Day 16

Employee ID: 201933938

Login ID: iamasif

Name: Shaik Asif

Task 1:

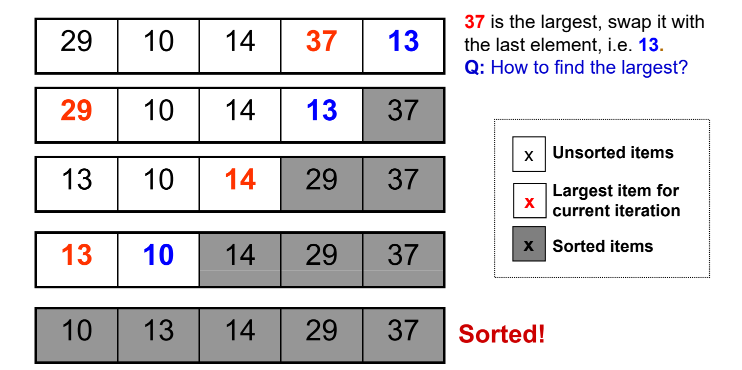
Write an algorithm / steps for selection sort.

1. Look through the list and find the largest element in the current range.
2. Swap that largest element with the last unsorted item.
3. Now, ignore the last item (because it’s sorted), and repeat the process for the rest of the list.

For expl :

Let’s take the list : [29, 10, 14, 37, 13]

1. The largest is 37 → swap it with the last element → [29, 10, 14, 13, 37]
2. Next, largest in [29, 10, 14, 13] is 29 → swap with 13 → [13, 10, 14, 29, 37]
3. Next, largest in [13, 10, 14] is 14 → swap with 14 (no change)
4. Next, largest in [13, 10] is 13 → swap with 10 → [10, 13, 14, 29, 37]



Task 2:

Write a pseudo code for the selection sort

class **Task2** {

// Method to perform selection sort

function selectionSort(array):

n = length of array

for i from 0 to n - 1:

minIndex = i

for j from i + 1 to n - 1:

if array[j] < array[minIndex]:

minIndex = j

// Swap the elements

temp = array[i]

array[i] = array[minIndex]

array[minIndex] = temp

}

11.13 to 11.17

Task 3:

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

**PROGRAM:**

// Class to perform Selection Sort

public class Task3 {

// Method to sort an array using Selection Sort

public static void selectionSort(int[] arr) {

int n = arr.length;

// Loop through each element in the array

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

int minIndex = i;

// Find the index of the smallest element in the unsorted part

for (int j = i + 1; j < n; j++) {

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

minIndex = j;

}

}

// Swap the found minimum element with the current element

int temp = arr[i];

arr[i] = arr[minIndex];

arr[minIndex] = temp;

}

}

// Method to print the array

public static void printArray(int[] arr) {

for (int value : arr) {

System.*out*.print(value + " ");

}

System.*out*.println();

}

// Main method to run the program

public static void main(String[] args) {

int[] numbers = {29, 10, 14, 37, 13};

System.*out*.println("Original array:");

*printArray*(numbers);

*selectionSort*(numbers); // Sorting the array

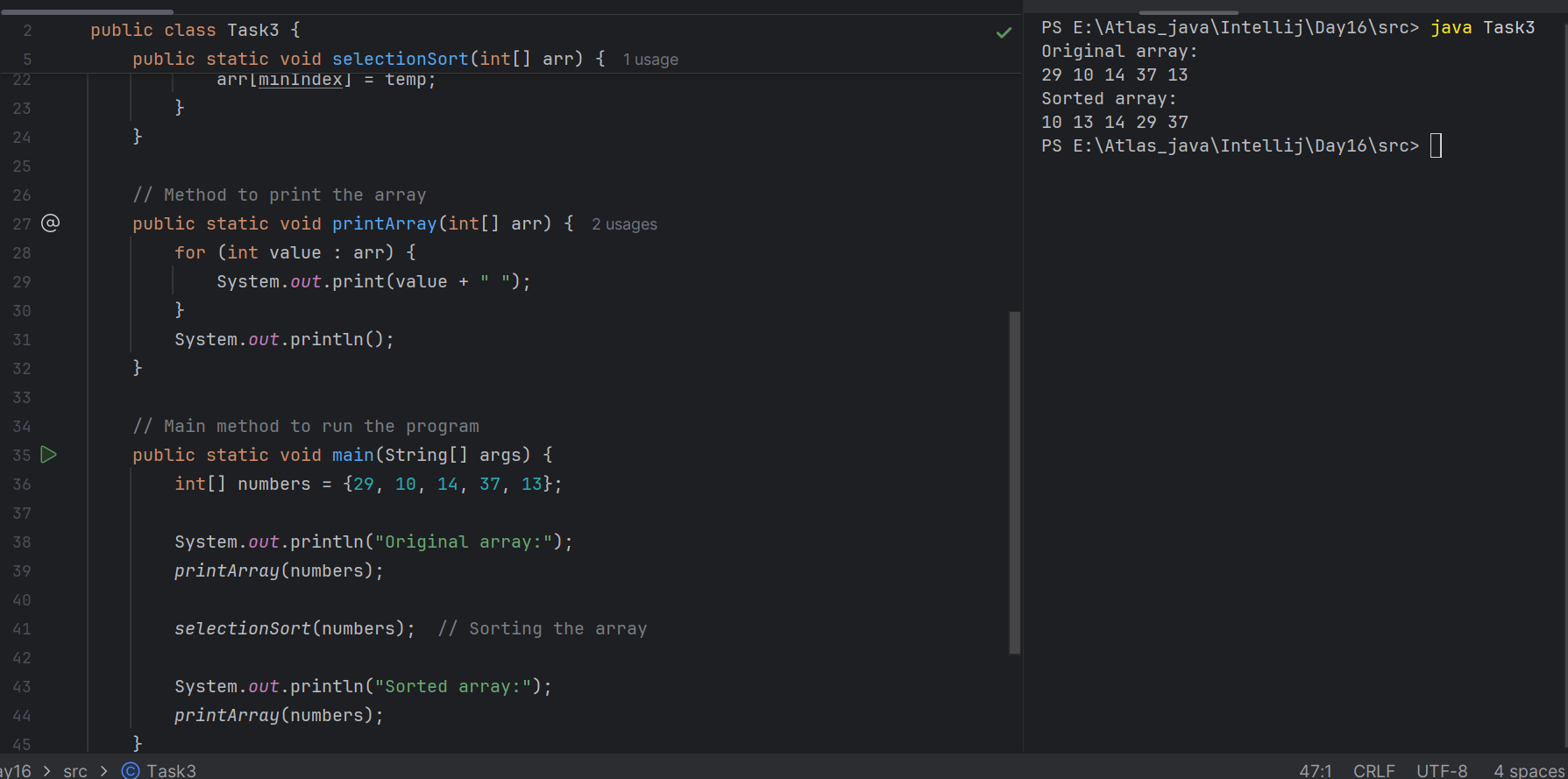
System.*out*.println("Sorted array:");

*printArray*(numbers);

}

}

**OUTPUT:**

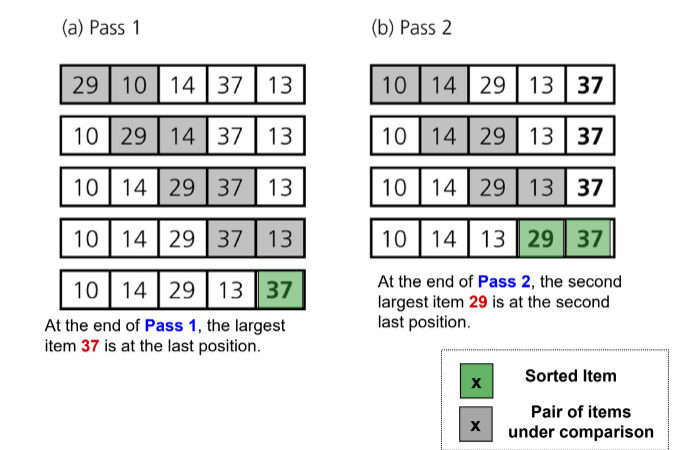


11.18 to 11.23

Task 4:

Write an algorithm / steps for bubble sort

1. Start with the first element of the array.
2. Compare the current element with the next element.
3. If the current element is greater, swap the two.
4. Move to the next pair and repeat the comparison and swap until the end of the array.
5. After one full pass, the largest element will be at the end (like a bubble floating up).
6. Repeat the same process for the remaining unsorted part of the array.
7. Continue this until no more swaps are needed or you’ve completed (n - 1) passes.



Task 5:

Write [psedo code for the bubble sort

function bubbleSort(array):

n = length of array

for i from 0 to n - 1:

for j from 0 to n - i - 2:

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

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

temp = array[j]

array[j] = array[j + 1]

array[j + 1] = temp

Task 6:

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

**PROGRAM:**

public class Task6 {

// Method to perform Bubble Sort

public static void bubbleSort(int[] arr) {

int n = arr.length;

// Outer loop for each pass

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

// Inner loop to compare adjacent elements

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

// Swap if elements are in the wrong order

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

int temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

}

}

}

}

// Method to print the array

public static void printArray(int[] arr) {

for (int value : arr) {

System.*out*.print(value + " ");

}

System.*out*.println();

}

// Main method to test the sorting

public static void main(String[] args) {

int[] numbers = {29, 10, 14, 37, 13};

System.*out*.println("Original array:");

*printArray*(numbers);

*bubbleSort*(numbers);

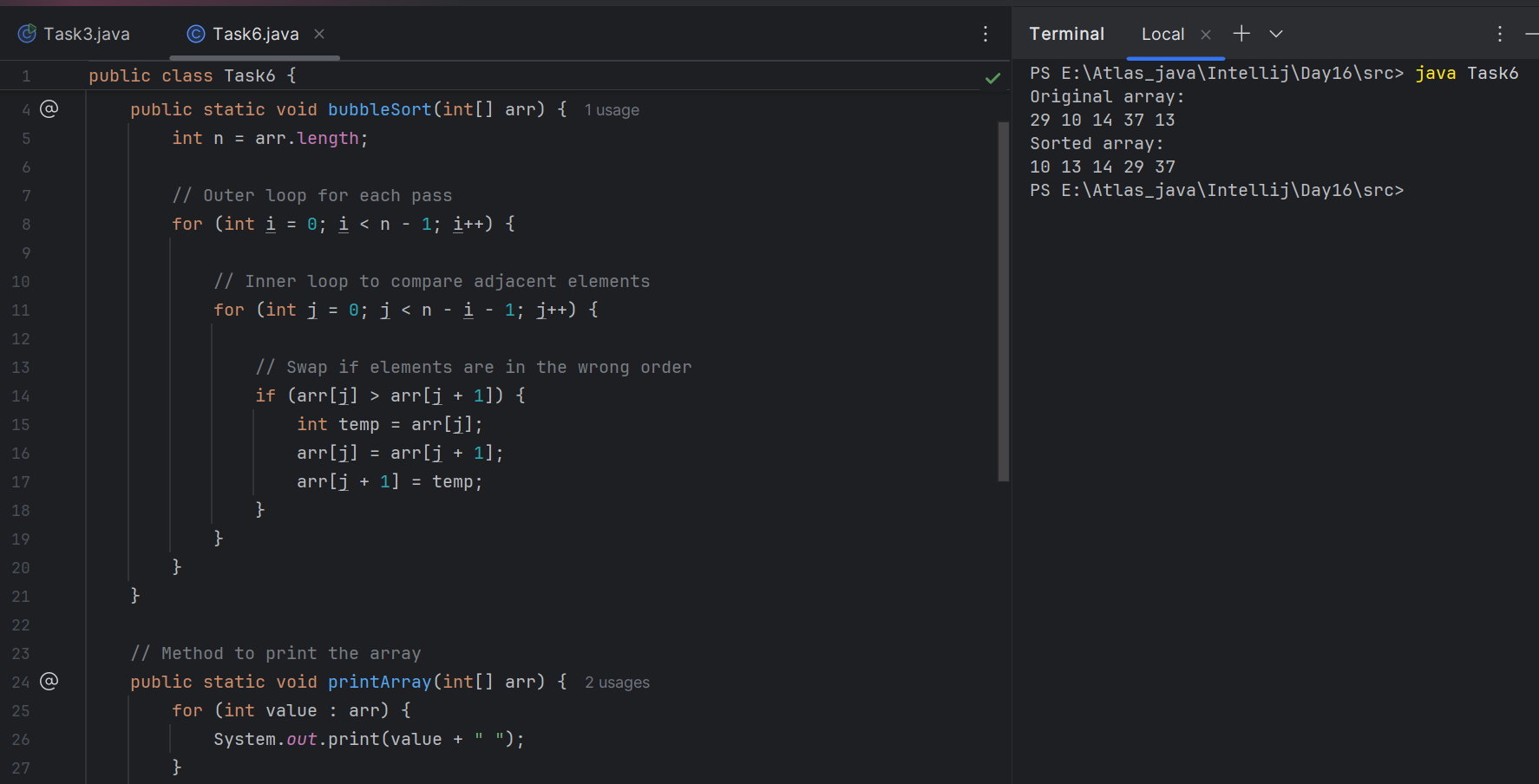
System.*out*.println("Sorted array:");

*printArray*(numbers);

}

}

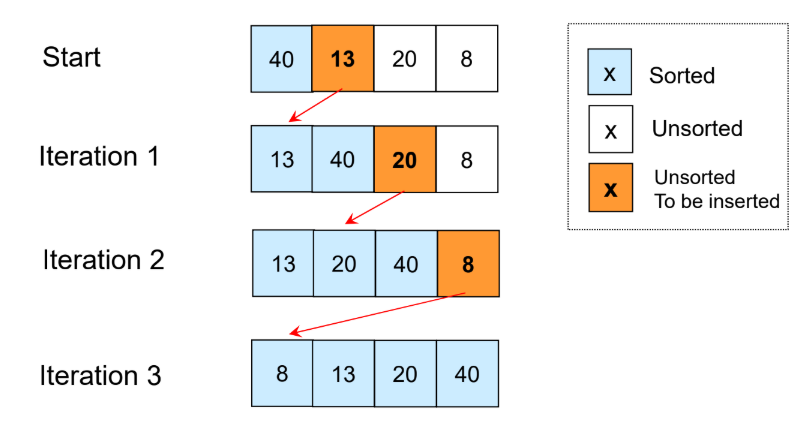
**OUTPUT:**



Task 7:

Write algorithm for the Insertion sort.

1. Start from the second element (index 1), because the first element is already "sorted."
2. Store the current element in a variable called key or next.
3. Compare this element with the elements before it (from right to left).
4. Shift the larger elements one position to the right.
5. Insert the current element (key) into the correct position.
6. Repeat the process for all elements in the array.



Task 8:

Write psedo code for the Insertion sort

function insertionSort(array):

n = length of array

for i from 1 to n - 1:

key = array[i]

j = i - 1

// Shift elements that are greater than key to one position ahead

while j >= 0 and array[j] > key:

array[j + 1] = array[j]

j = j - 1

// Place key at the correct position

array[j + 1] = key

Task 9:

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

**PROGRAM:**

public class Task9 {

// Method to perform Insertion Sort

public static void insertionSort(int[] arr) {

int n = arr.length;

// Start from the second element (index 1)

for (int i = 1; i < n; i++) {

int key = arr[i];

int j = i - 1;

// Move elements that are greater than key to one position ahead

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

arr[j + 1] = arr[j];

j = j - 1;

}

// Place the key at the correct location

arr[j + 1] = key;

}

}

// Method to print the array

public static void printArray(int[] arr) {

for (int num : arr) {

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

}

System.*out*.println();

}

// Main method to test the sort

public static void main(String[] args) {

int[] numbers = {29, 10, 14, 37, 13};

System.*out*.println("Original array:");

*printArray*(numbers);

*insertionSort*(numbers);

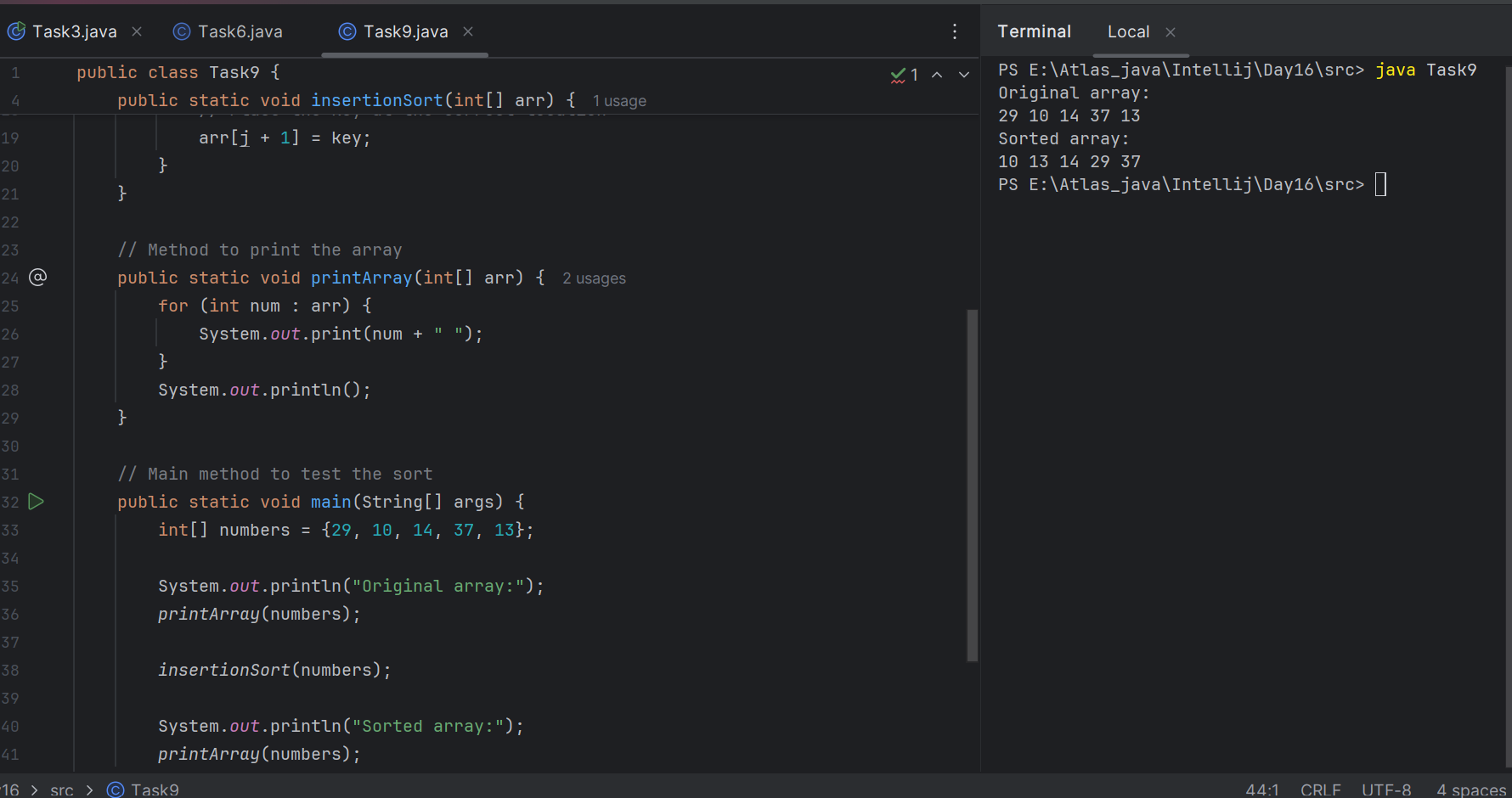
System.*out*.println("Sorted array:");

*printArray*(numbers);

}

}

**OUTPUT:**



Task 10:

What are the advantages and disadvantages of Bubble sort Algo?

List them

## **Advantages of Bubble Sort:**

1. Simple to Understand and Implement  
    Very easy for beginners to learn.
2. No Extra Space Needed  
    It’s an in-place sorting algorithm (does not use extra memory).
3. Stable Sort  
    It preserves the order of equal elements, which can be useful.
4. Good for Small Data Sets  
    Works fine when the list is very small or nearly sorted.

## **Disadvantages of Bubble Sort:**

1. Very Slow for Large Data Sets  
    Time complexity is O(n²) in worst and average cases.
2. Inefficient  
    Performs many unnecessary comparisons and swaps.
3. Not Suitable for Real-Time Use  
    Too slow for practical use with large inputs.
4. Better Algorithms Exist  
    Algorithms like Merge Sort, Quick Sort, or Insertion Sort are more efficient.

TAsk 11:

Write an algorithm for the Insertion sort.

If the list has 0 or 1 element, it is already sorted — return it.

Divide the list into two equal halves:

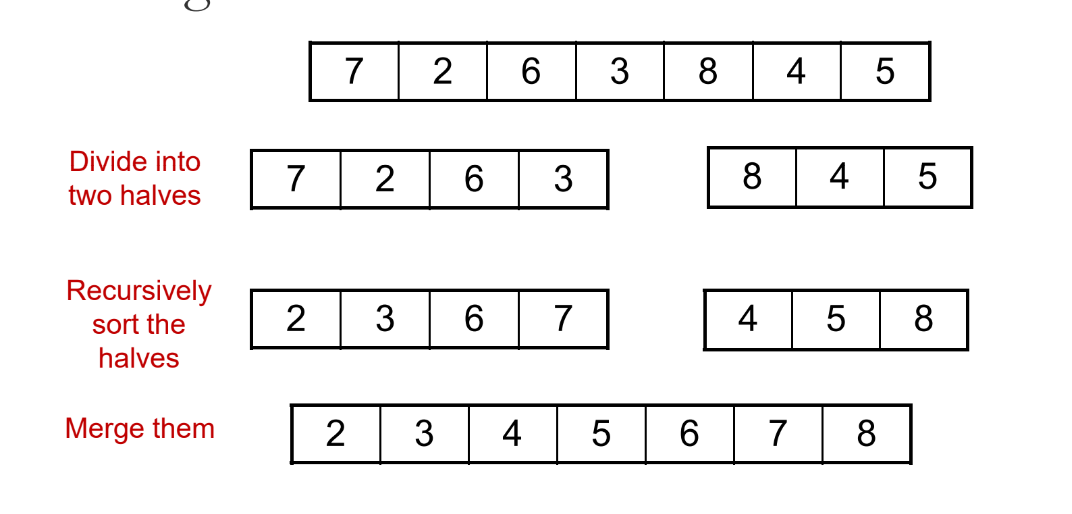
* Left half
* Right half

Recursively apply Merge Sort on:

* Left half
* Right half

Merge the two sorted halves into one sorted list:

* Compare the first element of both halves
* Pick the smaller one and add it to the result
* Repeat until all elements are merged



Task 12:

Class TaskMergeSort:

Method mergeSort(arr, left, right):

If left < right:

mid = (left + right) / 2

Call mergeSort(arr, left, mid)

Call mergeSort(arr, mid + 1, right)

Call merge(arr, left, mid, right)

Method merge(arr, left, mid, right):

Create two temporary arrays: L and R

Copy data from arr into L and R based on mid

Initialize i, j, k to start positions

While both L and R have elements:

If L[i] <= R[j]:

arr[k] = L[i]

i = i + 1

Else:

arr[k] = R[j]

j = j + 1

k = k + 1

Copy any remaining elements from L (if any)

Copy any remaining elements from R (if any)

Task 13:

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

**PROGRAM:**

**public class Task13 {**

**// Merge Sort function**

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

**if (left < right) {**

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

**// Recursively divide the array**

***mergeSort*(arr, left, mid);**

***mergeSort*(arr, mid + 1, right);**

**// Merge the sorted halves**

***merge*(arr, left, mid, right);**

**}**

**}**

**// Merge function**

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

**int n1 = mid - left + 1;**

**int n2 = right - mid;**

**int[] L = new int[n1];**

**int[] R = new int[n2];**

**// Copy data into temp arrays**

**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 the temp arrays back into arr[]**

**int i = 0, j = 0, k = left;**

**while (i < n1 && j < n2) {**

**if (L[i] <= R[j]) {**

**arr[k++] = L[i++];**

**} else {**

**arr[k++] = R[j++];**

**}**

**}**

**// Copy any remaining elements**

**while (i < n1) {**

**arr[k++] = L[i++];**

**}**

**while (j < n2) {**

**arr[k++] = R[j++];**

**}**

**}**

**// Main method to test merge sort**

**public static void main(String[] args) {**

**int[] arr = {7, 2, 6, 3, 8, 4, 5};**

**System.*out*.println("Original array:");**

**for (int val : arr)**

**System.*out*.print(val + " ");**

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

**System.*out*.println("\nSorted array:");**

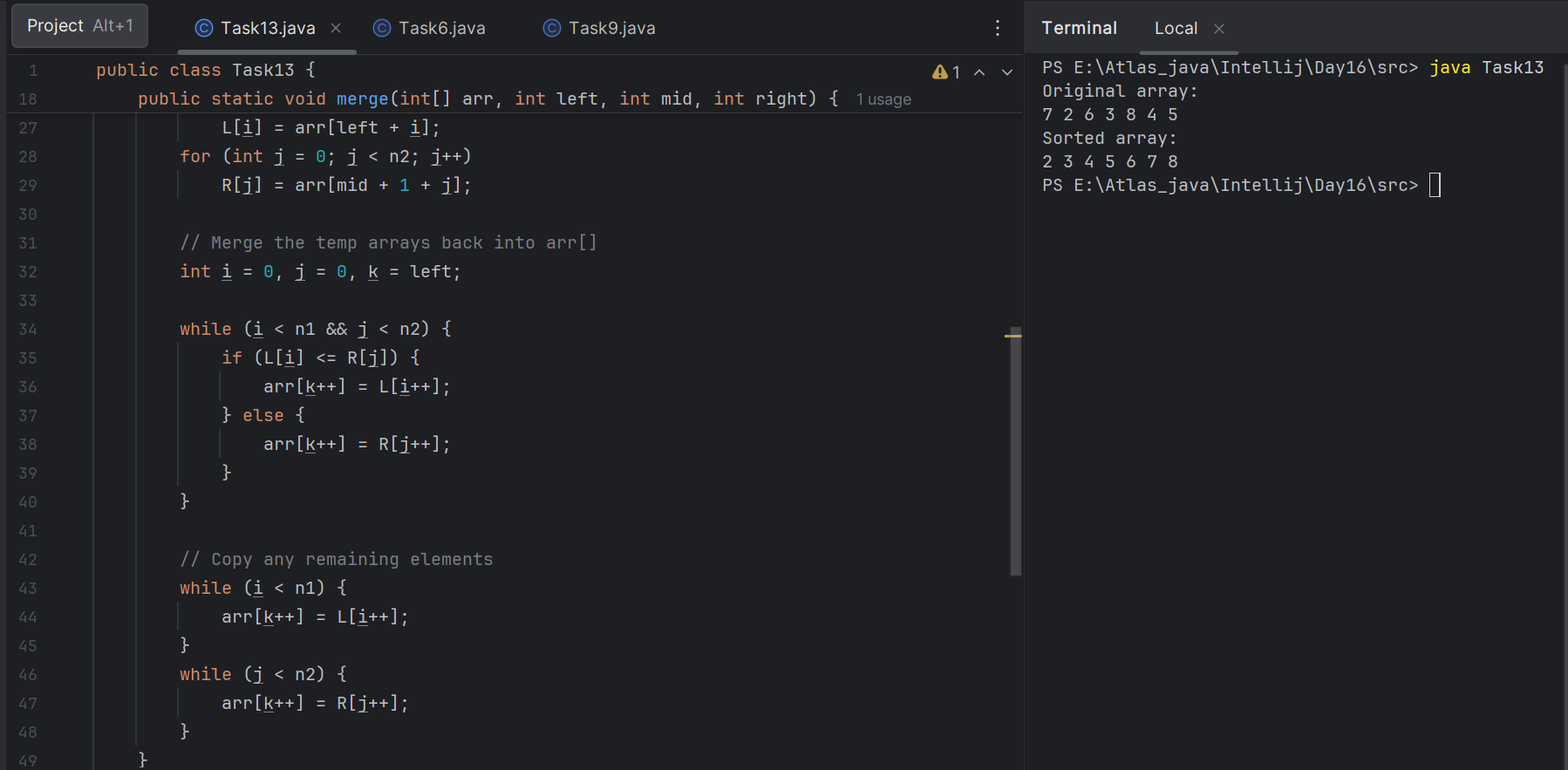
**for (int val : arr)**

**System.*out*.print(val + " ");**

**}**

**}**

**OUTPUT:**



Task 14:

Algo fro quick sort

1. Start with an array of elements you want to sort.
2. If the array has one or zero elements, it is already sorted. Return it.
3. Choose a pivot element from the array.

* You can pick the first, last, middle, or even a random element as the pivot.

1. Partition the remaining elements into two sub-arrays:

* Left sub-array: elements less than the pivot.
* Right sub-array: elements greater than or equal to the pivot.

1. Recursively apply the Quick Sort algorithm to the left and right sub-arrays.
2. Combine the sorted left sub-array, the pivot, and the sorted right sub-array into a single sorted array.
3. End.

Task 15:

Pseudo code for quick sort

class Task14 {

// QuickSort method

quickSort(arr, low, high)

if (low < high)

pivotIndex = partition(arr, low, high)

// Recursively sort left and right parts

quickSort(arr, low, pivotIndex - 1)

quickSort(arr, pivotIndex + 1, high)

// Partition method

partition(arr, low, high)

pivot = arr[high] // Choose the last element as pivot

i = low - 1

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

if (arr[j] < pivot)

i = i + 1

swap arr[i] and arr[j]

// Place pivot in the correct position

swap arr[i + 1] and arr[high]

return i + 1 // Return pivot index

}

Task 16:

Code for Quick sort

**PROGRAM:**

**public class Task16 {**

**// Quick Sort function**

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

**if (low < high) {**

**int pivotIndex = *partition*(arr, low, high);**

***quickSort*(arr, low, pivotIndex - 1); // Sort left part**

***quickSort*(arr, pivotIndex + 1, high); // Sort right part**

**}**

**}**

**// Partition function**

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

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

**int i = low - 1;**

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

**if (arr[j] < pivot) {**

**i++;**

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

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

**arr[i] = arr[j];**

**arr[j] = temp;**

**}**

**}**

**// Place pivot at the correct position**

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

**arr[i + 1] = arr[high];**

**arr[high] = temp;**

**return i + 1; // Return pivot index**

**}**

**// Display function**

**public static void display(int[] arr) {**

**for (int num : arr) {**

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

**}**

**System.*out*.println();**

**}**

**// Main method to test**

**public static void main(String[] args) {**

**int[] arr = { 27, 10, 38, 16, 45, 32 };**

**System.*out*.println("Original Array:");**

***display*(arr);**

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

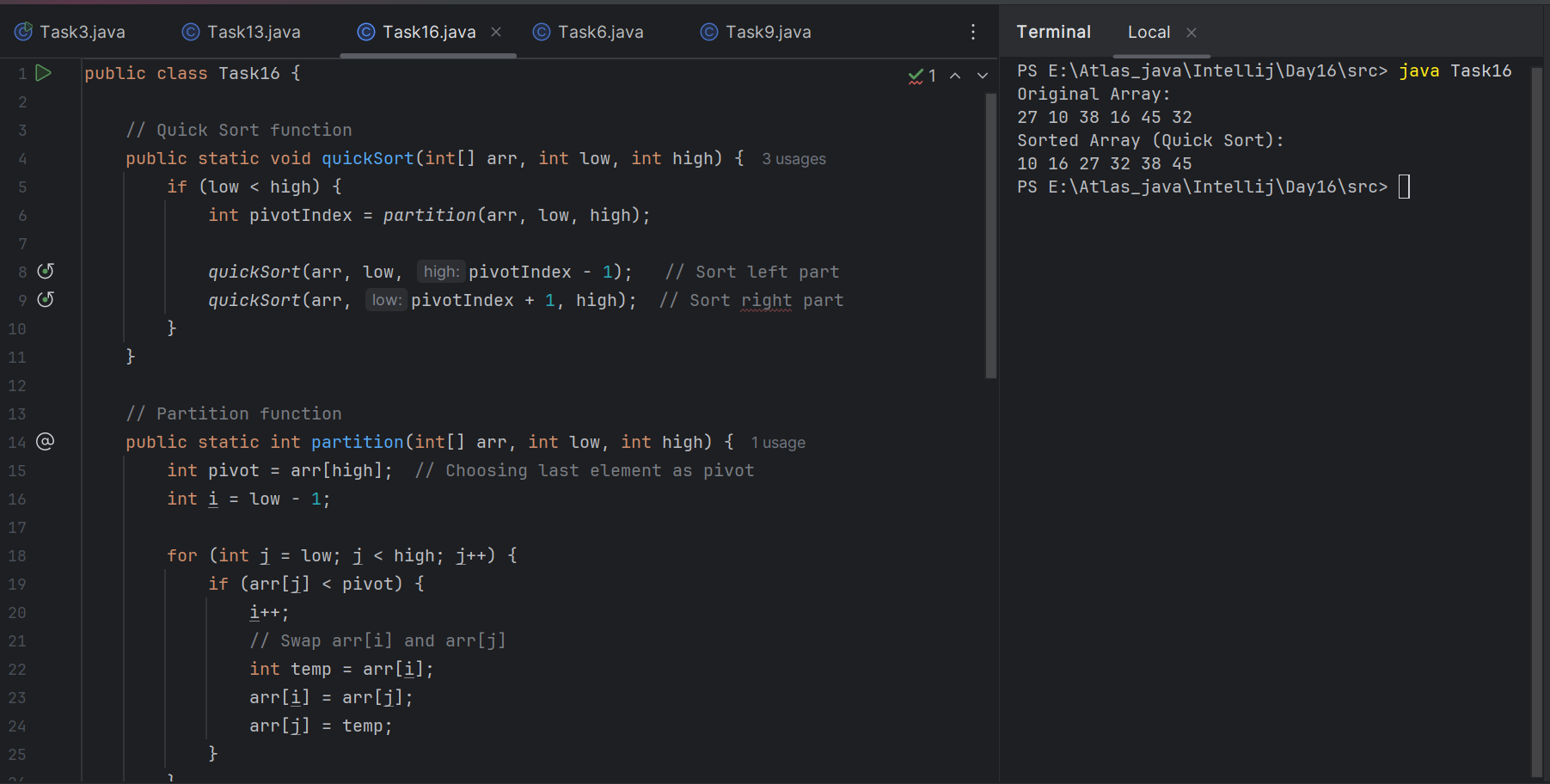
**System.*out*.println("Sorted Array (Quick Sort):");**

***display*(arr);**

**}**

**}**

**OUTPUT:**

****

Task 17:

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

Note:

Merge : O(n log n )

Bubble: O (n square).

Merge Sort is MUCH better than Bubble Sort in terms of time efficiency, especially for larger datasets.

Merge Sort uses a divide-and-conquer method.

* It always runs in O(n log n) time regardless of the input.
* It splits the array and merges them in sorted order efficiently.

Bubble Sort is simple but inefficient.

* In the worst and average case, it compares each element repeatedly, leading to O(n²) time.
* Only slightly faster in the best case (already sorted), but still not ideal.

Task 18:

Leet code qn:

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

https://leetcode.com/problems/merge-sorted-array/description/

**PROGRAM:**

**public class Task18 {**

**public static void mergeArrays(int[] A, int[] B) {**

**int n = A.length;**

**int m = B.length;**

**int[] merged = new int[n + m];**

**int i = 0, j = 0, k = 0;**

**// Merge both arrays by comparing elements**

**while (i < n && j < m) {**

**if (A[i] <= B[j]) {**

**merged[k++] = A[i++];**

**} else {**

**merged[k++] = B[j++];**

**}**

**}**

**// Add remaining elements from A (if any)**

**while (i < n) {**

**merged[k++] = A[i++];**

**}**

**// Add remaining elements from B (if any)**

**while (j < m) {**

**merged[k++] = B[j++];**

**}**

**// Print the merged array**

**System.*out*.print("Merged Array: ");**

**for (int x : merged) {**

**System.*out*.print(x + " ");**

**}**

**}**

**public static void main(String[] args) {**

**int[] A = {1, 3, 5, 7};**

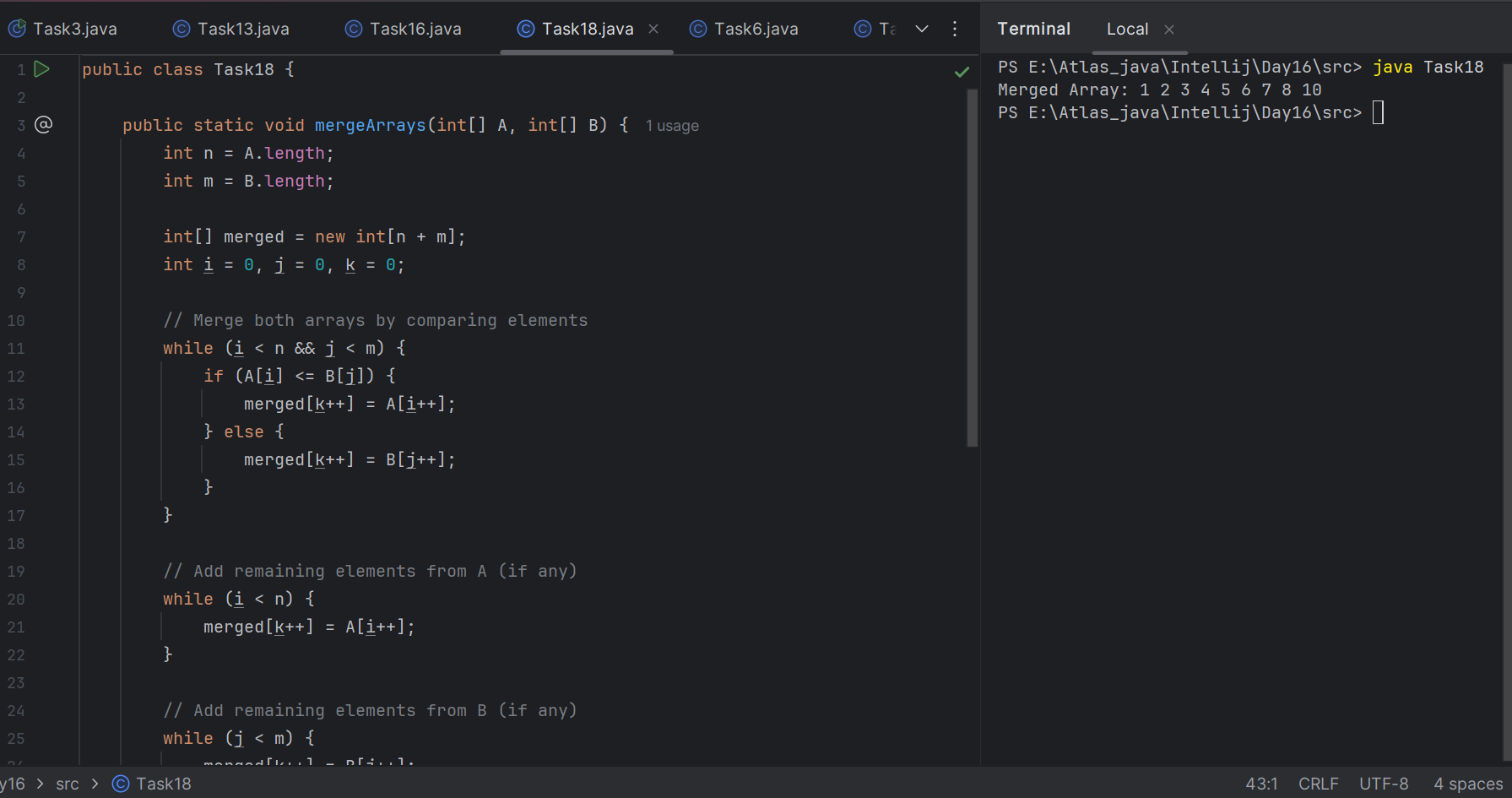
**int[] B = {2, 4, 6, 8, 10};**

***mergeArrays*(A, B);**

**}**

**}**

**OUTPUT:**

****

### **Time Complexity:**

* Total comparisons and insertions = n + m
* Time Complexity = O(n + m)

**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));

}

}

Plz be careful: Because recursive calls consume stack memory for every invocation and excessive depth can exceed system limits also..

**return n + calc(n);**

You're calling calc(n) with the same value of n, which means the function never progresses toward the base case (n == 0).

So what happens

* You call calc(10) → calls calc(10) again → and again → and again...
* It never reduces the value of n.
* This leads to infinite recursion → eventually StackOverflowError.

Task 20: binary search function can we use in unsorted list?

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;

}

}

No, you cannot use binary search on an unsorted list.

Binary Search works by repeatedly dividing the sorted array into halves to locate the target element. It assumes the elements are in a known sorted order (ascending or descending).

If the list is unsorted, the logic of "half the elements can be ignored" no longer holds true, because:

* You have no idea whether the target is on the left or right half.
* This breaks the efficiency and correctness of binary search.

**if (arr[mid] < toFind) {**

**left = mid + 1;**

**}**

assumes the array is sorted in ascending order. If the array is unsorted, this logic fails completely.

Binary search relies on sorted data to eliminate half of the elements in each step.

**int[] arr = {3, 8, 1, 6, 5}; // unsorted**

**int target = 5**;

Binary search might:

* Look at middle = 1
* Assume 5 > 1 → search right half
* But 5 is in left part → wrong logic

================================================================================================================================================

Add ons:

1. **What is the difference between binary tree and binary search tree (bst)**

### **Binary Tree:**

A binary tree is just a way of organizing data where each node has at most two children — one on the left, one on the right.  
 That’s it.  
 There are no rules about how the values are arranged.

You can put any number on the left or right.  
 It’s mainly used for things like expression trees, decision trees, or just storing data in a tree shape.

**Binary Search Tree (BST):**

A Binary Search Tree is a special type of binary tree.

It follows a very strict rule:

* The left child of a node must be smaller than the parent.
* The right child must be larger than the parent.

Because of this rule, BSTs allow us to search quickly — like how you search in a phone book or dictionary.

1. **Can you explain the difference between the structure and operation of Binary tree and BST.**

## **1. Structure**

### Binary Tree

* It is a general tree.
* Each node can have 0, 1, or 2 children.
* There’s no rule for how values are placed.

## **2. Operations**

### Binary Tree

* Insertion: You can insert nodes anywhere (no rule).
* Search: You may have to look at every node (O(n) time).
* Traversal: Inorder, preorder, postorder (same as BST).
* Used for general problems like expression trees or file structures.

A Binary Tree is like a general folder system — place things anywhere.

A BST is like a sorted bookshelf — books (data) are always in order for fast access.

1. **Difference between static and dynamic arrays.. Plz list them**

**Size:**

* **Static Array:** Fixed size (set during creation).
* **Dynamic Array:** Can grow or shrink during runtime.

**Memory Allocation:**

* **Static Array:** Memory is allocated **at compile time**.
* **Dynamic Array:** Memory is allocated **at runtime**.

**Flexibility:**

* **Static Array:** Not flexible — can’t resize.
* **Dynamic Array:** Flexible — can expand or shrink.

**Performance:**

* **Static Array:** Slightly faster (no resizing overhead).
* **Dynamic Array:** Slightly slower (may involve copying during resizing).

**Storage Location:**

* **Static Array:** Stored in **stack** (usually).
* **Dynamic Array:** Stored in **heap** (uses pointers).

**Ease of Use:**

* **Static Array:** Simple and easy to use.
* **Dynamic Array:** Requires more code (especially in low-level languages like C).

**Examples:**

**Static (Java):** int[] arr = new int[5];

* **Dynamic (Java):** ArrayList<Integer> list = new ArrayList<>();

**4.**

**In BFS, DFS which one is more preferred in terms of shortest path for the unweighted graphs.**

**Note:**

**BFS explores nodes in increasing distance order from the source, ensuring shortest paths are found first.**

In unweighted graphs, BFS (Breadth-First Search) is more preferred for finding the shortest path.

### **Why BFS is Better for Shortest Path in Unweighted Graphs:**

* BFS explores all nodes at current depth (distance) before going deeper.
* It guarantees that the first time it visits a node, it's via the shortest possible path from the source.
* BFS uses a queue, so it processes nodes in the order they were discovered, layer by layer.

**Why DFS is Not Ideal:**

* DFS explores as far as possible along a branch before backtracking.
* It may miss the shortest path initially because it does not guarantee minimal distance.
* DFS is better suited for:  
  + Detecting cycles
  + Traversing entire trees
  + Solving puzzles like mazes or backtracking problems

For shortest path in unweighted graphs,

BFS is preferred because it guarantees the shortest path.

5.

Write a code to implement a stack using an array.

Note: plz ensure bounds are checked to avoid overflow/underflow

**PROGRAM:**

**public class AddonTask05 {**

**private int maxSize;**

**private int[] stack;**

**private int top;**

**// Constructor to initialize stack**

**public AddonTask05(int size) {**

**maxSize = size;**

**stack = new int[maxSize];**

**top = -1;**

**}**

**// