**ID – hrajranj**

**Day 16 – 09th July 2025**

**Task 1:**

**Write an algorithm / steps for selection sort.**

**Answer:**

Here are the steps for the selection sort algorithm:

1. Start with the first element of the array as the current position.
2. Find the smallest element in the unsorted portion of the array.
3. Swap the smallest element found with the element at the current position.
4. Move to the next position in the array.
5. Repeat steps 2-4 until the entire array is sorted.

**Task 2:**

**Write a pseudo code for the selection sort**

**Answer:**

selection  sort (A)

for i =1 to n - 1

min j = i

min x = A[i]

for j = i+1 to n

if A[j] <min x then

min j = j

min x = A[j]

A[min j] = A[i]

A[i] = min x

**Task 3:**

**Wap to make sure your list is sorted using selection sort.**

**Answer:**

import java.util.\*;

class Selectionsorting {

   public static void printArray(int arr[]) {

       for(int i=0; i<arr.length; i++) {

           System.out.print(arr[i]+" ");

       }

       System.out.println();

   }

   public static void main(String args[]) {

       int arr[] = {7, 8, 1, 3, 2};

       //selection sort

       for(int i=0; i<arr.length-1; i++) {

           int smallest = i;

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

               if(arr[j] < arr[smallest]) {

                   smallest = j;

               }

           }

           //swap

           int temp = arr[smallest];

           arr[smallest] = arr[i];

           arr[i] = temp;

       }

       printArray(arr);

   }

}

**Bubble Sort:**

**Task 4:**

**Write algorithm for the Bubble sort.**

**Ans:**

Algorithm:

1. **Start**
2. Let A be the array of size n
3. Repeat steps 4–6 for i = 0 to n - 1  
   4.  Set a flag swapped = false  
   5.  Repeat steps 6 for j = 0 to n - i - 2  
   6.    If A[j] > A[j + 1], then  
         Swap A[j] and A[j + 1]  
         Set swapped = true  
   7.  If swapped == false, then **break** (array is already sorted)
4. **End**

**Task 5:**

**Write [psedo code for the bubble sort**

**Ans:**

Procedure BubbleSort(A, n)

for i ← 0 to n - 1 do

swapped ← false

for j ← 0 to n - i - 2 do

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

temp ← A[j]

A[j] ← A[j + 1]

A[j + 1] ← temp

swapped ← true

end if

end for

if swapped = false then

break

end if

end for

End Procedure

**Task 6:**

**Wap to make sure your list is sorted using Bubble sort.**

**Ans:**

import java.util.\*;

class Bubblesort {

   public static void printArray(int arr[]) {

       for(int i=0; i<arr.length; i++) {

           System.out.print(arr[i]+" ");

       }

       System.out.println();

   }

   public static void main(String args[]) {

       int arr[] = {7, 8, 1, 3, 2};

       //bubble sort

       for(int i=0; i<arr.length-1; i++) {

           for(int j=0; j<arr.length-i-1; j++) {

               if(arr[j] > arr[j+1]) {

                   //swap

                   int temp = arr[j];

                   arr[j] = arr[j+1];

                   arr[j+1] = temp;

               }

           }

       }

       printArray(arr);

   }

}

**Insertion sort:**

**Task 7:**

**Write algorithm for the Insertion sort.**

**Ans:**

**Steps:**

1. **Start**
2. Let A be an array of size n
3. **For** i = 1 to n - 1  
     a. Set key = A[i]  
     b. Set j = i - 1  
     c. **While** j >= 0 and A[j] > key  
       i. Move A[j] to A[j + 1]  
       ii. Decrease j by 1  
     d. Set A[j + 1] = key
4. **End**

**Task 8:**

**Write psedo code for the Insertion sort**

**Ans:**

Procedure InsertionSort(A, n)

for i ← 1 to n - 1 do

key ← A[i]

j ← i - 1

while j ≥ 0 and A[j] > key do

A[j + 1] ← A[j]

j ← j - 1

end while

A[j + 1] ← key

end for

End Procedure

**Task 9:**

**Wap to make sure your list is sorted using Insertion sort.**

**Ans:**

import java.util.\*;

class Insertionsorting {

   public static void printArray(int arr[]) {

       for(int i=0; i<arr.length; i++) {

           System.out.print(arr[i]+" ");

       }

       System.out.println();

   }

   public static void main(String args[]) {

       int arr[] = {7, 8, 1, 3, 2};

       //insertion sort

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

           int current = arr[i];

           int j = i - 1;

               while(j >= 0 && arr[j] > current) {

                   //Keep swapping

                   arr[j+1] = arr[j];

                   j--;

               }

           arr[j+1] = current;

       }

       printArray(arr);

   }

}

**Task 10:**

**What are the advantages and disadvantages of Bubble sort Algo?**

**List them**

**Ans:**

**Advantages of Bubble Sort**

1. **Simple and Easy to Implement**
   * The logic is straightforward and easy to understand, making it great for beginners learning sorting.
2. **In-Place Sorting**
   * It requires only a constant amount (O(1)) of extra memory; no need for additional data structures.
3. **Stable Sort**
   * Maintains the relative order of equal elements, which is useful when sorting records with multiple fields.
4. **Best Case Optimization**
   * With an optimization (checking if any swaps occurred), it can finish early on already sorted arrays — O(n) in best case.

**Disadvantages of Bubble Sort**

1. **Inefficient for Large Datasets**
   * Time complexity is **O(n²)** in worst and average cases, making it impractical for large arrays.
2. **Unnecessarily Repetitive**
   * Continues comparing even if most of the array is already sorted (unless optimized).
3. **Not Adaptive by Default**
   * Without modification, it doesn't adapt well to partially sorted data.
4. **High Number of Swaps**
   * It performs many unnecessary swaps compared to algorithms like Insertion Sort or Selection Sort.

**Task 11:**

**Algo for merge sort,**

**Ans:**

**Algorithm**

To sort an array A using the **Divide and Conquer** technique.

**Steps:**

1. **Start**
2. If the array has **1 or 0 elements**, it is already sorted.
3. Otherwise:

* Divide the array into two halves: left and right
* Recursively apply **Merge Sort** on left
* Recursively apply **Merge Sort** on right
* **Merge** the two sorted halves into a single sorted array

1. **End**

**Task 12**

**pseudo code for merge sort,**

**Ans:**

MERGE\_SORT(arr, start, end):

if start < end:

mid = (start + end) / 2

MERGE\_SORT(arr, start, mid) // Sort left half

MERGE\_SORT(arr, mid + 1, end) // Sort right half

MERGE(arr, start, mid, end) // Merge sorted halves

MERGE(arr, start, mid, end):

Create temporary arrays: left[] and right[]

Copy data to left[] and right[] from arr[start..mid] and arr[mid+1..end]

Merge left[] and right[] back into arr[start..end] in sorted order

**Task 13**

**Wap to make sure your list is sorted using Merge sort.**

**Ans:**

import java.util.Arrays;

public class MergeSort {

    public static void main(String[] args) {

        // Example list

        int[] arr = {38, 27, 43, 3, 9, 82, 10};

        System.out.println("Original List: " + Arrays.toString(arr));

        // Sort using merge sort

        mergeSort(arr, 0, arr.length - 1);

        System.out.println("Sorted List using Merge Sort: " + Arrays.toString(arr));

    }

    // Merge Sort Function

    public static void mergeSort(int[] arr, int left, int right) {

        if (left < right) {

            int mid = left + (right - left) / 2;

            // Sort left half

            mergeSort(arr, left, mid);

            // Sort right half

            mergeSort(arr, mid + 1, right);

            // Merge sorted halves

            merge(arr, left, mid, right);

        }

    }

    // Merge Function

    public static void merge(int[] arr, int left, int mid, int right) {

        // Sizes

        int n1 = mid - left + 1;

        int n2 = right - mid;

        // Temp arrays

        int[] L = new int[n1];

        int[] R = new int[n2];

        // Copy data

        for (int i = 0; i < n1; i++) L[i] = arr[left + i];

        for (int j = 0; j < n2; j++) R[j] = arr[mid + 1 + j];

        // Merge temp arrays back into original

        int i = 0, j = 0, k = left;

        while (i < n1 && j < n2)

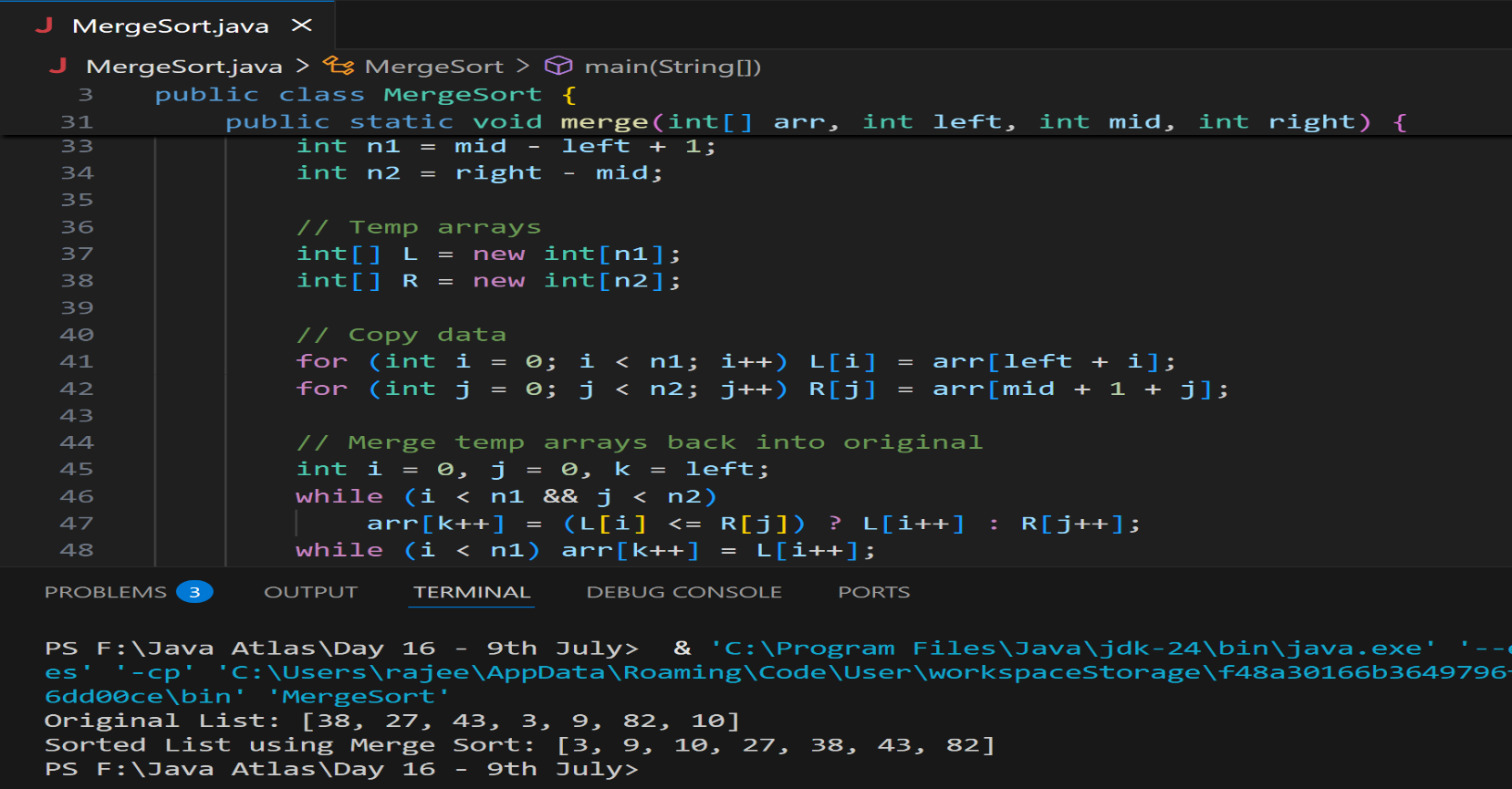
            arr[k++] = (L[i] <= R[j]) ? L[i++] : R[j++];

        while (i < n1) arr[k++] = L[i++];

        while (j < n2) arr[k++] = R[j++];

    }

}



**Task 14:**

**Algo fro quick sort**

**Ans:**

Step 1: Start

Step 2: Choose a pivot element from the array (typically the last element)

