

**UNIVERSIDAD MARIANO GALVEZ DE  
GUATEMALA CAMPUS  
JUTIAPA, JUTIAPA  
FACULTAD DE INGENIERIA EN SISTEMAS DE  
INFORMACION**



**CURSO:  
PROGRAMACION III**

**DOCENTE:  
ING. WALTER ANIBAL CORDOVA ZECEÑA**

**ALUMNO:  
JEFERSÓN OSLEE CERMEÑO PINEDA**

**TAREA SOBRE:  
ARREGLOS Y ALGORITMOS EN JAVA**

**FECHA: 13/02/2026**

# Parte 1: Ingeniería Inversa del Jar

## Evidencia:

The image shows two windows from the Java Decomplier. The left window displays the file structure of the jar file 'introduction-0.0.1-SNAPSHOT.jar'. The right window shows the decompiled Java code for the 'BubbleSort.class' file.

**File Structure (Left Window):**

- META-INF
- maven.umg.edu.gt.data-structure.introduction
- MANIFEST.MF
- umg.edu.gt.data\_structure
- array
  - BubbleSort
  - MergeSortDemo
  - QuickSort
  - SumArray
- introduction
  - App

**Decompiled Code (Right Window):**

```
package umg.edu.gt.data_structure.array;

public class BubbleSort {
    public void bubbleSort(int[] arr) {
        int n = arr.length;

        for (int i = 0; i < n - 1; ++i) {
            boolean swapped = false;

            for (int j = 0; j < n - 1 - i; ++j) {
                if (arr[j] >= arr[j + 1]) {
                    continue;
                }
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
                swapped = true;
            }

            if (!swapped)
                return;
        }
    }
}
```

## Código analizado y decompilado:

- Qué clases existen.

Según lo que se ve en el árbol:

- ✚ Paquete:  
umg.edu.gt.data\_structure
- ✚ Subpaquete: array
  - Contiene:
    - BubbleSort
    - MergeSortDemo
    - QuickSort
    - SumArray
- ✚ Subpaquete: introduction
  - Contiene:
    - App

- Que operaciones de realizan sobre los arreglos.

➤ Clase Analizada: **BubbleSort**

En esta clase se realizan las siguientes operaciones:

### a) Obtención del tamaño del arreglo

int n = arr.length;

Se usa la propiedad .length para conocer el tamaño.

## b) Recorrido del arreglo

```
for (int i = 0; i < n - 1; ++i)
for (int j = 0; j < n - 1 - i; ++j)
```

Se utilizan ciclos for anidados para recorrer el arreglo múltiples veces.

## c) Comparación entre elementos consecutivos

```
if (arr[j] <= arr[(j + 1)])
```

Se comparan elementos adyacentes del arreglo.

## d) Intercambio de elementos

```
int temp = arr[j];
arr[j] = arr[(j + 1)];
arr[(j + 1)] = temp;
```

Se intercambian posiciones dentro del arreglo.

## e) Optimización con variable booleana

```
boolean swapped
```

Si no hay intercambios, el algoritmo termina antes.

### ➤ Clase Analizada: **MergeSortDemo**

Aquí se realizan varias operaciones importantes:

## a) Verificación del tamaño del arreglo

```
if (a.length <= 1)
```

Se usa .length para determinar si el arreglo ya está ordenado (caso base).

## b) División del arreglo

```
int mid = a.length / 2;
```

Luego:

```
int[] left = Arrays.copyOfRange(a, 0, mid);
int[] right = Arrays.copyOfRange(a, mid, a.length);
```

Se crean **subarreglos** usando:

**Arrays.copyOfRange**

Esto pertenece a:

java.util.Arrays

Operación realizada:

- Copia parcial de arreglos
- División del arreglo original en dos partes

### c) Llamadas recursivas

```
mergeSort(left);
mergeSort(right);
```

Se aplica el algoritmo recursivamente a los subarreglos.

