**day16\_107856406\_dsdipt\_sudipto\_9july2025**

**Employee Code:** 107856406

**Login ID:** dsdipt

**Email :** dsdipt@amazon.com

**Name:** Sudipto Das

**Date:** 9 July 2025 (Day 16)

### ***Task 1: Selection Sort Algorithm (Ascending Order)***

**Input:** An array arr of size n  
**Output:** Sorted array in ascending order

**Step-by-Step Algorithm:**

1. **Start**
2. **Repeat from i = 0 to n - 2** (i.e., for each position in the array except the last):
   * Assume the element at index i is the **minimum**.  
      → minIndex = i
   * Loop from j = i + 1 to n - 1:
     + If arr[j] < arr[minIndex], then:  
        → Update minIndex = j
   * After inner loop ends:
     + If minIndex != i, then swap arr[i] with arr[minIndex]  
        → swap(arr[i], arr[minIndex])
3. **Repeat until entire array is sorted**
4. **End**

### ***Task 2: Selection Sort pseudo code***

public class PseudocodeSelectionSort {

procedure selectionSort(array)

n ← length of array

for i ← 0 to n - 2 do

minIndex ← i

for j ← i + 1 to n - 1 do

if array[j] < array[minIndex] then

minIndex ← j

end if

end for

if minIndex ≠ i then

// Swap array[i] and array[minIndex]

temp ← array[i]

array[i] ← array[minIndex]

array[minIndex] ← temp

end if

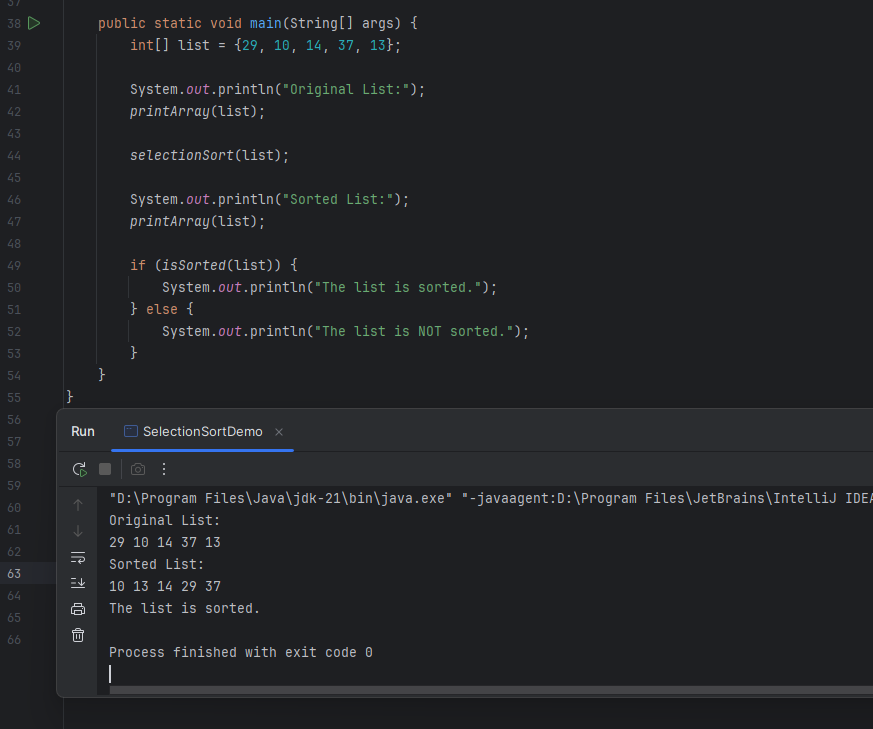
end for

end procedure

}

### ***Task 3: Implement Selection Sort***





### ***Task 4: Bubble Sort Algorithm***

**Input:** An array arr of size n  
**Output:** Sorted array in ascending order

**Step-by-Step Algorithm:**

1. **Start**
2. **Repeat from i = 0 to n - 2** (Outer loop to control number of passes)
   * **Repeat from j = 0 to n - i - 2** (Inner loop to compare adjacent elements)
     + If arr[j] > arr[j + 1] then  
        → Swap arr[j] and arr[j + 1]
3. **Repeat until no more swaps are needed or array is sorted**
4. **End**

### ***Task 5: Bubble Sort pseudo code***

procedure bubbleSort(array)

n ← length of array

for i ← 0 to n - 2 do

swapped ← false

for j ← 0 to n - i - 2 do

if array[j] > array[j + 1] then

// Swap array[j] and array[j + 1]

temp ← array[j]

array[j] ← array[j + 1]

array[j + 1] ← temp

swapped ← true

end if

end for

// If no elements were swapped, the array is sorted

if swapped = false then

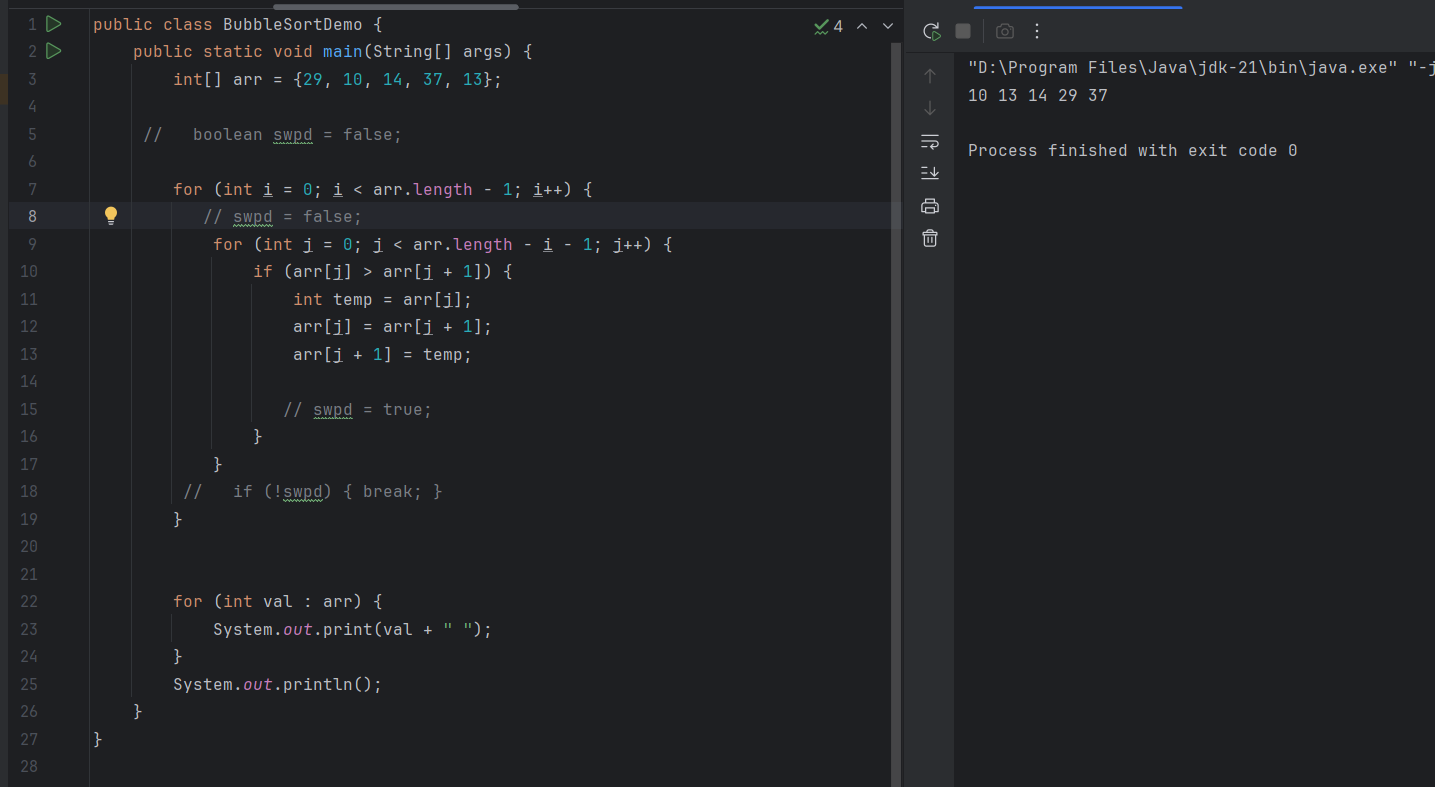
break

end if

end for

end procedure

### ***Task 6: Implement Bubble Sort***



### ***Task 7: Insertion Sort Algorithm***

**Input:** An array arr of size n  
**Output:** arr sorted in ascending order

**Algorithm:**

1. Start
2. Repeat from i = 1 to n - 1:  
    (Assume the first element is already sorted)
   * Store the current element:  
      → key = arr[i]
   * Set j = i - 1
   * While j >= 0 and arr[j] > key:  
      → Shift arr[j] one position to the right:  
      arr[j + 1] = arr[j]  
      → Decrement j = j - 1
   * Place key at correct position:  
      → arr[j + 1] = key
3. Repeat until all elements are inserted in sorted order
4. End

### ***Task 8: Insertion Sort pseudo code***

procedure insertionSort(array)

n ← length of array

for i ← 1 to n - 1 do

key ← array[i]

