla intermedia
 * @param c     Varilla destino
 */
public static void hanoi(int n, char a, char b, char c) {
    if (n > 0) {
        hanoi(n-1, a, c, b);
        System.out.println("Un disco pasa de " + a + " a " + c + ".");
        hanoi(n-1, b, a, c);

    } else {
        throw new IllegalArgumentException("Número no válido.");
    }
}
```

Tamaño de la entrada

El número de discos: n

Operaciones elementales

Pasar un disco de una varilla a otra.

Función de recurrencia $T(n)$ para el código presentado:

$$T(n) = \begin{cases} \overbrace{0}^{\texttt{IllegalArgumentException}} & \text{si } n = 0 \\ \underbrace{T(n-1)}_{\texttt{hanoi(n-1,a,c,b)}} + \underbrace{1}_{\texttt{System.out}} + \underbrace{T(n-1)}_{\texttt{hanoi(n-1,b,a,c)}} & \text{si } n \geq 1 \end{cases}$$

$$T(n) = \begin{cases} 0 & \text{si } n = 0 \\ 2T(n-1) + 1 & \text{si } n \geq 1 \end{cases}$$

Fibonacci

Ecuaciones de recurrencia

Método del polinomio característico

➡️ **Ecuación de recurrencia general**

Definición:

Sea $p(n)$ un polinomio de grado d_j y los valores $a_i, b_j \in \mathbb{R},\ \forall i \in [1, k],\ \forall j \in [1, m]$.

$$T(n) = a_1 T(n-1) + \dots + a_k T(n-k) + b_1^n p_1(n) + \dots + b_m^n p_m(n)$$

La forma de la solución se obtendrá usando:

➡️ **Polinomio característico general**

Definición:

$$(x^k - a_1 x^{k-1} - \dots - a_{k-1} x^1 - a_k)(x - b_1)^{d_1+1} \cdot \ldots \cdot (x - b_m)^{d_m+1} = 0$$

📝 El polinomio característico **no es la solución**, es la herramienta que la genera.

➡️ **Término dominante de una ecuación de recurrencia**

Aquel cuya complejidad asociada sea la más elevada.

🧠 **Raíz de un polinomio**

Sea $p(x)$ un polinomio, una raíz es un valor v tal que $p(v) = 0$.

$$p(x) : (x - a)(x + b)^c = 0$$

Los valores a y $-b$ son raíces de $p(x)$, porque $p(a) = p(-b) = 0$.

- a con multiplicidad 1; y b con multiplicidad c.

Ejemplo (Complejidad 2, diapositivas 28-29)

Obtener el polinomio característico de las siguientes ecuaciones.

Supongamos que tenemos la ecuacion $T(n) = 2T(n-1) + 2^n$, entonces el polinomio caracteristico es:

$$(x-2)(x-2) = 0$$

por lo que $x = 2$ es una raiz doble del polinomio y las soluciones son de la forma:

$$T(n) = a2^n + bn2^n$$

Supongamos que tenemos la ecuacion $T(n) = 5T(n-1) + 2^n + n3^n$, entonces el polinomio caracteristico es:

$$(x-5)(x-2)(x-3)^2 = 0$$

por lo que $x = 5, x = 2, x = 3$ son las raices del polinomio y las soluciones son de la forma:

$$T(n) = a5^n + b2^n + c3^n + dn3^n$$

Desarrollo

📝 Ejemplo del desarrollo completo y detallado en el Ejercicio 2 de la Relación.

Los ejemplos de los pasos siguientes son independientes entre sí, no describen un problema común.

1. **Pasar de la ecuación recursiva al polinomio característico.**

   Ejemplo

   Hallar el polinomio característico de $M(n) = 3M(n-2) - M(n-4) - 2M(n-6)$.

   $$M(n) - 3M(n-2) + M(n-4) + 2M(n-6) = 0$$

   Como la llamada más profunda es $M(n-6)$, se necesitarán 7 términos: $\alpha_0, \dots, \alpha_6$.

   $$\underbrace{a_0x^6}_{1\cdot M(n)} + \underbrace{a_1x^5}_{0\cdot M(n-1)} + \underbrace{a_2x^4}_{-3\cdot M(n-2)} + a_3x^3 + a_4x^2 + a_5x^1 + \underbrace{a_6x^0}_{-2\cdot M(n-6)} = 0$$

   Finalmente se asignan los cocientes de $M(n)$ a la ecuación, en orden.

   $$(1x^6 + 0x^5 - 3x^4 + 0x^3 + 1x^2 + 0x^1 + 2x^0) = 0$$

   $$x^6 - 3x^4 + x^2 + 2 = 0$$

2. **Plantear la ecuación característica usando las raíces del polinomio característico.**

   La ecuación tendrá tantos términos como el mayor nivel de profundidad y, por cada raíz r y según su multiplicidad m, se obtendrán una serie de valores de la forma $r^n \cdot n^{m-1}$ que serán los coeficientes de dichos términos.

   Ejemplo

   Suponiendo que las raíces del polinomio característico anterior fueran:

   $$(x-2)^3(x-5)^2(x-3) = 0$$

   Plantear la ecuación característica.

📝 **Raíces de una ecuación VS Soluciones de una ecuación**
Una ecuación puede expresarse como $(x-a)(x+b)^c = 0$, siendo $-a$ y b raíces.
Esto indica que las soluciones son a, y $-b$ repetida c veces.

El polinomio está compuesto por las raíces de la ecuación característica anterior (e), obteniéndose los siguientes datos:

Raíces	Multiplicidad	Coeficientes
2	3	$2^n, 2^n \cdot n^1, 2^n \cdot n^2$
5	2	$5^n, 5^n \cdot n^1$
3	1	3^n

📝 Si una solución fuera $-s$ (negativa), sus valores serían: $(-s)^n$, $(-s)^n n$, $(-s)^n n^2$...

La ecuación característica tendrá la forma:

$$\mathbf{M(n) = a \cdot 2^n + b \cdot 2^n \cdot n + c \cdot 2^n \cdot n^2 + d \cdot 5^n + e \cdot 5^n \cdot n + f \cdot 3^n}, \quad \mathbf{a, \ldots, f} \in \mathbb{N}$$

❗ Las ecuaciones de recurrencia expresan coste, **no pueden ser negativas**.

Esto significa que el término dominante de la ecuación nunca debe ser negativo.

- Tampoco podrá ser negativo el segundo dominante, si el primero se anulase.

---

3. **Componer la solución general mediante la ecuación característica y los casos bases.**

   Se busca crear un sistema de ecuaciones con el fin de despejar los coeficientes de la ecuación característica.

   Dicho sistema vendrá dado por las ecuaciones características igualadas a los casos base.

Ejemplo

Hallar la ecuación de la solución general suponiendo los siguientes datos:

$$M(n) = 2M(n-1) + 1, \qquad M(0) = 0, \quad M(1) = 1$$

Siendo la ecuación característica:

$$M(n) = a \cdot 2^n + b \cdot 2^n n$$

Encontrar los valores a, b, c.

$$[1] \quad \begin{cases} M(0) & = & a \cdot 2^{(0)} + b \cdot 2^{(0)}(0) & = & \mathbf{a} \\ M(0) & = & \mathbf{0} \end{cases}$$

$$[2] \quad \begin{cases} M(1) & = & a \cdot 2^{(1)} + b \cdot 2^{(1)}(1) & = & \mathbf{2a + 2b} \\ M(1) & = & \mathbf{1} \end{cases}$$

$$\begin{cases} a & & & = & 0 & [1] \\ 2a & + & 2b & = & 1 & [2] \end{cases}$$

4. **Resolver el sistema y presentar la ecuación de recurrencia.**

Ejemplo

Continuación del ejemplo anterior.

$$\begin{cases} a & & = & 0 & [1] \\ 2a & + \ 2b & = & 1 & [2] \end{cases} \longrightarrow \quad \mathbf{(a, b)} = \left(\mathbf{0}, \frac{\mathbf{1}}{\mathbf{2}}\right)$$

Finalmente, al sustituir en $M(n)$, se obtiene:

$$M(n) = 0 \cdot 2^n + \frac{1}{2} \cdot 2^n \cdot n = \underbrace{\frac{1}{2} \cdot 2^n \cdot n}_{\text{Dominante}}$$

$$\mathbf{M(n) \in \Theta(2^n \cdot n)}$$

Teorema Maestro

📖 **Teorema Maestro (general)**

Sirve para extraer la complejidad de una ecuación de recurrencia de forma directa.

Sea una ecuación de la forma $T(n) = aT\left(\frac{n}{b}\right) + f(n)$.

- $a \geq 1, \ b \geq 2$.

$$T(n) = \begin{cases} \Theta\left(n^{\log_b(a)}\right) & \text{si} & \exists \epsilon > 0 : f(n) \in O(n^{\log_b(a) - \epsilon}) \\ \Theta\left(n^{\log_b(a)} \cdot \log^{k+1}(n)\right) & \text{si} & f(n) \in \Theta\left(n^{\log_b(a)} \cdot \log^k(n)\right) \\ \Theta\left(f(n)\right) & \text{si} & \exists \epsilon > 0 : f(n) \in \Omega(n^{\log_b(a) + \epsilon}) \\ & & \exists e < 1, n_0 \geq 0 : \forall n \geq n_0, \ af\left(\frac{n}{b}\right) \leq ef(n) \end{cases}$$

❗ El tercer caso verifica la ***condición de regularidad***.

📝 **Información que ofrece la ecuación de recurrencia**

- $T(n)$: coste total del problema.
- $aT\left(\frac{n}{b}\right)$: coste de repetir a veces un subproblema de tamaño b.
  - Es decir, $T(n)$ se divide en b partes y se trabaja con a de ellas.
- $f(n)$: coste de la fusión de las partes. Esto es el caso general.

Ejemplo (clase)

Análisis de $T(n) = a^k T\left(\frac{n}{b^k}\right), \quad T(1) = 1, \quad k \in \mathbb{N}$.

Se sabe que el caso base es $T(1) = 1$.

Por tanto, $\exists k \quad / \quad T\left(\frac{n}{b^k}\right) = T(1)$.

$$T\left(\frac{n}{b^k}\right) = T(1) \quad \rightarrow \quad \frac{n}{b^k} = 1 \quad \rightarrow \quad n = b^k \quad \rightarrow \quad \log_b(n) = k$$

Finalmente:

$$T(n) = a^k T\left(\frac{n}{b^k}\right) = a^k \Big(T\left(1\right)\Big) = a^k \cdot 1 = a^k = a^{(\log_b(n))} = n^{\log_b(a)}$$

$$\mathbf{T(n)} \in \mathbf{\Theta(n)}$$

📖 **Teorema Maestro reducido**

Sirve para extraer la complejidad de una ecuación de recurrencia de forma directa, pero con menos precisión que la definición general.

Sea una ecuación de la forma $T(n) = aT\left(\frac{n}{b}\right) + f(n)$.

- $f(n) \in \Theta\left(n^d\right)$.
- $a \geq 1,\ b \geq 2,\ d \geq 0$.

$$T(n) = \begin{cases} \Theta\Big(n^{\log_b(a)}\Big) & \text{si } \ a > b^d \\ \Theta\Big(n^d \cdot \log(n)\Big) & \text{si } \ a = b^d \\ \Theta\Big(n^d\Big) & \text{si } \ a < b^d \end{cases}$$

❗ Solo puede usarse con **polinomios**, no con $\log(n)$.

Ejemplo (clase)

Una persona piensa en un número del 1 al 1000 y otra debe averiguar cuál es, mediante preguntas.

¿Cuál es la complejidad del algoritmo que describe el proceso de encontrar ese número, usando *divide y vencerás*?

La metodología de *divide y vencerás* consiste en reducir un problema en problemas más fáciles, con el fin de llegar a un problema que se resuelva de forma inmediata (caso base) y se use esa solución para resolver el problema anterior y sucesivos.

En este caso, la metodología prodía aplicarse preguntando a la persona si su número es mayor o menor que la mitad, y después mayor o menor dentro de esa mitad, y así sucesivamente, hasta encontrarlo.

Por tanto, una posible ecuación de recurrencia para este algoritmo sería:

$$\underbrace{T(n)}_{problema} = \underbrace{T\left(\frac{n}{2}\right)}_{\text{dividirlo por la mitad}} + \underbrace{1}_{\text{preguntar}}$$

Aplicando el Teorema Maestro reducido:

$$T(n) = \begin{cases} \Theta\Big(n^{\log_b(a)}\Big) & \text{si } \ a > b^d \\ \Theta\Big(n^d \cdot \log(n)\Big) & \text{si } \ a = b^d \\ \Theta\Big(n^d\Big) & \text{si } \ a < b^d \end{cases}$$

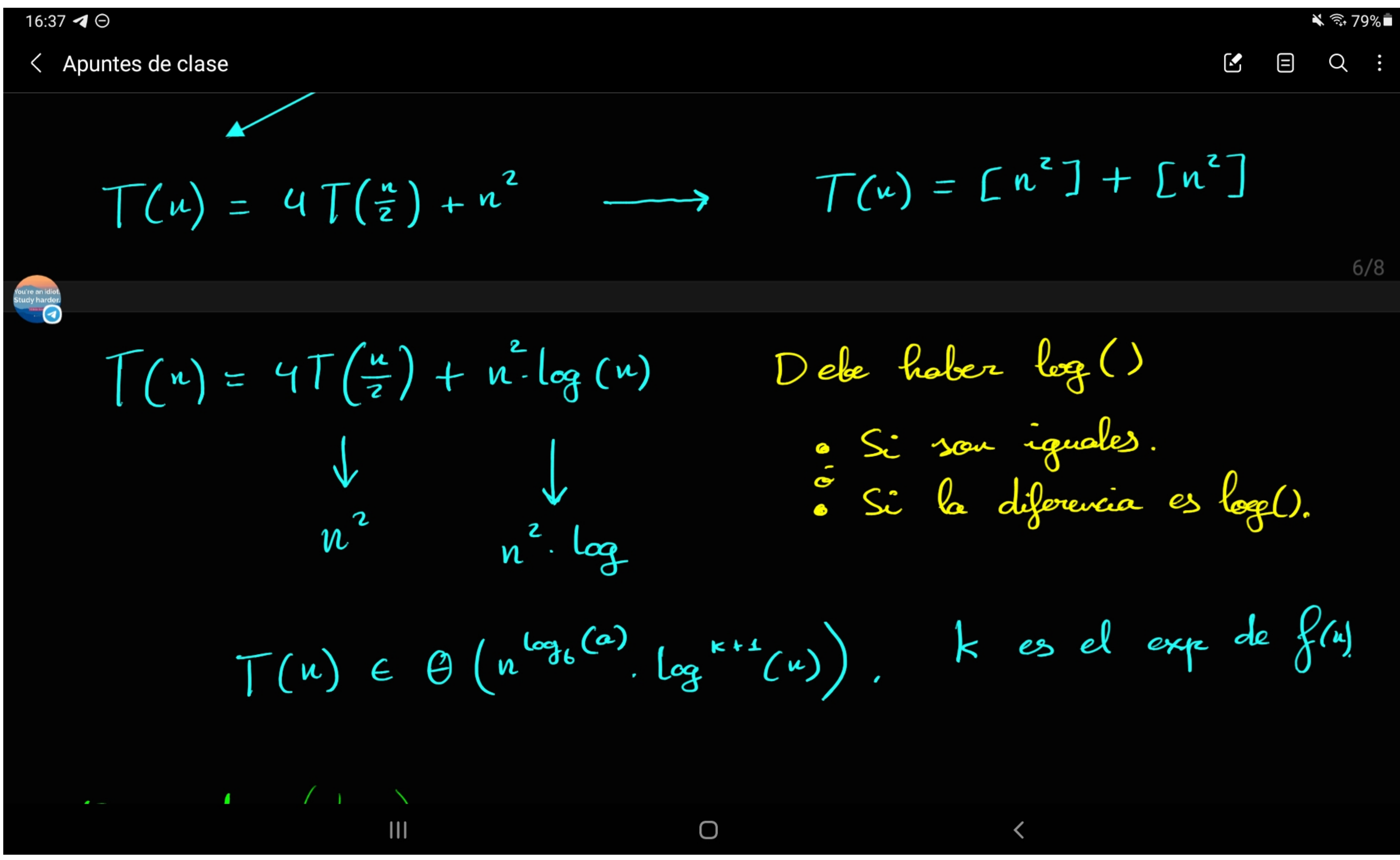
En este caso, $a = 1$, $b = 2$ y $d = 0$ (porque $f(n) = 1 = 1 \cdot n^0$), entonces:

$$T(n) \in \Theta\Big(n^d \cdot \log{(n)}\Big) \text{ porque } a = b^d \quad \longrightarrow \quad T(n) \in \Theta\Big(n^0 \cdot \log{(n)}\Big)$$

$$\mathbf{T(n)} \in \boldsymbol{\Theta}\Big(\log{(\mathbf{n})}\Big)$$

Ejemplo (clase)

Hallar la complejidad de $T(n) = 4T\left(\frac{n}{2}\right) + n^2 \cdot \log(n)$ usando la versión reducida del Teorema Maestro.



Cambio de variable

Búsqueda Binaria

Clases centrales de Complejidad

Recordando la primera definición del tema:

Complejidad temporal

- **Clase EXP**

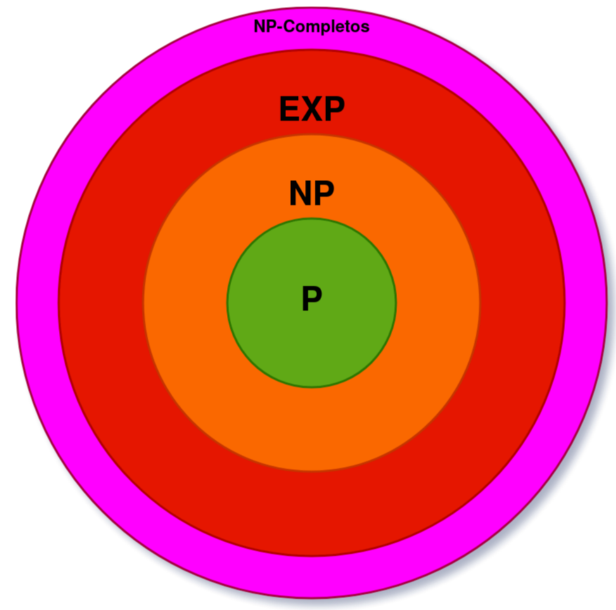
  Problemas resolubles en tiempo exponencial.

- **Clase NP**

  Problemas intratables, pero existe un algoritmo polinómico que comprueba soluciones.

- **Clase P**

  Problemas resolubles en tiempo polinómico.



❓ No se sabe si $|EXP| = |NP|$.

Complejidad espacial

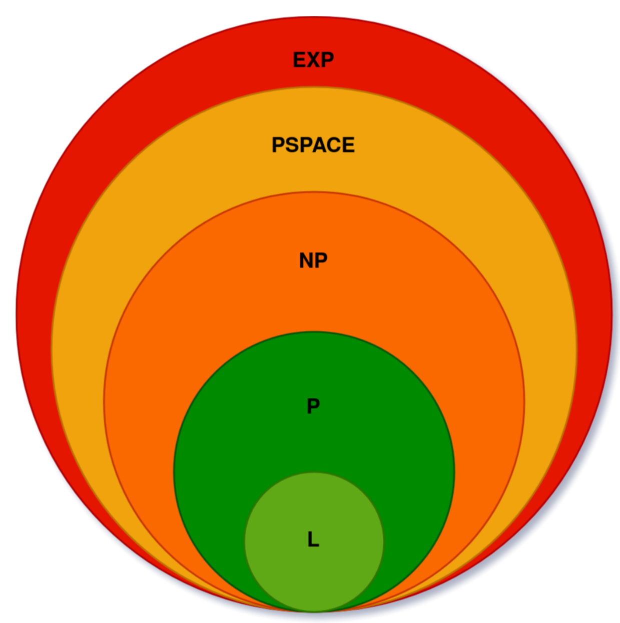
Se encarga de estudiar el coste de memoria principal que requieren los algoritmos para funcionar.

- **Clase PSPACE**

  Problemas resolubles utilizando espacio polinómico.

- **Clase L**

  Problemas resolubles usando espacio logarítmico respecto a la entrada.



💡 $\mathrm{L} \subseteq \mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{PSPACE} \subseteq \mathrm{EXP}$

Complejidad temporal vs espacial

🚨 **El espacio es reutilizable**, pero **el tiempo no**.