### d) Fusión de arreglos (merge)

En el método merge:

```
while ((i < left.length) && (j < right.length))
```

Se comparan elementos de ambos arreglos y se insertan en el arreglo original:

```
a[(k++)] = left[(i++)];
a[(k++)] = right[(j++)];
```

Operaciones realizadas:

- Comparación
- Copia de elementos
- Reconstrucción del arreglo ordenado

➤ Clase Analizada: **QuickSort**

### a) Recursividad sobre el arreglo

```
quickSort(arr, low, pivotIndex - 1);
quickSort(arr, pivotIndex + 1, high);
```

Se divide el arreglo en subarreglos usando índices.

No crea nuevos arreglos, trabaja sobre el mismo.

### b) Selección de pivote

```
int pivot = arr[high];
```

Se toma el último elemento como pivote.

### c) Recorrido parcial del arreglo

```
for (int j = low; j < high; ++j)
```

Se recorre desde low hasta high - 1.

### d) Comparación con el pivote

```
if (arr[j] <= pivot)
```

Se comparan los elementos con el pivote.

### e) Intercambio de elementos

```
int temp = arr[i];
arr[i] = arr[j];
arr[j] = temp;
```

Y también al final:

```
arr[i + 1] = arr[high];
arr[high] = temp;
```

Se intercambian elementos dentro del arreglo.

➤ Clase Analizada: **SumArray**

### a) Recorrido del arreglo

```
for (int n : nums)
```

Se utiliza un **for-each** (recorrido directo del arreglo).

Esto es equivalente a:

```
for (int i = 0; i < nums.length; i++)
```

### b) Acceso a cada elemento

En cada iteración:

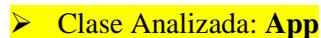
```
total += n;
```

Se accede al valor almacenado en el arreglo y se acumula.

## c) Operación matemática sobre los elementos

Se realiza:

- Suma acumulativa
- Uso de variable auxiliar total



En esta clase se hacen varias operaciones importantes:

### a) Declaración e inicialización de arreglos

Ejemplos:

```
int[] d1 = { 3, 5, 2, 9 };  
int[] d2 = { 8, 3, 7, 4, 9, 2 };
```

Se crean arreglos con valores predefinidos.

### b) Uso de método de suma

```
sumArray.sum(d1);
```

Se pasa un arreglo como parámetro a otro método.

### c) Aplicación de algoritmos de ordenamiento

Se utilizan tres algoritmos diferentes:

- mergeSort(d2)
- bubbleSort(arr)
- quickSort(arr1, 0, arr1.length - 1)

### d) Recorrido del arreglo con for-each

```
for (n : arr)
```

Se imprime cada elemento del arreglo.

### e) Uso de método de la librería estándar

```
Arrays.sort(arr2);
```

Aquí se utiliza:

`java.util.Arrays`

Métodos usados:

- `Arrays.sort()`
- `Arrays.toString()`

## • Qué algoritmos de ordenamiento se utilizan.

➤ Clase Analizada: **BubbleSort**

Se utiliza el algoritmo:

## Bubble Sort (Ordenamiento Burbuja)

Características que lo demuestran:

- Dos ciclos for anidados
- Comparación de elementos consecutivos
- Intercambio si el anterior es mayor
- Optimización con bandera swapped

### Complejidad del algoritmo

#### Tiempo

- Peor caso:  
 $O(n^2)$
- Mejor caso (si ya está ordenado y no hay swaps):  
 $O(n)$

#### Espacio

- $O(1)$  (solo usa una variable temporal)

La clase BubbleSort implementa el algoritmo de ordenamiento Burbuja (Bubble Sort). Este algoritmo recorre el arreglo utilizando dos ciclos anidados, comparando elementos consecutivos y realizando intercambios cuando el elemento actual es mayor que el siguiente.

Se utiliza una variable booleana llamada swapped para optimizar el proceso, permitiendo terminar anticipadamente si el arreglo ya se encuentra ordenado.

La complejidad temporal en el peor caso es  $O(n^2)$  y el espacio utilizado es  $O(1)$ .

➤ Clase Analizada: **MergeSortDemo**

Se utiliza: Merge Sort

Características que lo confirman:

- División del arreglo en mitades
- Uso de recursividad
- Método merge para combinar
- Uso de arreglos auxiliares

## Complejidad del algoritmo:

### Tiempo

- Siempre:  
 $O(n \log n)$

Porque:

- Divide en  $\log n$  niveles
- En cada nivel recorre  $n$  elementos

### Espacio

- $O(n)$

Porque crea arreglos auxiliares (left y right).

