

Programación I

# Ordenamiento

# BubbleSort

BubbleSort es un algoritmo de ordenamiento simple que compara pares adyacentes de elementos en el arreglo y los intercambia si están en el orden incorrecto. Este proceso se repite hasta que el arreglo esté completamente ordenado y no se realicen intercambios en un paso completo.

Eficiencia: BubbleSort es menos eficiente en comparación con QuickSort y MergeSort, ya que tiene un tiempo de ejecución promedio de  $O(n^2)$  en el peor y mejor caso.

Es adecuado para arreglos pequeños o casi ordenados, pero no es práctico para arreglos grandes debido a su ineficiencia.

# BubbleSort

1. Iteras sobre el arreglo y comparas elementos adyacentes.
2. Si un elemento es mayor que el siguiente, los intercambias.
3. Repites este proceso para cada par de elementos en el arreglo, moviendo los elementos más grandes hacia el final.
4. Repites todo el proceso varias veces hasta que ningún intercambio sea necesario, lo que significa que el arreglo está ordenado.

# BubbleSort

Iteración 1:

Comparación: (64, 34) -> Intercambio -> [34, 64, 25, 12, 22, 11, 90]

Comparación: (64, 25) -> Intercambio -> [34, 25, 64, 12, 22, 11, 90]

Comparación: (64, 12) -> Intercambio -> [34, 25, 12, 64, 22, 11, 90]

Comparación: (64, 22) -> Intercambio -> [34, 25, 12, 22, 64, 11, 90]

Comparación: (64, 11) -> Intercambio -> [34, 25, 12, 22, 11, 64, 90]

Comparación: (64, 90) -> Sin intercambio -> [34, 25, 12, 22, 11, 64, 90]

# BubbleSort

Iteración 2:

Comparación: (34, 25) -> Intercambio -> [25, 34, 12, 22, 11, 64, 90]

Comparación: (34, 12) -> Intercambio -> [25, 12, 34, 22, 11, 64, 90]

Comparación: (34, 22) -> Intercambio -> [25, 12, 22, 34, 11, 64, 90]

Comparación: (34, 11) -> Intercambio -> [25, 12, 22, 11, 34, 64, 90]

Comparación: (34, 64) -> Sin intercambio -> [25, 12, 22, 11, 34, 64, 90]

# BubbleSort

Iteración 3:

Comparación: (25, 12) -> Intercambio -> [12, 25, 22, 11, 34, 64, 90]

Comparación: (25, 22) -> Intercambio -> [12, 22, 25, 11, 34, 64, 90]

Comparación: (25, 11) -> Intercambio -> [12, 22, 11, 25, 34, 64, 90]

Comparación: (25, 34) -> Sin intercambio -> [12, 22, 11, 25, 34, 64, 90]



# BubbleSort

Iteración 4:

Comparación: (12, 22) -> Sin intercambio -> [12, 22, 11, 25, 34, 64, 90]

Comparación: (22, 11) -> Intercambio -> [12, 11, 22, 25, 34, 64, 90]

# BubbleSort

Iteración 5:

Comparación: (12, 11) -> Intercambio -> [11, 12, 22, 25, 34, 64, 90]



# QuickSort

QuickSort es un algoritmo de ordenamiento eficiente que utiliza la estrategia de dividir y conquistar. El algoritmo elige un elemento pivote y particiona el arreglo en dos sub-arreglos alrededor del pivote, de manera que los elementos más pequeños que el pivote estén a su izquierda y los elementos más grandes estén a su derecha. Luego, aplica recursivamente el mismo proceso a los sub-arreglos.

Eficiencia: QuickSort es generalmente rápido y eficiente, con un tiempo de ejecución promedio de  $O(n \log n)$ . Sin embargo, en el peor caso puede degradarse a  $O(n^2)$  si el pivote seleccionado no divide el arreglo de manera equilibrada.

# QuickSort

1. Elige un elemento en el arreglo como pivote.
2. Divide el arreglo en dos sub-arreglos: uno con elementos más pequeños que el pivote y otro con elementos más grandes.
3. Aplica el mismo proceso recursivamente a cada sub-arreglo.
4. Combina los sub-arreglos ordenados para obtener el arreglo completo ordenado.