j ← i - 1

while j ≥ 0 and array[j] > key do

array[j + 1] ← array[j] // Shift element right

j ← j - 1

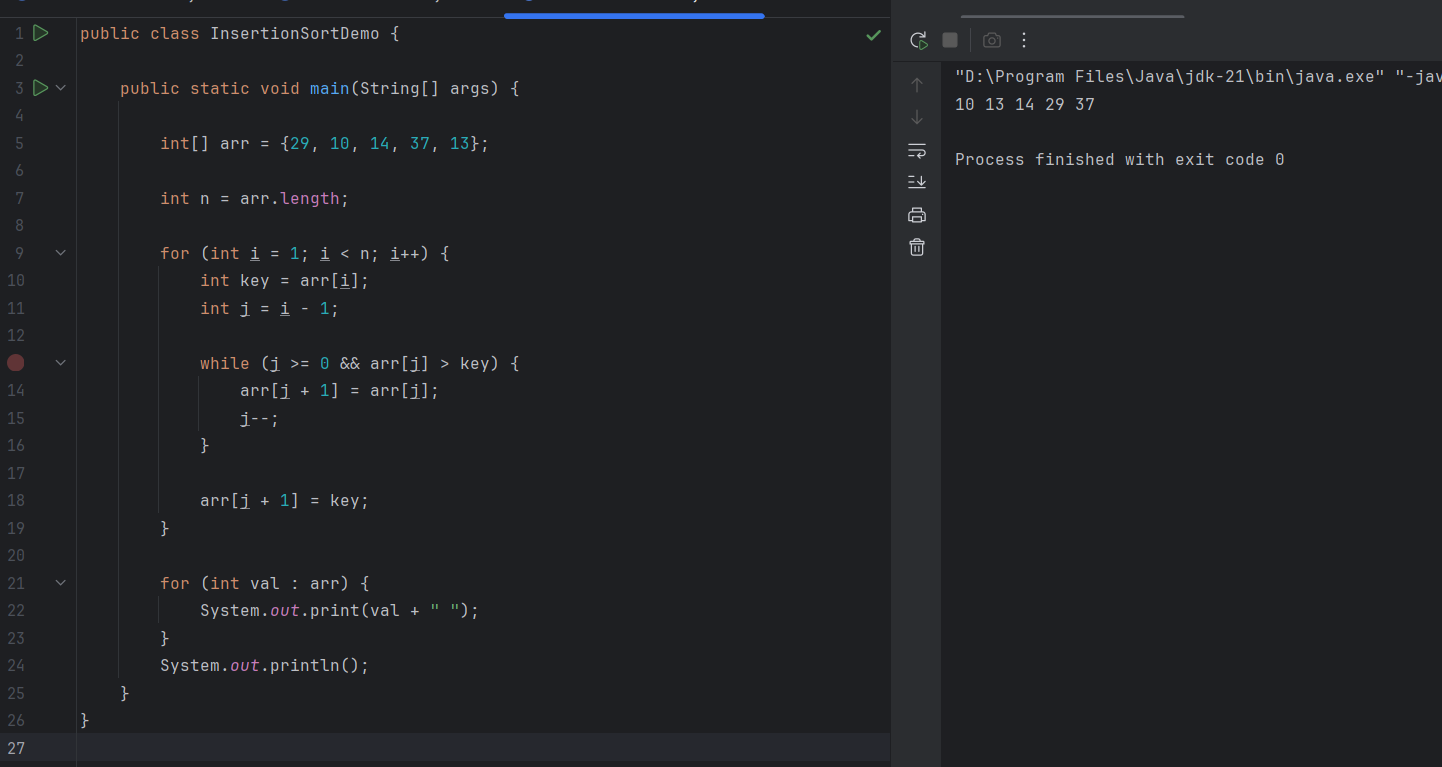
end while

array[j + 1] ← key // Insert key at correct position

end for

end procedure

### ***Task 9: Implement Insertion Sort***



### ***Task 10: Advantages and Disadvantages of Bubble Sort Algorithm***

**Advantages of Bubble Sort:**

* Very easy to understand and implement.
* In-place algorithm – no extra memory needed.
* Stable sort – maintains order of equal elements.
* Can be optimized to stop early if already sorted.

**Disadvantages of Bubble Sort:**

* Very slow for large data (O(n²) time).
* Performs too many unnecessary swaps.
* Not adaptive unless manually optimized.
* Not used in real-world applications due to poor performance.

### ***Task 11: Merge Sort Algorithm***

**Start**

If n <= 1, the array is already sorted — return

**Divide** the array into two halves:

* Find the middle index: mid = n / 2
* Left half: arr[0...mid-1]
* Right half: arr[mid...n-1]

**Recursively** apply Merge Sort on the left and right halves

**Merge** the two sorted halves into a single sorted array:

* Compare elements from both halves
* Copy the smaller element into the original array
* Continue until all elements are merged

**End**

### ***Task 12: Merge Sort Pseudocode***

procedure mergeSort(array)

if length of array > 1 then

mid ← length of array / 2

leftHalf ← array[0 ... mid - 1]

rightHalf ← array[mid ... end]

mergeSort(leftHalf)

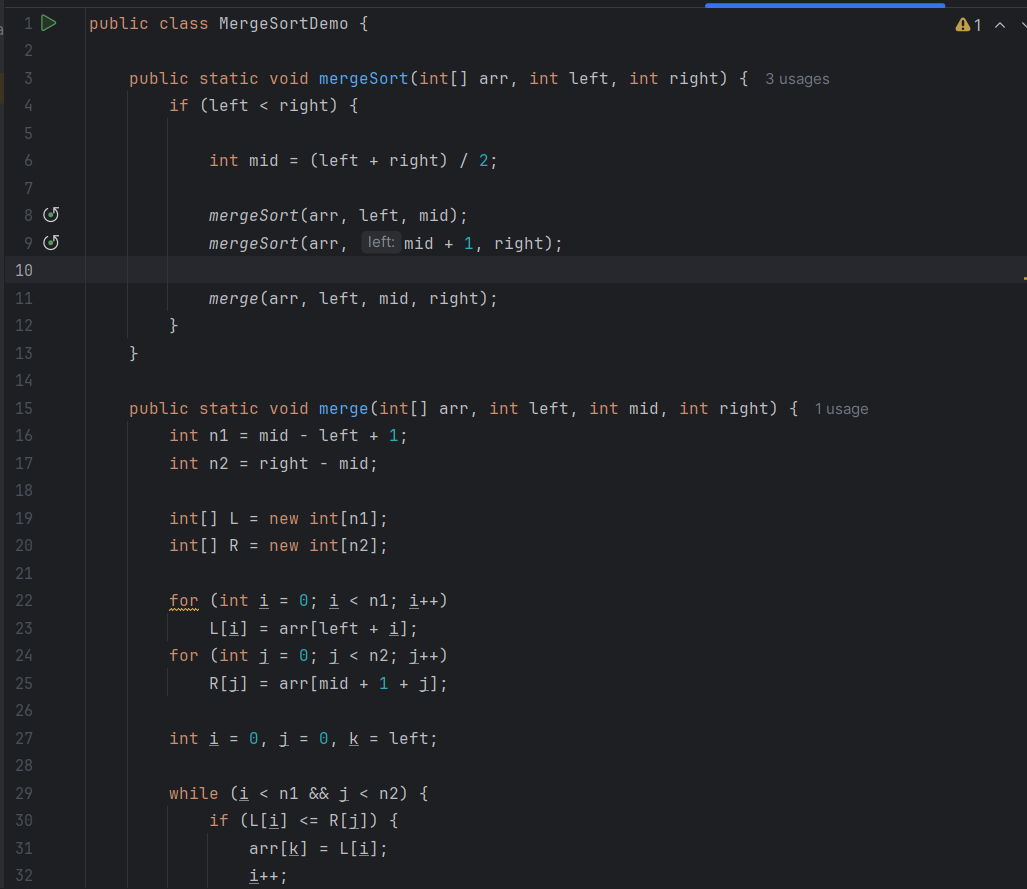
mergeSort(rightHalf)

merge(leftHalf, rightHalf, array)

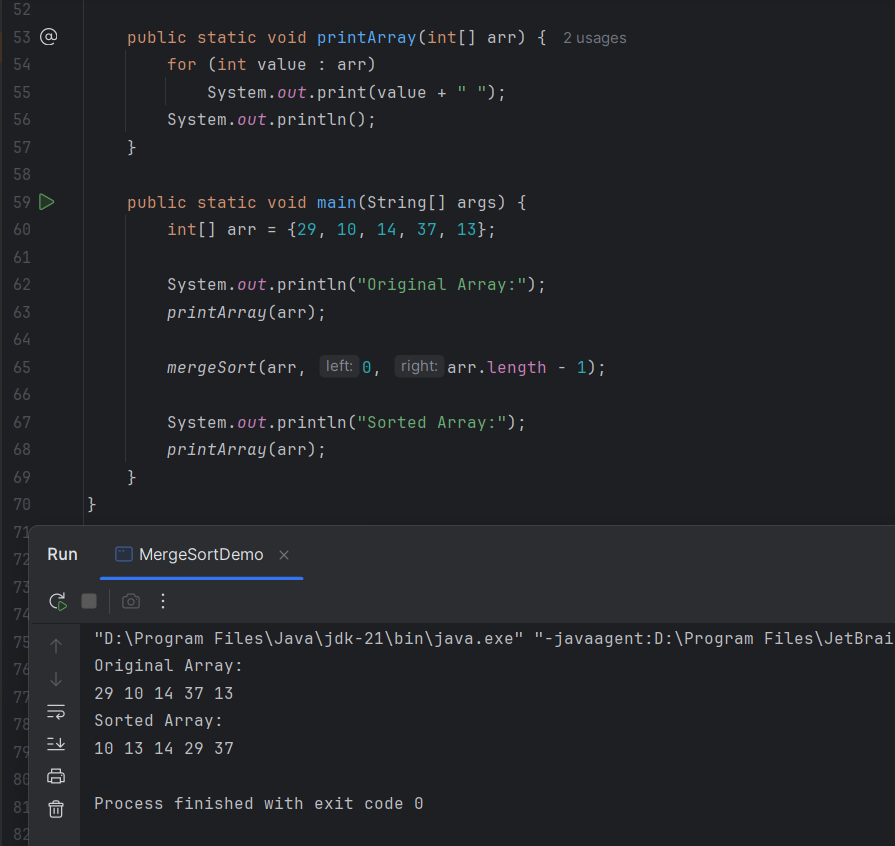
end if

end procedure

### ***Task 13: Implement Merge Sort***







### ***Task 14: Quick Sort Algorithm***

**Start**

If low < high, then:

1. **Partition the array:**
   * Choose a pivot element (commonly the last element)
   * Rearrange elements so that:
     + Elements less than pivot are on the left
     + Elements greater than pivot are on the right
   * Let the pivotIndex be the final position of the pivot
2. **Recursively apply QuickSort on:**
   * Left subarray → QuickSort(arr, low, pivotIndex - 1)
   * Right subarray → QuickSort(arr, pivotIndex + 1, high)