Step 3: Partition the array around the pivot

- Move all elements smaller than pivot to its left

- Move all elements greater than pivot to its right

Step 4: Recursively apply Quick Sort to the left sub-array

Step 5: Recursively apply Quick Sort to the right sub-array

Step 6: Done (when sub-arrays have size 0 or 1)

**Task 15:**

**Pseudo code for quick sort**

QUICK\_SORT(array, low, high)

if low < high then

pivotIndex ← PARTITION(array, low, high)

QUICK\_SORT(array, low, pivotIndex - 1)

QUICK\_SORT(array, pivotIndex + 1, high)

PARTITION(array, low, high)

pivot ← array[high] // choose the last element as pivot

i ← low - 1 // index of smaller element

for j ← low to high - 1 do

if array[j] < pivot then

i ← i + 1

SWAP(array[i], array[j])

SWAP(array[i + 1], array[high]) // place pivot in correct position

return i + 1

**Task 16:**

**Code for Quick sort**

import java.util.Arrays;

public class QuickSort {

    public static void main(String[] args) {

        int[] arr = {10, 7, 8, 9, 1, 5};

        System.out.println("Original List: " + Arrays.toString(arr));

        // Call quick sort

        quickSort(arr, 0, arr.length - 1);

        System.out.println("Sorted List using Quick Sort: " + Arrays.toString(arr));

    }

    // Quick Sort Function

    static void quickSort(int[] arr, int low, int high) {

        if (low < high) {

            // Partition the array

            int pi = partition(arr, low, high);

            // Recursively sort elements before and after partition

            quickSort(arr, low, pi - 1);

            quickSort(arr, pi + 1, high);

        }

    }

    // Partition Function

    static int partition(int[] arr, int low, int high) {

        int pivot = arr[high];  // choosing last element as pivot

        int i = (low - 1);       // index of smaller element

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

            // If current element is smaller than or equal to pivot

            if (arr[j] <= pivot) {

                i++;

                // Swap arr[i] and arr[j]

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

            }

        }

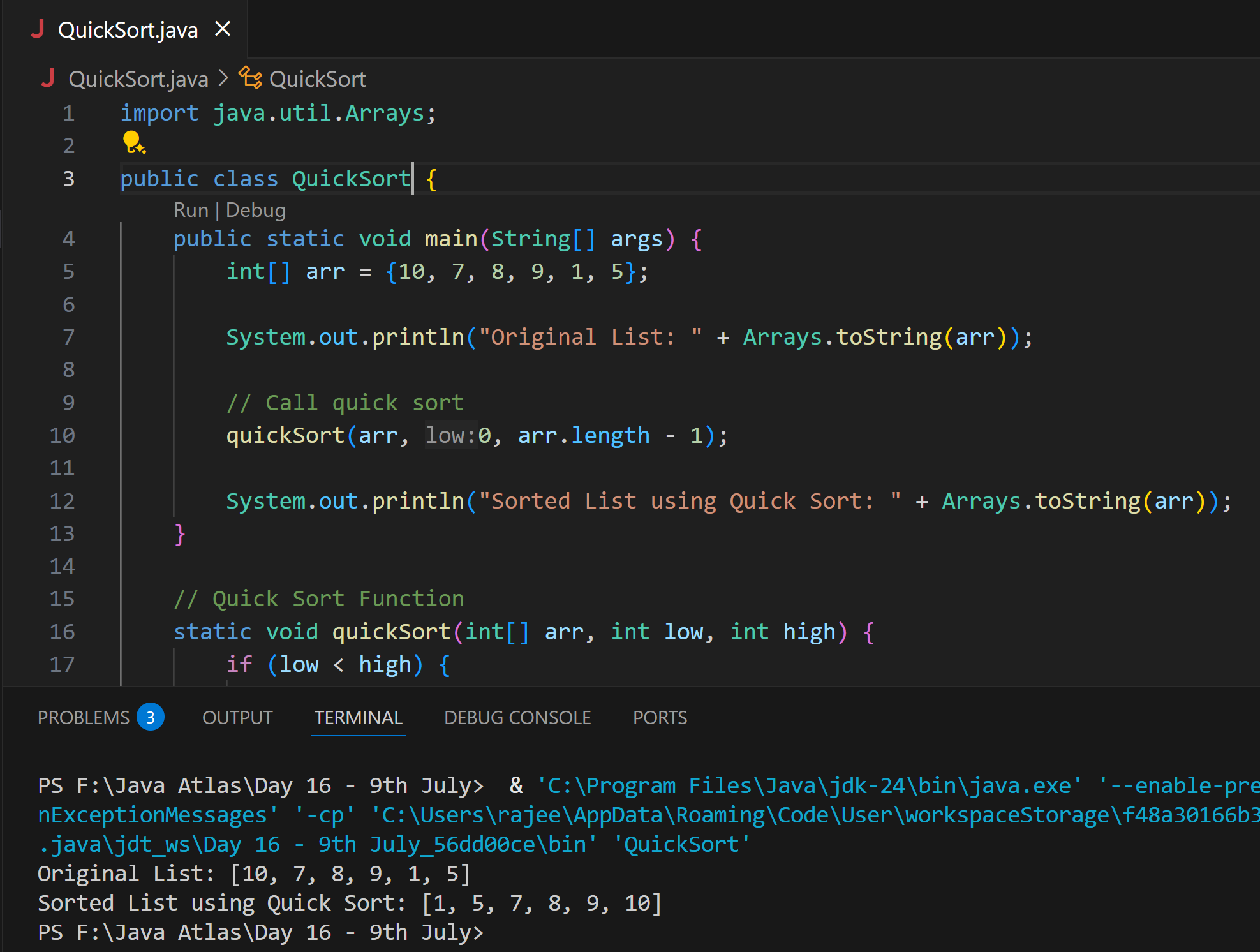
        // Swap arr[i+1] and pivot (arr[high])

