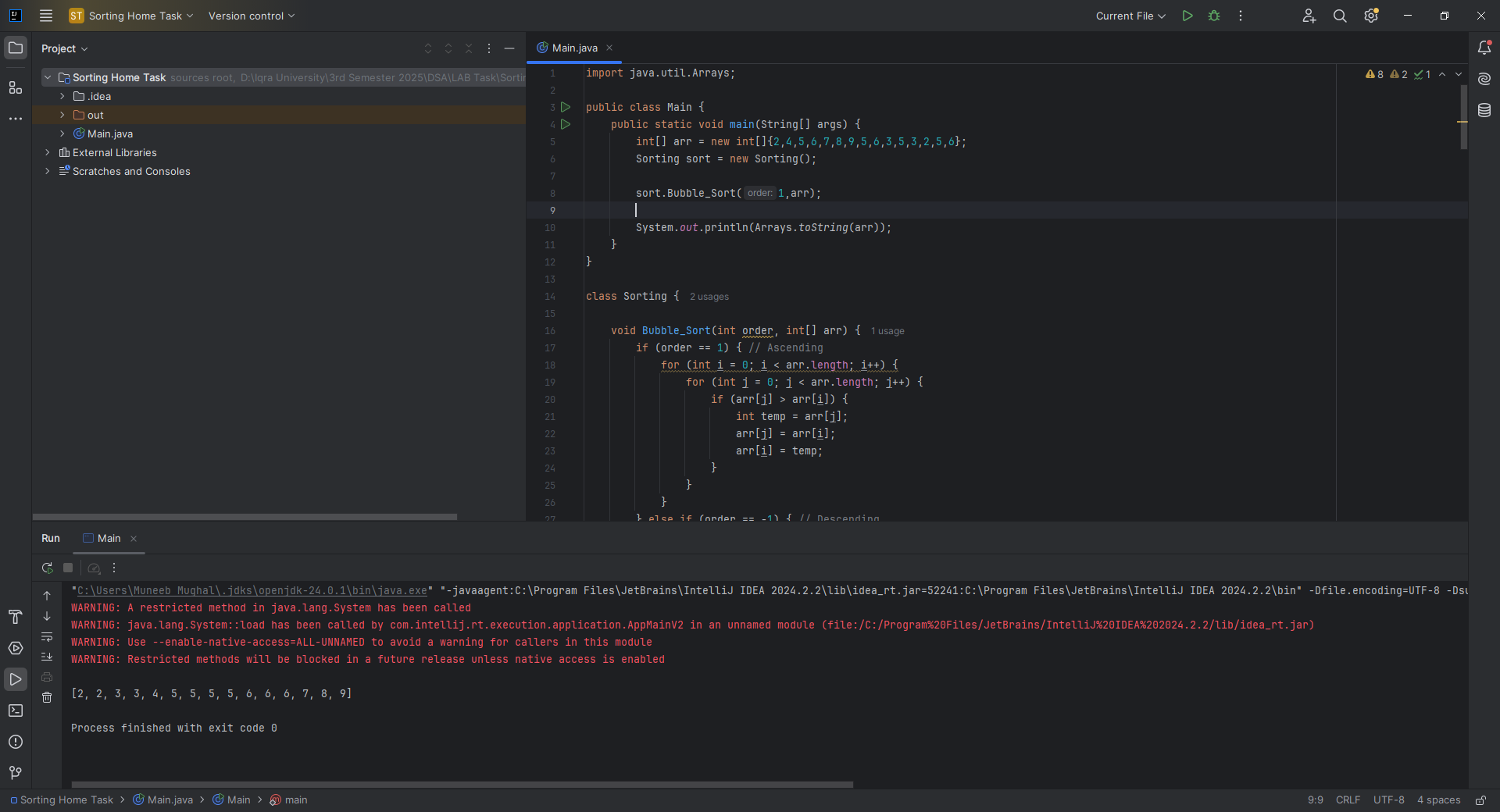
**CLASS TASK 1:**

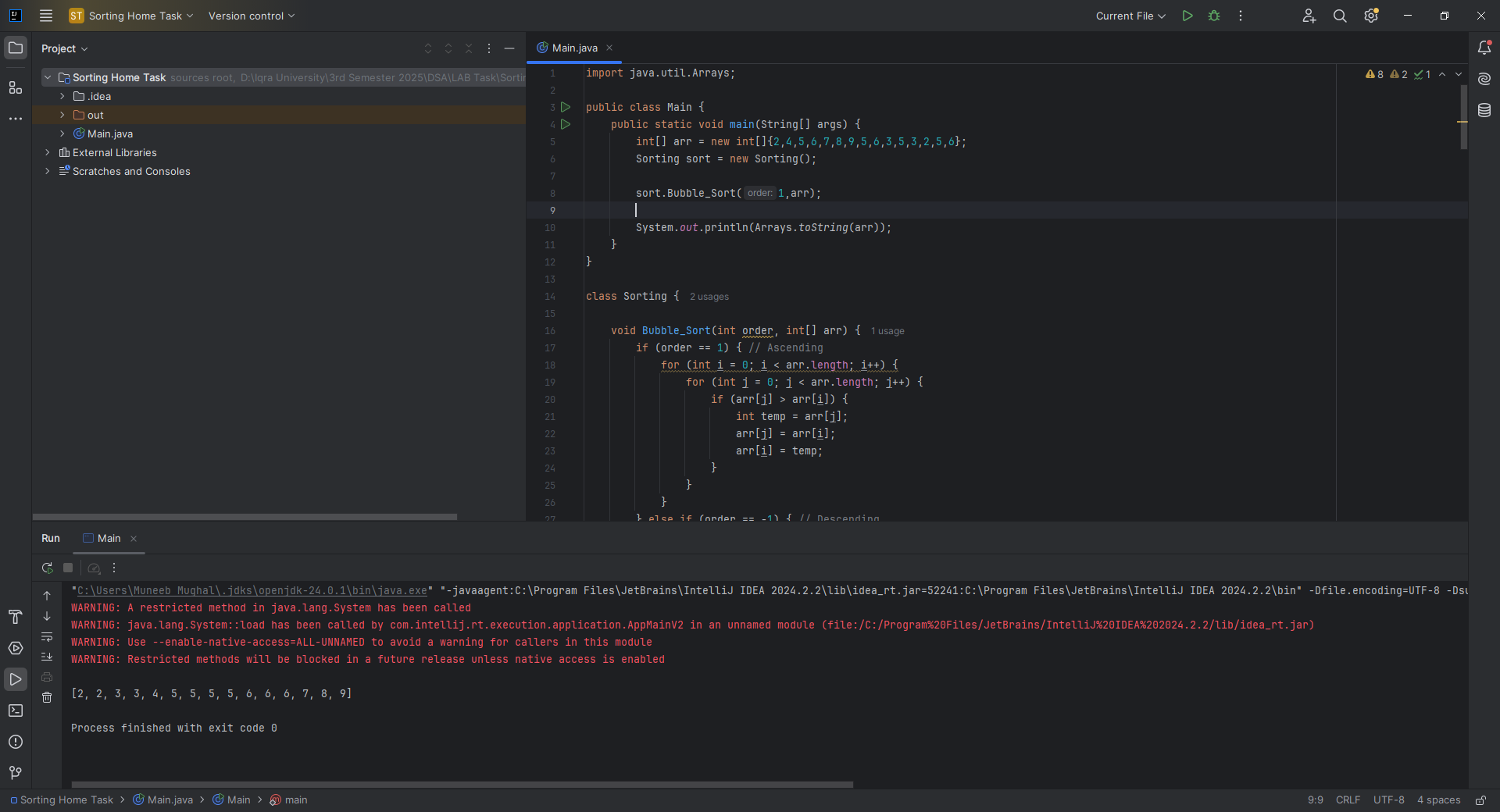
void Bubble\_Sort(int order, int[] arr) {  
 if (order == 1) { // Ascending  
 for (int i = 0; i < arr.length; i++) {  
 for (int j = 0; j < arr.length; j++) {  
 if (arr[j] > arr[i]) {  
 int temp = arr[j];  
 arr[j] = arr[i];  
 arr[i] = temp;  
 }  
 }  
 }  
 } else if (order == -1) { // Descending  
 for (int i = 0; i < arr.length; i++) {  
 for (int j = 0; j < arr.length; j++) {  
 if (arr[j] < arr[i]) {  
 int temp = arr[j];  
 arr[j] = arr[i];  
 arr[i] = temp;  
 }  
 }  
 }  
 } else {  
 throw new RuntimeException("Invalid order");  
 }  
}