La clase MergeSortDemo implementa el algoritmo Merge Sort.

El algoritmo divide el arreglo en dos mitades utilizando el método `Arrays.copyOfRange`, aplicando recursividad hasta que los subarreglos tienen un solo elemento.

Posteriormente, utiliza un método auxiliar `merge` para combinar los arreglos ordenados comparando sus elementos.

La complejidad temporal del algoritmo es  $O(n \log n)$  y su complejidad espacial es  $O(n)$  debido al uso de arreglos auxiliares.

➤ Clase Analizada: **QuickSort**

Se utiliza: QuickSort

Características que lo confirman:

- Uso de pivote
- Método partition
- Recursividad
- División en subarreglos mediante índices
- Ordenamiento in-place (sin arreglos auxiliares)

## Complejidad:

### Tiempo

- Mejor caso:  
 $O(n \log n)$
- Caso promedio:  
 $O(n \log n)$
- Peor caso (si el pivote es el menor o mayor siempre):  
 $O(n^2)$

### Espacio

- $O(\log n)$  por la pila de recursión
- No crea arreglos auxiliares

La clase QuickSort implementa el algoritmo de ordenamiento QuickSort.

El método selecciona el último elemento como pivote y utiliza el método partition para reorganizar el arreglo colocando los elementos menores al pivote a la izquierda y los mayores a la derecha. Posteriormente, aplica recursividad sobre las particiones generadas.

La complejidad promedio es  $O(n \log n)$ , mientras que en el peor caso puede ser  $O(n^2)$ . El algoritmo trabaja in-place, por lo que no requiere arreglos auxiliares adicionales.

#### ➤ Clase Analizada: **SumArray**

Ninguno.

Esta clase **no realiza ordenamiento**.

Solo realiza una operación básica sobre arreglos: **sumatoria de elementos**.

## Complejidad:

### Tiempo

Recorre el arreglo una sola vez:

**$O(n)$**

### Espacio

Solo usa una variable adicional:

**$O(1)$**

La clase SumArray realiza una operación básica sobre arreglos, específicamente la sumatoria de sus elementos.

Utiliza un ciclo for-each para recorrer el arreglo y acumular los valores en una variable auxiliar.

La complejidad temporal del método es  $O(n)$ , ya que recorre el arreglo una sola vez, y la complejidad espacial es  $O(1)$ .

➤ Clase Analizada: **App**

Del análisis completo, el JAR contiene:

## 1. Bubble Sort

Implementado en la clase BubbleSort.

## 2. Merge Sort

Implementado en MergeSortDemo.

## 3. Quick Sort

Implementado en QuickSort.

## 4. Ordenamiento usando librería estándar

Arrays.sort() (que internamente usa Dual-Pivot QuickSort para enteros).

El archivo JAR contiene implementaciones de distintos algoritmos de ordenamiento sobre arreglos, incluyendo Bubble Sort, Merge Sort y QuickSort. Además, se implementa una clase para realizar la suma de elementos de un arreglo y una clase principal que demuestra el uso de estos algoritmos. También se hace uso de la clase java.util.Arrays para ordenar e imprimir arreglos utilizando métodos estándar de Java.

## Parte 2: Ejercicio algorítmico

### Pseudocódigo:

Inicio

Si tamaño del arreglo < 2 entonces

    Mostrar "El arreglo debe tener al menos dos elementos"

    Terminar

FinSi

mayor ← -∞

segundoMayor ← -∞

```
menor ← +∞  
segundoMenor ← +∞
```

Para cada elemento x en el arreglo hacer

```
// Actualizar mayor y segundo mayor  
Si x > mayor entonces  
    segundoMayor ← mayor  
    mayor ← x  
Sino si x > segundoMayor Y x ≠ mayor entonces  
    segundoMayor ← x  
FinSi
```

```
// Actualizar menor y segundo menor  
Si x < menor entonces  
    segundoMenor ← menor  
    menor ← x  
Sino si x < segundoMenor Y x ≠ menor entonces  
    segundoMenor ← x  
FinSi
```

FinPara

```
Mostrar "Segundo mayor: ", segundoMayor  
Mostrar "Segundo menor: ", segundoMenor
```

Fin

## Explicación de por qué el algoritmo funciona

El algoritmo funciona porque mantiene cuatro variables auxiliares:

- mayor
- segundoMayor
- menor
- segundoMenor

Durante una única pasada por el arreglo:

- Si un número es mayor que el mayor actual, el valor anterior de mayor pasa a ser segundoMayor.