**End**

### ***Task 15: Quick Sort Pseudocode***

procedure quickSort(array, low, high)

if low < high then

pivotIndex ← partition(array, low, high)

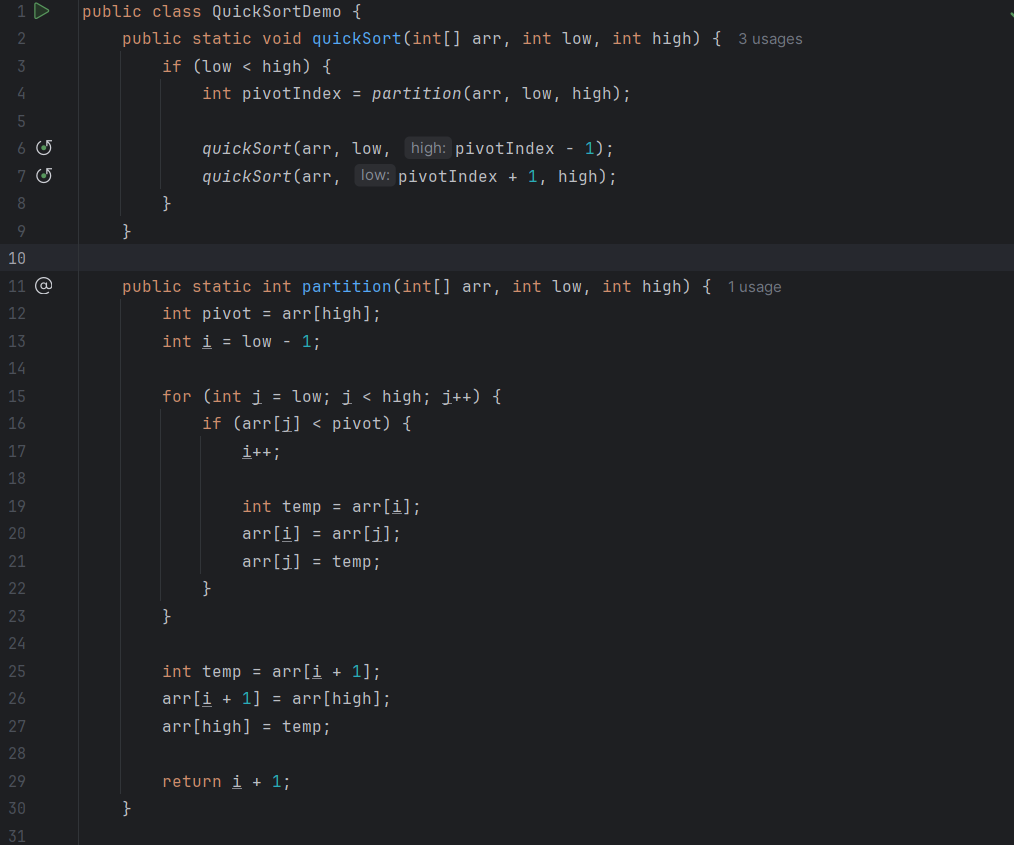
quickSort(array, low, pivotIndex - 1)

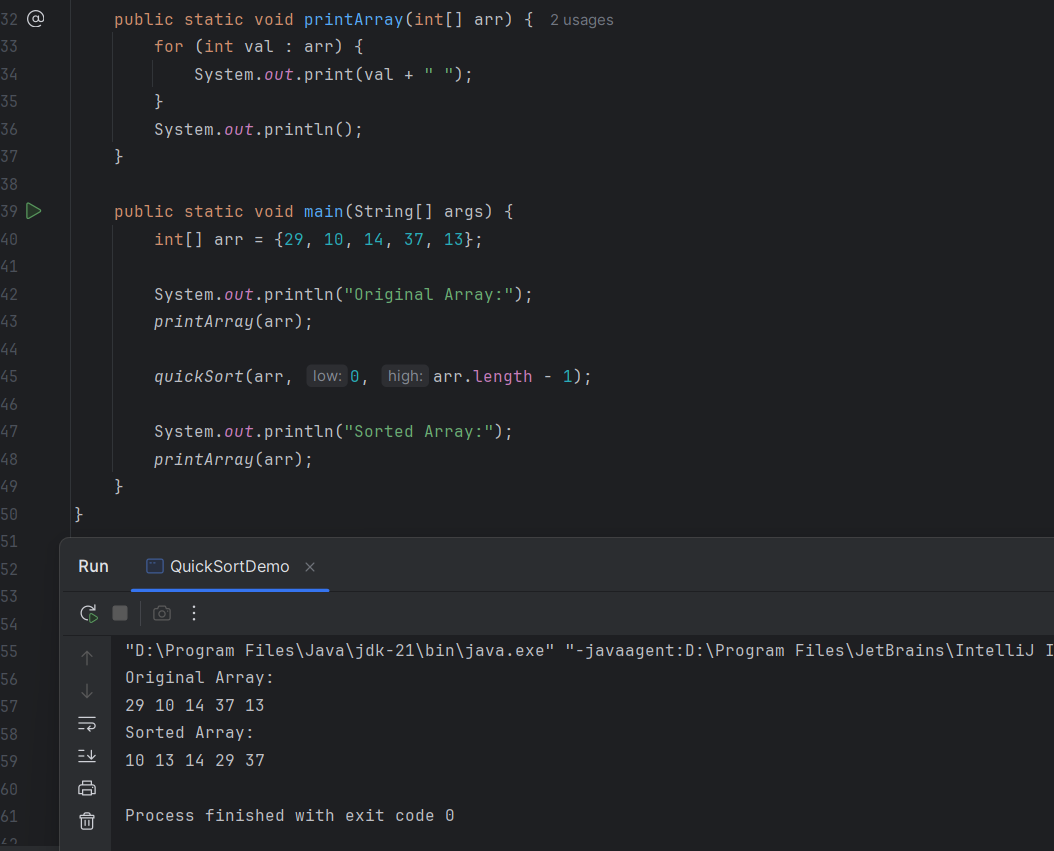
quickSort(array, pivotIndex + 1, high)

end if

end procedure

### ***Task 16: Implement Quick Sort***





### ***Task 17: Comparison of Merge Sort vs Bubble Sort (Time Complexity)***

**🔹 Merge Sort:**

* **Best Case:** O(n log n)
* **Average Case:** O(n log n)
* **Worst Case:** O(n log n)
* **Stable Sort:** Yes
* **Divide and Conquer:** Yes
* **Space Complexity:** O(n)