**CLASS TASK 2**

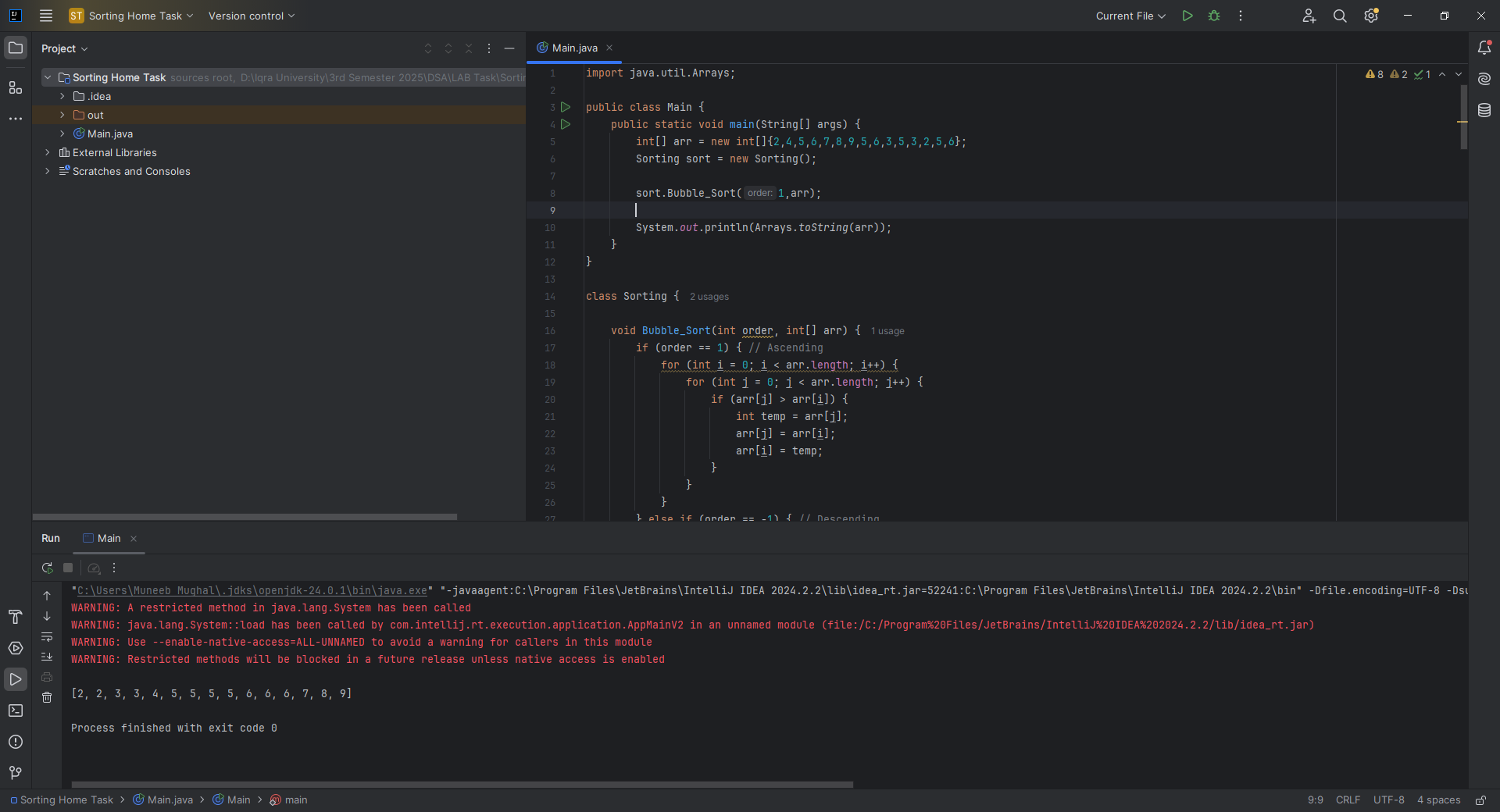
**CODE:**

void selection\_Sorting(int order, int[] arr) {  
 for (int i = 0; i < (arr.length - 1); i++) {  
 int min = i;  
 for (int j = i + 1; j < arr.length; j++) {  
 if (arr[j] < arr[min]) {  
 min = j;  
 }  
 }  
 int temp = arr[i];  
 arr[i] = arr[min];  
 arr[min] = temp;  
 }  
}