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

        return i + 1;  // return the partition index

    }

}



**Task 17:**

**Which one is better merge or bubble in terms of time complexity?**

Note:  industrial applications like huge data bases … prefer merge sort.

Note:

Merge:  O (n log n)

Bubble: O (n square)

**Ans:** **Merge Sort is clearly better than Bubble Sort** in terms of time complexity, especially for large or unsorted data.

**Task 18:**

Leet code qn:

Find the time complexity of the given merge operation between two sorted arrays.

**Ans:**

public class MergeSortedArrays {

    public static void main(String[] args) {

        int[] arr1 = {1, 4, 6, 8};

        int[] arr2 = {2, 3, 5, 7, 9};

        int[] merged = merge(arr1, arr2);

        System.out.print("Merged array: ");

        for (int num : merged) {

            System.out.print(num + " ");

        }

    }

    public static int[] merge(int[] a, int[] b) {

        int n = a.length;

        int m = b.length;

        int[] result = new int[n + m];

        int i = 0, j = 0, k = 0;

        while (i < n && j < m) {

            if (a[i] < b[j]) {

                result[k++] = a[i++];

            } else {

                result[k++] = b[j++];

            }

        }

        while (i < n) {

            result[k++] = a[i++];

        }

        while (j < m) {

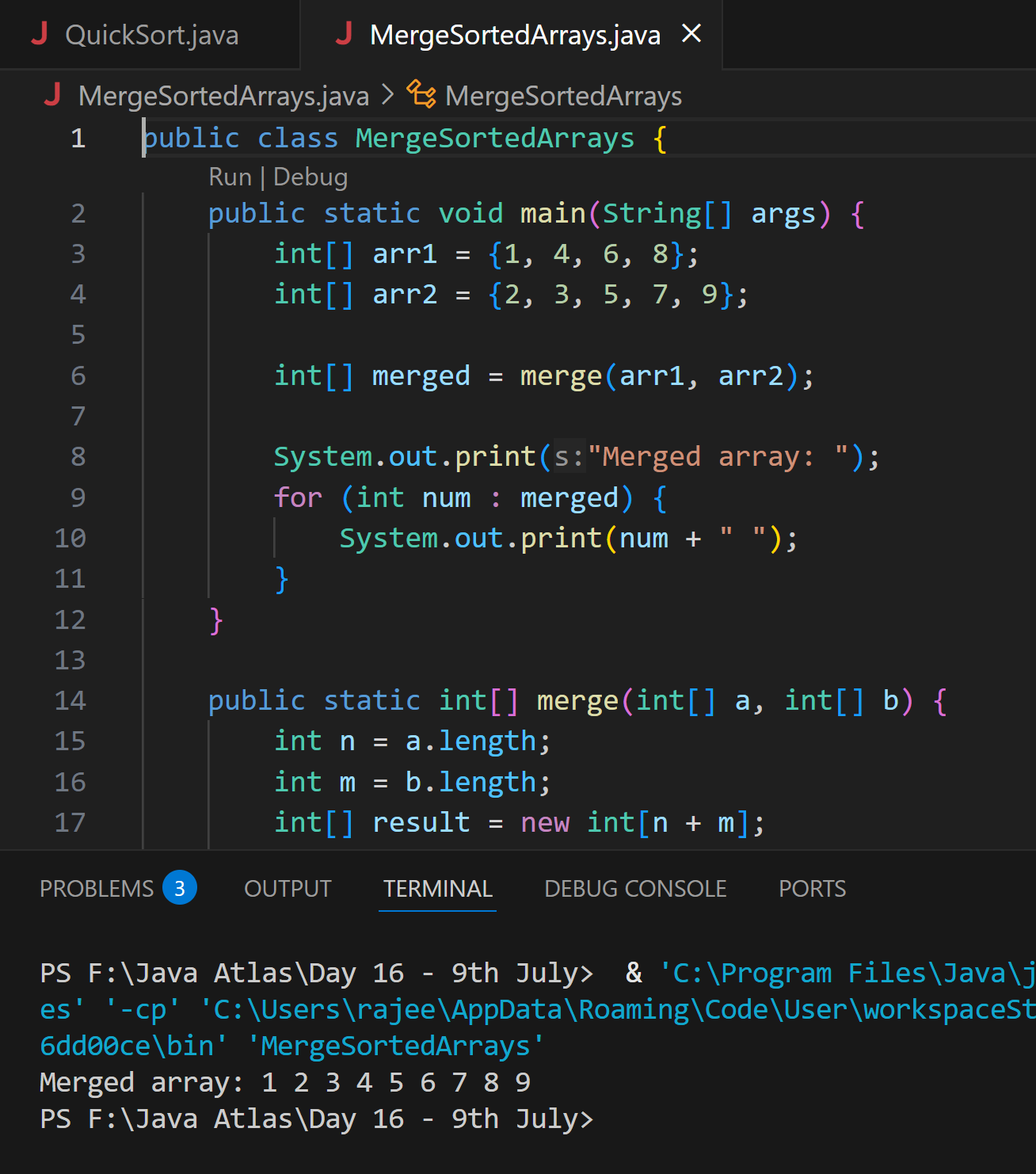
            result[k++] = b[j++];

        }

        return result;

    }

}



**Task 19:**

This code is going overflow of stack.. Can you plz help me fix it guys.. ☹️

public class RecLoop {

     public int calc(int n) {

        if (n == 0) return 0;

        return n + calc(n);

    }

    psvm(String[] args) {

        System.out.println(new RecLoop().calc(10));

    }

}

**Ans:**

The issue lies in this line inside the calc() method. We are not reducing the input n in the recursive call. So instead of getting closer to the base case (n == 0), we're calling calc(n) with the same value again and again. This causes StackOverflowError because infinite recursive calls means no base case reached and hence call stack overflows. We should decrement n in each recursive call: return n + calc(n - 1); Now n gets smaller each time, eventually reaching 0, and the recursion stops.

**Task 20:**

public class BinarySearchNew {

    public int search(int[] arr, int toFind) {

        int left = 0, right = arr.len - 1;

        while (left <= right) {

            int mid = left + (right - left) / 2;

            if (arr[mid] == toFind) {

                return mid;

            } else if (arr[mid] < toFind) {

                left = mid + 1;

            } else {

                right = mid - 1;

            }

        }

        return -1;

    }

}

 binary search function can we use in unsorted list?

Will it give you correct index no?

Ans:

No, we cannot use Binary Search on an unsorted list. Binary Search only works on sorted arrays/lists (ascending or descending). It relies on the middle element comparison and halving the search space based on sorted order.