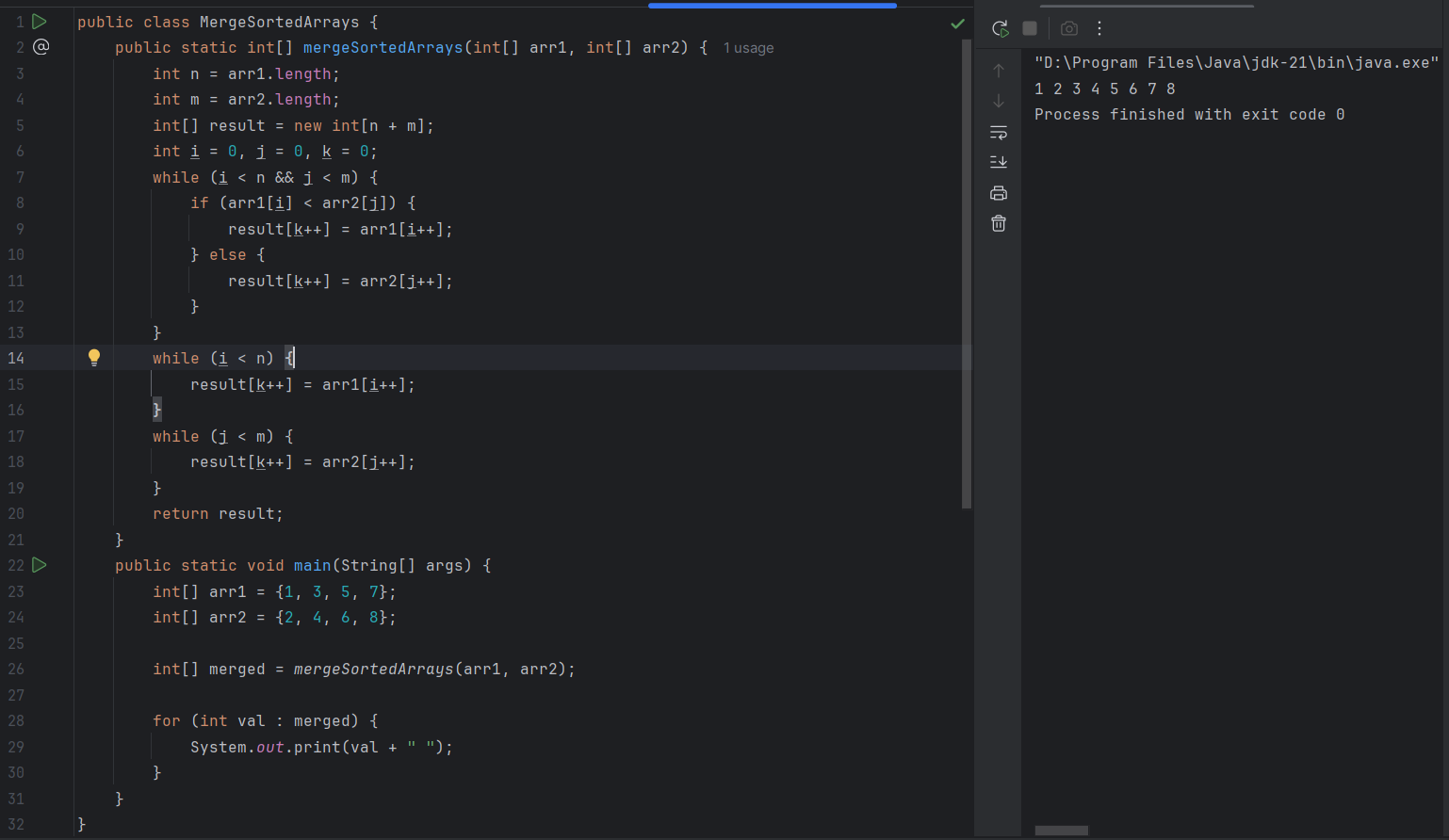
**🔹 Bubble Sort:**

* **Best Case:** O(n) (when array is already sorted, with optimization)
* **Average Case:** O(n²)
* **Worst Case:** O(n²)
* **Stable Sort:** Yes
* **Simple and Easy:** Yes
* **Space Complexity:** O(1)

**🔍 Conclusion:**

* **Time Complexity:** Merge Sort is **much more efficient** for larger datasets.
* **Use Bubble Sort only** for educational purposes or very small datasets.
* **Merge Sort is preferred** in most real-world applications due to better performance.

***Task 18: Time Complexity of Merge Operation Between Two Sorted Arrays***

******

**Time Complexity:**

* O(m + n)
* Because each element of both arrays is visited at most once, regardless of their values.

**Space Complexity:**

* If done **in-place (as required by LeetCode)** → **O(1)**
* If extra array used → **O(m + n)**

***Task 19: Fixing Stack Overflow in Recursive Code***

public int calc(int n) {

if (n == 0) return 0;

return n + calc(n); // Recursive call with same value `n`

}

public class RecLoop {

public int calc(int n) {

if (n == 0) return 0; // Base case

return n + calc(n - 1); // Decreasing `n` to avoid infinite recursion

}

public static void main(String[] args) {

System.out.println(new RecLoop().calc(10)); // Output: 55

}

}

***Task 20: Can we apply Binary Search on an Unsorted Tree?***

**No,** binary search cannot be applied to an unsorted binary tree.

**Explanation:**  
Binary Search relies on sorted data to eliminate half the search space at each step. In a binary tree (that is not a Binary Search Tree), nodes can be arranged in any order, so there’s no guarantee about where a value might be.  
To search in such a tree, you'd need to do a full traversal (like DFS or BFS), which is not binary search.