**CLASS TASK 3**

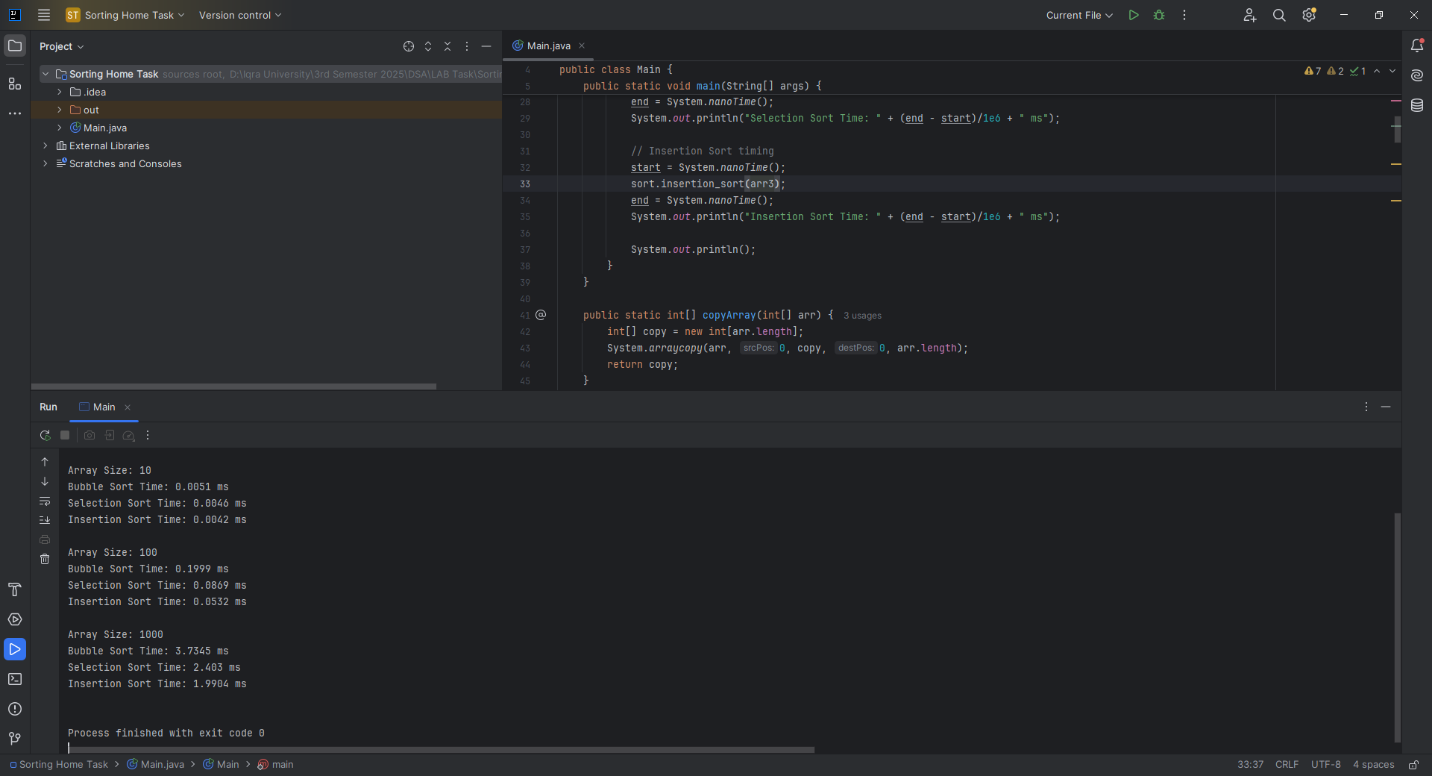
**CODE:**  
void insertion\_sort(int[] arr) {  
 for (int i = 1; i < arr.length; i++) {  
 int key = arr[i];  
 int j = i - 1;  
 while (j >= 0 && arr[j] > key) {  
 arr[j + 1] = arr[j];  
 j--;  
 }  
 arr[j + 1] = key;  
 }  
}

****

**CLASS TASK 4**

**CODE:**

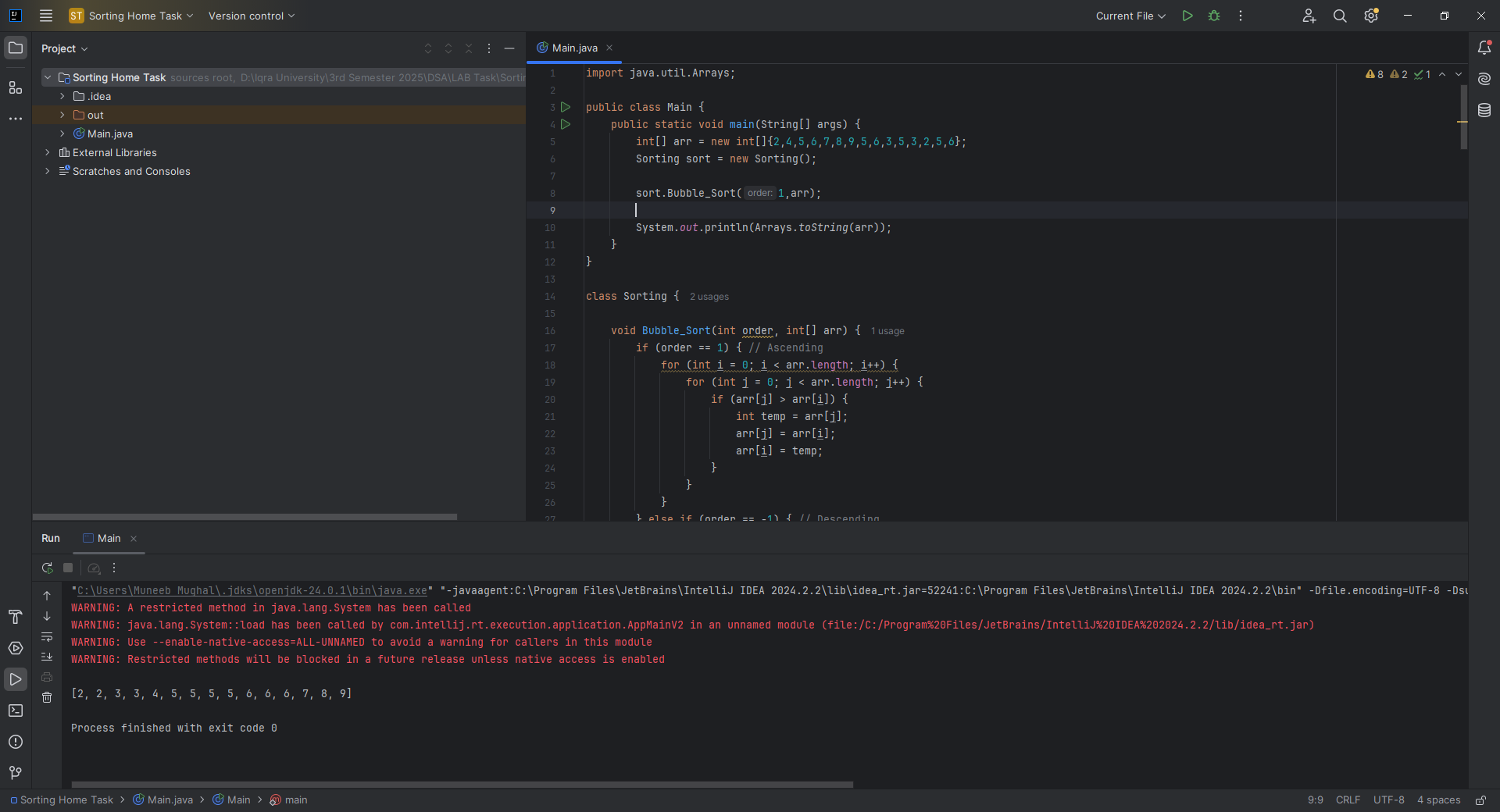
import java.util.Arrays;  
import java.util.Random;  
  
public class Main {  
 public static void main(String[] args) {  
 Sorting sort = new Sorting();  
 int[] sizes = {10, 100, 1000};  
  
 for(int size : sizes) {  
 System.*out*.println("Array Size: " + size);  
 int[] original = *generateRandomArray*(size, 10000);  
  
 int[] arr1 = *copyArray*(original);  
 int[] arr2 = *copyArray*(original);  
 int[] arr3 = *copyArray*(original);  
  
 long start, end;  
  
 // Bubble Sort timing  
 start = System.*nanoTime*();  
 sort.Bubble\_Sort(1,arr1);  
 end = System.*nanoTime*();  
 System.*out*.println("Bubble Sort Time: " + (end - start)/1e6 + " ms");  
  