- Si no es el mayor pero es mayor que segundoMayor, se actualiza segundoMayor.
- Lo mismo ocurre con menor y segundoMenor.

De esta forma, en cada iteración se actualizan los extremos sin necesidad de ordenar el arreglo, garantizando que al finalizar el recorrido se tienen correctamente el segundo mayor y el segundo menor.

## Análisis de Complejidad

### Complejidad Temporal

El arreglo se recorre una sola vez.

Si el tamaño del arreglo es n:

$O(n)$

Porque solo existe un ciclo que recorre todos los elementos.

### Complejidad Espacial

Solo se utilizan cuatro variables adicionales:

$O(1)$

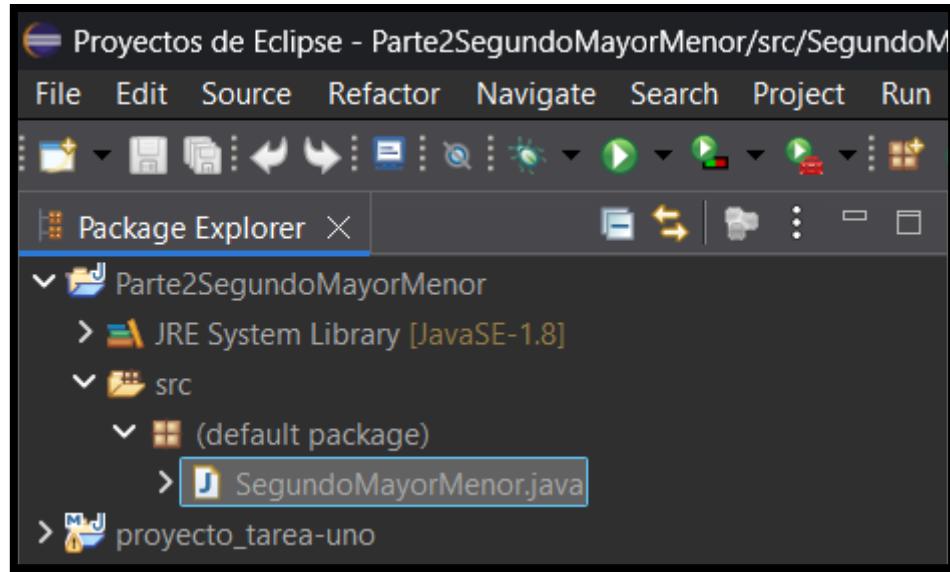
Espacio constante, ya que no depende del tamaño del arreglo.

## RESTRICCIÓN

- Se recorre una sola vez
- No se utiliza ningún algoritmo de ordenamiento
- No se usa Arrays.sort()
- No se crean arreglos auxiliares
- Solo se usan variables simples

El algoritmo evita el uso de estructuras adicionales y no altera el orden original del arreglo, cumpliendo estrictamente la restricción de recorrerlo una sola vez sin aplicar ordenamiento.

**Quise realizar también el ejercicio en Eclipse y probarlo.**



Primero cree un nuevo proyecto java con el nombre del ejercicio.

Segundo donde me aparece mi SRC, haga clic derecho y cree una clase y la llame SegundoMayorMenor.

Luego me metí a investigar un poco porque quería encontrar una página o algo, para que pasara mi pseudocódigo a código java para probar si obtenía un resultado correcto.

La página que utilicé es esta: “CodeConvert”

Inicio

Si tamaño del arreglo < 2 entonces  
Mostrar "El arreglo debe tener al menos dos elementos"  
Terminar  
FinSi

mayor ← -∞  
segundoMayor ← -∞  
menor ← +∞  
segundoMenor ← +∞

```
22- importar java.util.Arrays;
23-
24- clase pública SegundoMayorMenor {
25-     público estático void main ( String [ ] args ) {
26-         int [ ] arreglo = { 5 , 3 , 9 , 1 , 6 , 9 , 2 };
27-
28-         si ( arreglo . length < 2 ) {
29-             Sistema . afuera . println ( "El arreglo debe tener al menos dos elementos" );
30-             devolver ;
31-         }
32-
33-         int mayor = Entero . VALOR_MIN ;
34-         int segundoMayor = Entero . VALOR_MIN ;
35-         int menor = Integer . MAX_VALUE ;
36-         int segundoMenor = Integer . MAX_VALUE ;
37-
38-         para ( int x : arreglo ) {
39-             si ( x > mayor ) {
40-                 mayor = x ;
41-             }
42-             si ( x < menor ) {
43-                 menor = x ;
44-             }
45-             si ( x > segundoMayor ) {
46-                 segundoMayor = x ;
47-             }
48-             si ( x < segundoMenor ) {
49-                 segundoMenor = x ;
50-             }
51-         }
52-     }
53- }
```

The screenshot shows the Eclipse IDE interface. In the top menu bar, it says "Proyectos de Eclipse - Parte2SegundoMayorMenor/src/SegundoMayorMenor.java - Eclipse IDE". The left sidebar shows a project structure with a package named "Parte2SegundoMayorMenor" containing a source folder "src" with a file "SegundoMayorMenor.java". The main editor window displays the Java code for "SegundoMayorMenor". The code initializes an array with values {5, 3, 9, 1, 6, 9, 2}, checks if it has less than 2 elements, and then iterates through the array to find the second largest and second smallest numbers. The output in the Console view shows "Segundo mayor: 6" and "Segundo menor: 2".

```

3  public class SegundoMayorMenor {
4●   public static void main(String[] args) {
5     int[] arreglo = {5, 3, 9, 1, 6, 9, 2};
6
7●     if (arreglo.length < 2) {
8       System.out.println("El arreglo debe tener al menos dos elementos");
9       return;
10    }
11
12    int mayor = Integer.MIN_VALUE;
13    int segundoMayor = Integer.MIN_VALUE;
14    int menor = Integer.MAX_VALUE;
15    int segundoMenor = Integer.MAX_VALUE;
16
17●    for (int x : arreglo) {
18      if (x > mayor) {
19        segundoMayor = mayor;
20        mayor = x;
21      } else if (x > segundoMayor && x != mayor) {
22        segundoMayor = x;
23      }
24
25●      if (x < menor) {
26        segundoMenor = menor;
27        menor = x;
28      } else if (x < segundoMenor && x != menor) {
29        segundoMenor = x;
30      }
31    }
32
33    System.out.println("Segundo mayor: " + segundoMayor);
34    System.out.println("Segundo menor: " + segundoMenor);
35  }
36
37

```

Problems Javadoc Declaration Console Eclipse IDE for Java Developers 2026-03 M2  
<terminated> SegundoMayorMenor [Java Application] C:\Program Files\Java\jdk-21\bin\javaw.exe (13 feb 2026, 17:09:16 -)  
Segundo mayor: 6  
Segundo menor: 2

## Resultado esperado con el arreglo:

Yo realice un arreglo con datos alazar que serían estos:

10, 15, 4, 8, 5

Haciendo la operación me quedo haci:

Mayor = 15

Segundo mayor = 10

Menor = 4

Segundo menor = 5

**Y como respuesta tenía:** segundo mayor 10, y segundo menor 5.

- Ya comprobándolo en el programa java que hice me dio esto ya que lo hice con el código que me genero la página “CodeConvert”

int[] arreglo = {5, 3, 9, 1, 6, 9, 2};

Mayor = 9

Segundo mayor = 6

Menor = 1

Segundo menor = 2

Salida en consola:

The screenshot shows the Eclipse IDE interface with the focus on the Console view. The title bar says "Problems Javadoc Declaration Console Eclipse IDE". Below it, it says "<terminated> SegundoMayorMenor [Java Application] C:\Program Files\Java\jdk-21\bin\javaw.exe (13 feb 2026, 17:09:16 -)". The console output is displayed in a black background with white text, showing "Segundo mayor: 6" and "Segundo menor: 2".