# QuickSort

Iteración 1 (pivote: 22):

Arreglo original: [64, 34, 25, 12, 22, 11, 90]

Partición: [12, 11, 22, 25, 64, 34, 90]

Sub-arreglos resultantes: [12, 11] [22] [25, 64, 34, 90]

# QuickSort

Iteración 2 (pivote: 11, 64):

Arreglo original: [12, 11, 22, 25, 64, 34, 90]

Partición: [11, 12] [22] [25, 64, 34, 90]

Sub-arreglos resultantes: [11] [12] [22] [25, 64, 34, 90]

# QuickSort

Iteración 3 (pivote: 25, 64):

Arreglo original: [11, 12, 22, 25, 64, 34, 90]

Partición: [11, 12, 22, 25] [64] [34, 90]

Sub-arreglos resultantes: [11, 12, 22, 25] [34] [64] [90]

# QuickSort

Iteración 4 (pivote: 12, 34, 90):

Arreglo original: [11, 12, 22, 25, 34, 64, 90]

Partición: [11, 12] [22, 25, 34] [64, 90]

Sub-arreglos resultantes: [11] [12] [22, 25, 34] [64, 90]

# QuickSort

Iteración 5 (pivote: 11, 12, 22, 25, 34, 64, 90):

Arreglo original: [11, 12, 22, 25, 34, 64, 90]

Partición: [11, 12, 22, 25, 34] [64, 90]

Sub-arreglos resultantes: [11, 12, 22, 25, 34] [64, 90]



# QuickSort

Iteración 6 (pivote: 34, 90):

Arreglo original: [11, 12, 22, 25, 34, 64, 90]

Partición: [11, 12, 22, 25, 34] [64, 90]

Sub-arreglos resultantes: [11, 12, 22, 25, 34] [64, 90]

# QuickSort

Iteración 7 (pivote: 11, 12, 22, 25, 34, 64, 90):

Arreglo original: [11, 12, 22, 25, 34, 64, 90]

Partición: [11, 12, 22, 25, 34, 64, 90]

Sub-arreglos resultantes: [11, 12, 22, 25, 34, 64, 90]

# MergeSort

MergeSort es un algoritmo de ordenamiento estable que también utiliza la estrategia de dividir y conquistar. Divide el arreglo en dos mitades, ordena recursivamente cada mitad y luego combina las mitades ordenadas en un solo arreglo ordenado.

Eficiencia: MergeSort es eficiente y siempre tiene un tiempo de ejecución garantizado de  $O(n \log n)$ , independientemente de la distribución de los elementos en el arreglo. Sin embargo, requiere espacio adicional para almacenar el arreglo auxiliar durante el proceso de fusión.

# MergeSort

1. Divide el arreglo en mitades hasta que cada mitad tenga un solo elemento (considerado ya ordenado).
2. Combina las mitades ordenadas en una sola lista ordenada fusionando los elementos en orden.

# MergeSort

1. Iteración 1 (División):
2. Arreglo original: [64, 34, 25, 12, 22, 11, 90]
3. Se divide en dos mitades: [64, 34, 25, 12] y [22, 11, 90]

# MergeSort

1. Iteración 2 (División):
2. Arreglo dividido izquierdo: [64, 34] y [25, 12]
3. Arreglo dividido derecho: [22, 11] y [90]