 // Selection Sort timing  
 start = System.*nanoTime*();  
 sort.selection\_Sorting(1,arr2);  
 end = System.*nanoTime*();  
 System.*out*.println("Selection Sort Time: " + (end - start)/1e6 + " ms");  
  
 // Insertion Sort timing  
 start = System.*nanoTime*();  
 sort.insertion\_sort(arr3);  
 end = System.*nanoTime*();  
 System.*out*.println("Insertion Sort Time: " + (end - start)/1e6 + " ms");  
  
 System.*out*.println();  
 }  
 }  
  
 public static int[] copyArray(int[] arr) {  
 int[] copy = new int[arr.length];  
 System.*arraycopy*(arr, 0, copy, 0, arr.length);  
 return copy;  
 }  
  
 public static int[] generateRandomArray(int size, int bound) {  
 int[] array = new int[size];  
 Random rand = new Random();  
 for(int i = 0; i < size; i++) {  
 array[i] = rand.nextInt(bound);  
 }  
 return array;  
 }  
}

****

**HOME TASK 1**

**CODE:**

void mergeSort(int[] array) {  
 if (array.length < 2) return;  
  
 int mid = array.length / 2;  
 int[] leftArray = Arrays.*copyOfRange*(array, 0, mid);  
 int[] rightArray = Arrays.*copyOfRange*(array, mid, array.length);  
  
 mergeSort(leftArray);  
 mergeSort(rightArray);  
 merge(array, leftArray, rightArray);  
}  
  
private void merge(int[] array, int[] left, int[] right) {  
 int i = 0, j = 0, k = 0;  
  
 while (i < left.length && j < right.length) {  
 if (left[i] < right[j]) {  
 array[k++] = left[i++];  
 } else {  
 array[k++] = right[j++];  
 }  
 }  
  
 while (i < left.length) array[k++] = left[i++];  
 while (j < right.length) array[k++] = right[j++];  
}



**1. Divide**

The array is divided into two halves:

* If the array has more than one element, it is split into two subarrays of approximately equal size.
* This splitting continues **recursively** until each subarray has **only one element**, which is inherently sorted.

**2. Conquer (Recursively Sort)**

Each pair of one-element arrays is merged into a sorted array.

Merging happens in the reverse order of splitting.

Finally, the two sorted halves are merged.

**Time Complexity Analysis**

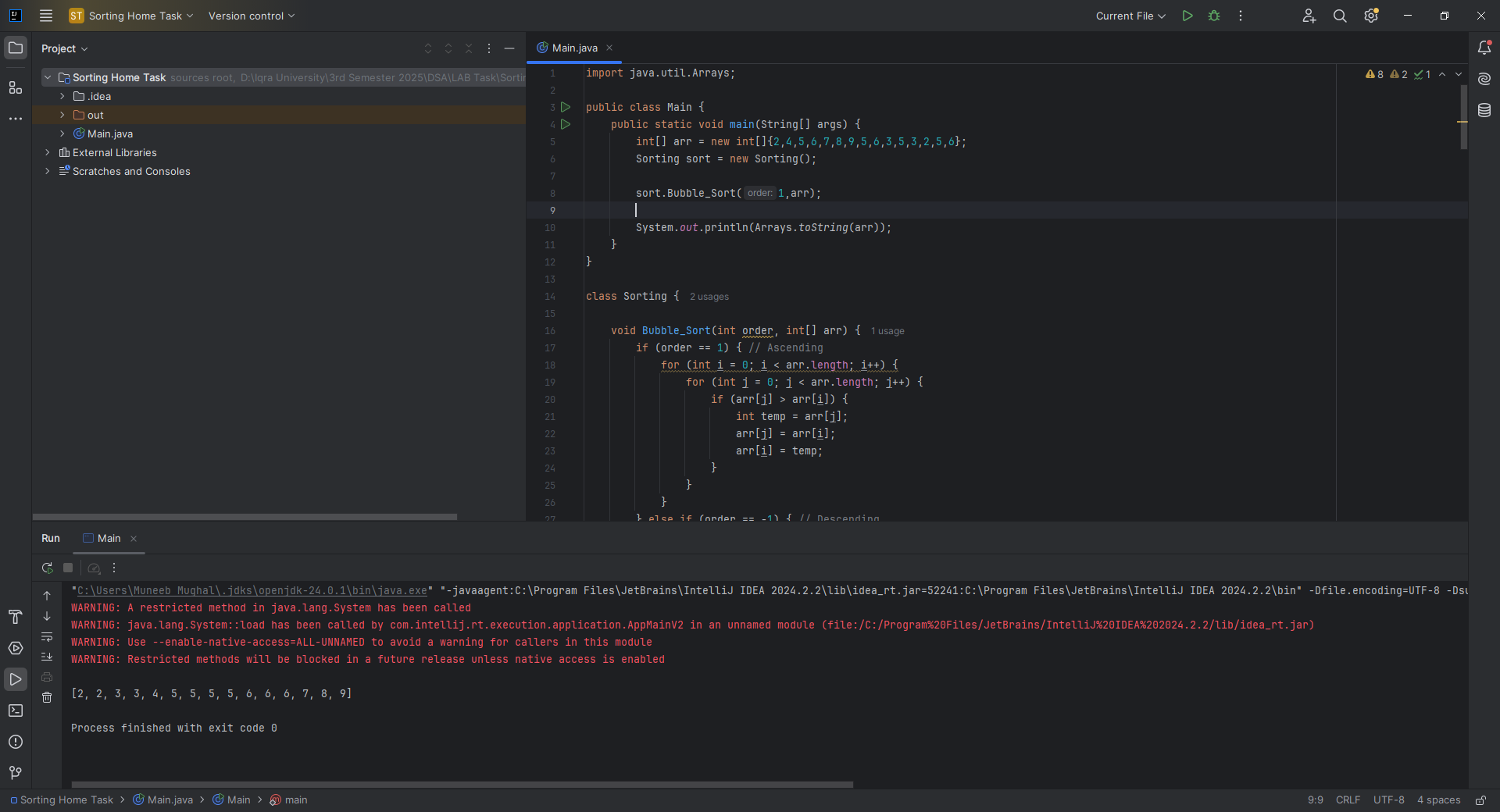
Merge Sort’s efficiency comes from how it breaks down and builds up the sorted array:

| **Input Size** | **Merge Sort (n log n)** | **Bubble/Insertion Sort (n²)** |
| --- | --- | --- |
| 10 | ~33 operations | ~100 operations |
| 1000 | ~10,000 operations | ~1,000,000 operations |

**HOME TASK 2**

**CODE:**

void quickSort(int[] arr) {  
 quickSortRecursive(arr, 0, arr.length - 1);  
}  
  
private void quickSortRecursive(int[] arr, int low, int high) {  
 if (low < high) {  
 int pi = quickPartition(arr, low, high);  
 quickSortRecursive(arr, low, pi - 1);  
 quickSortRecursive(arr, pi + 1, high);  
 }  
}  
  
private int quickPartition(int[] arr, int low, int high) {  
 int pivot = arr[high];  
 int i = low - 1;  
 for (int j = low; j < high; j++) {  
 if (arr[j] < pivot) {  
 i++;  
 int temp = arr[i]; arr[i] = arr[j]; arr[j] = temp;  
 }  
 }  
 int temp = arr[i + 1]; arr[i + 1] = arr[high]; arr[high] = temp;  
 return i + 1;  
}

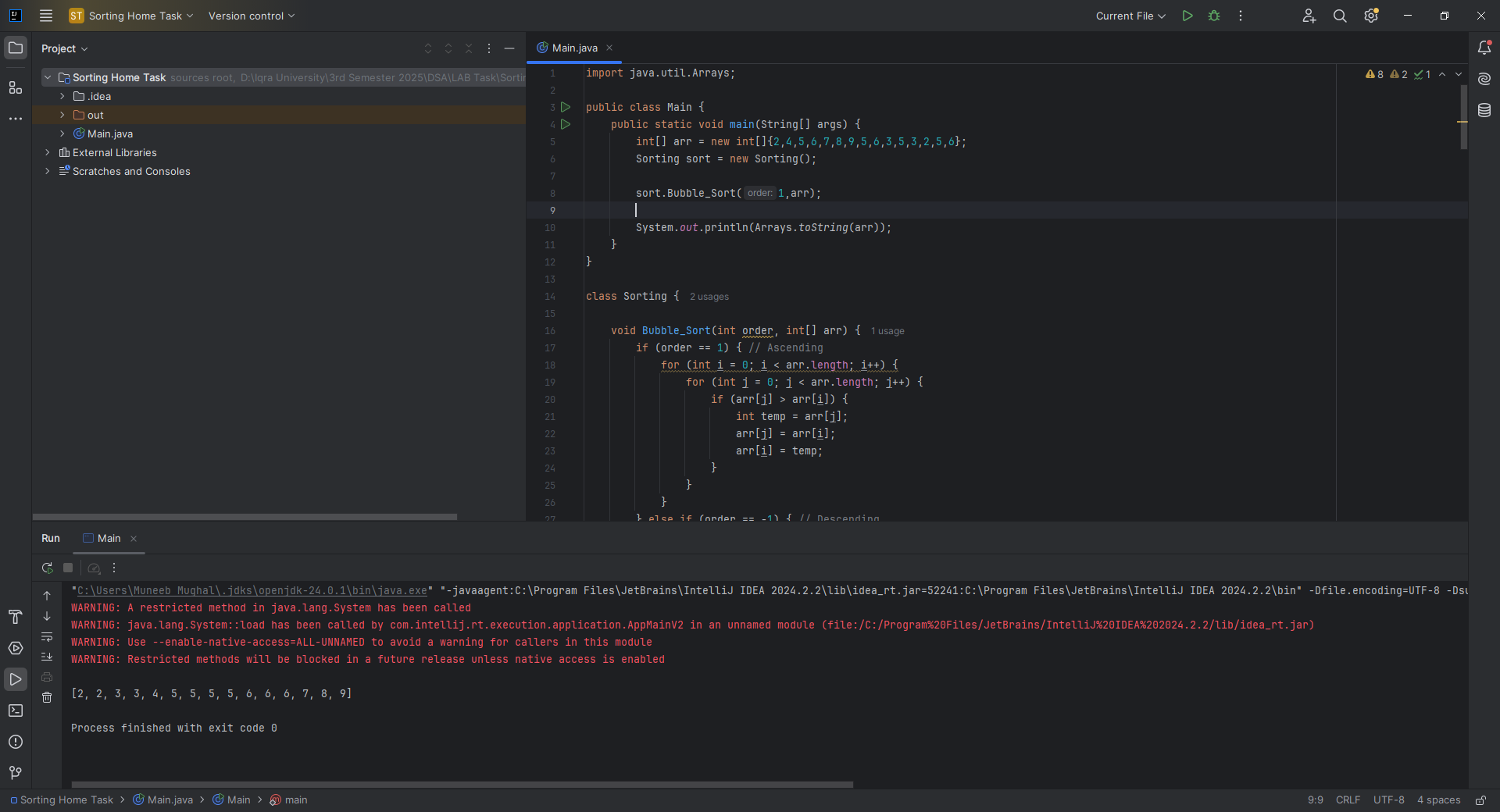


| **Case** | **Time Complexity** | **Reason** |
| --- | --- | --- |
| **Best Case** | O(n log n) | Pivot always splits the array into equal halves. |
| **Average Case** | O(n log n) | Pivot *usually* splits array reasonably well (not perfectly even, but not too unbalanced). |
| **Worst Case** | O(n²) | Pivot is the smallest or largest element every time, leading to highly unbalanced splits. |

**HOME TASK 3**

**CODE:**

void heapSort(int[] arr) {  
 int n = arr.length;  
  
 // Build max heap  
 for (int i = n / 2 - 1; i >= 0; i--) heapify(arr, n, i);  
  
 // Extract elements  
 for (int i = n - 1; i > 0; i--) {  
 int temp = arr[0]; arr[0] = arr[i]; arr[i] = temp;  
 heapify(arr, i, 0);  
 }  
}  
  
private void heapify(int[] arr, int heapSize, int root) {  
 int largest = root;  
 int left = 2 \* root + 1;  
 int right = 2 \* root + 2;  
  
 if (left < heapSize && arr[left] > arr[largest]) largest = left;  
 if (right < heapSize && arr[right] > arr[largest]) largest = right;  
  
 if (largest != root) {  
 int temp = arr[root];  
 arr[root] = arr[largest]; arr[largest] = temp;  
 heapify(arr, heapSize, largest);  
 }  
}



**Heap Sort** has:

* **Time Complexity:**
  + **O(n log n)** in **best**, **average**, and **worst** cases.
* **Memory Efficiency:**
  + **In-place sorting**: requires **O(1)** extra space (no additional arrays needed).

**Efficient for large datasets** where memory is limited.  
Not stable (doesn't preserve order of equal elements).