# MergeSort

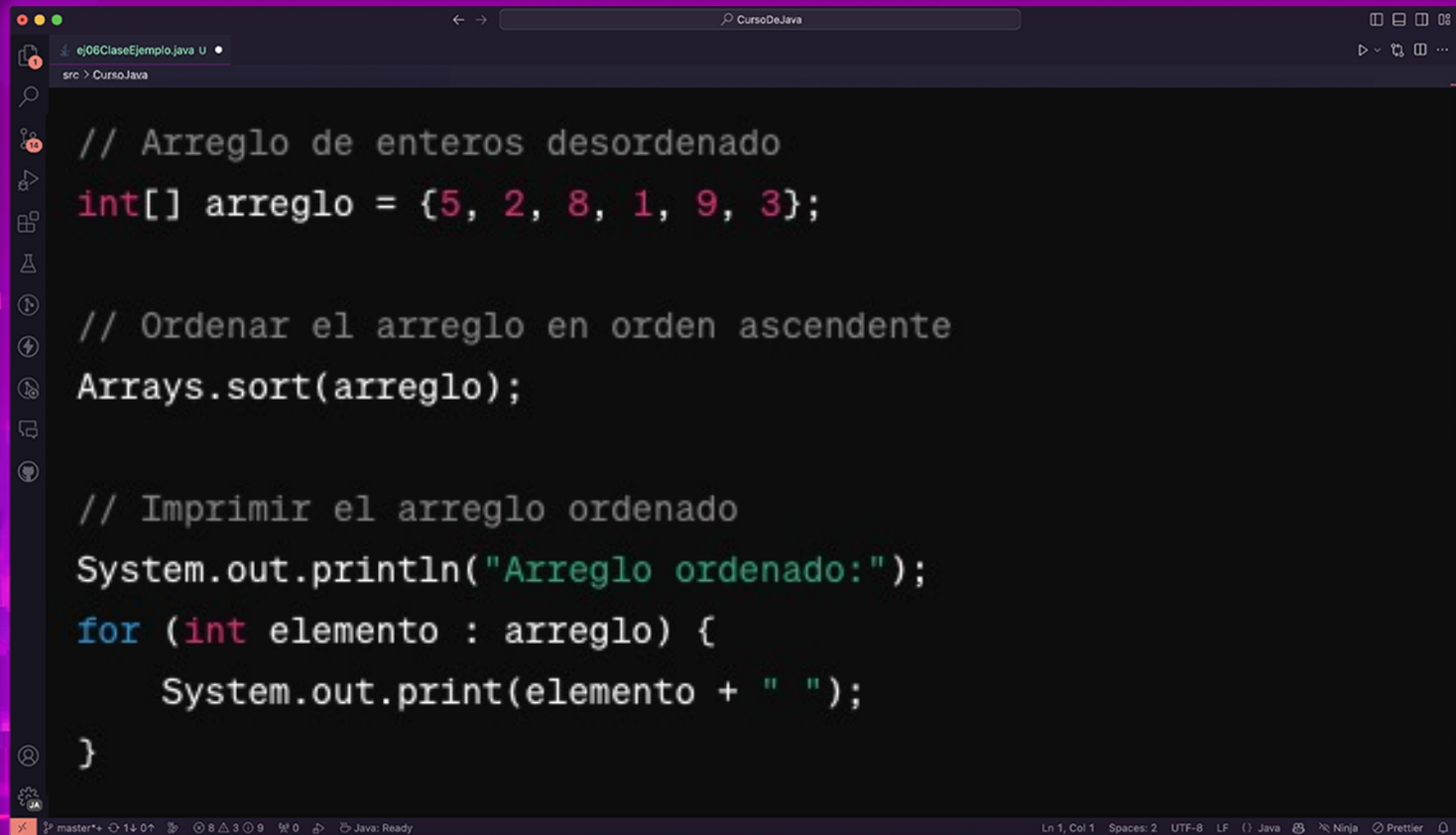
1. Ahora comenzaremos a fusionar los arreglos divididos:
2. Iteración 4 (Fusión):
3. Fusionamos [64] y [34] -> [34, 64]
4. Fusionamos [25] y [12] -> [12, 25]
5. Fusionamos [22] y [11] -> [11, 22]
6. Fusionamos [34, 64] y [12, 25] -> [12, 25, 34, 64]
7. Fusionamos [11, 22] y [90] -> [11, 22, 90]



# MergeSort

1. Iteración 5 (Fusión Final):
2. Fusionamos [12, 25, 34, 64] y [11, 22, 90] -> [11, 12, 22, 25, 34, 64, 90]

# Ordenamiento secuencial

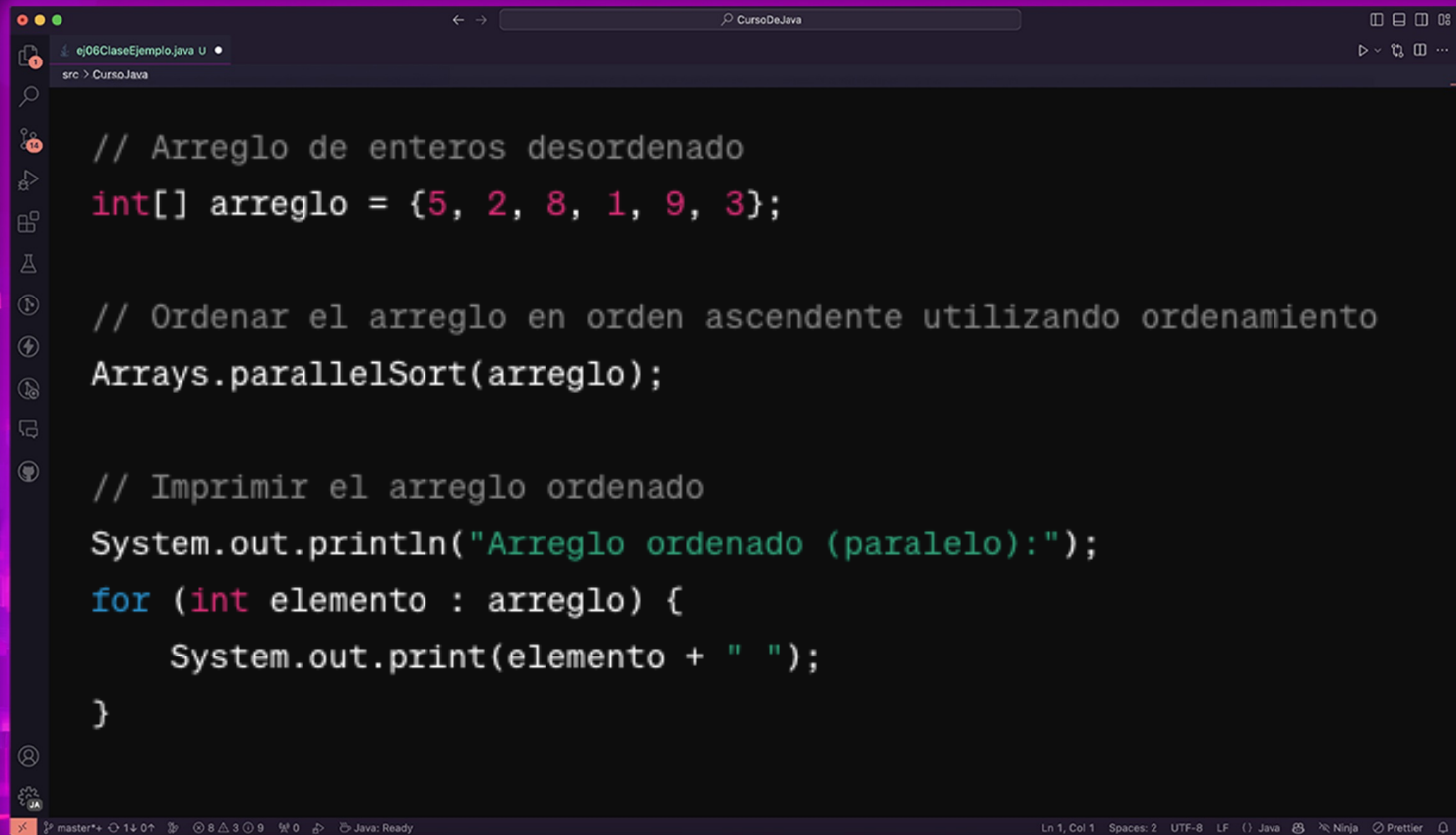
A screenshot of a code editor window with a dark theme. The window title is 'ej06ClaseEjemplo.java U'. The file path is 'src > CursoJava'. The code is in Java and demonstrates sorting an array of integers. The code includes comments in Spanish: '// Arreglo de enteros desordenado', '// Ordenar el arreglo en orden ascendente', and '// Imprimir el arreglo ordenado'. The array is initialized with {5, 2, 8, 1, 9, 3}. It uses 'Arrays.sort(arreglo);' to sort the array and a 'for' loop to print each element. The status bar at the bottom shows 'Ln 1, Col 1', 'Spaces: 2', 'UTF-8', 'LF', 'Java', and 'Prettier'.

```
// Arreglo de enteros desordenado
int[] arreglo = {5, 2, 8, 1, 9, 3};

// Ordenar el arreglo en orden ascendente
Arrays.sort(arreglo);

// Imprimir el arreglo ordenado
System.out.println("Arreglo ordenado:");
for (int elemento : arreglo) {
    System.out.print(elemento + " ");
}
```

# Ordenamiento paralelo

A screenshot of a code editor window with a dark theme. The title bar shows 'ej06ClaseEjemplo.java' and 'CursoDeJava'. The code is in Java and demonstrates parallel sorting using `Arrays.parallelSort`. The code includes comments in Spanish, an array initialization, a sorting call, and a loop to print the sorted elements. The status bar at the bottom shows 'master++', '140%', '8 3 9', '0', 'Java: Ready', 'Ln 1, Col 1', 'Spaces: 2', 'UTF-8', 'LF', 'Java', 'Ninja', and 'Prettier'.

```
// Arreglo de enteros desordenado
int[] arreglo = {5, 2, 8, 1, 9, 3};

// Ordenar el arreglo en orden ascendente utilizando ordenamiento
Arrays.parallelSort(arreglo);

// Imprimir el arreglo ordenado
System.out.println("Arreglo ordenado (paralelo):");
for (int elemento : arreglo) {
    System.out.print(elemento + " ");
}
```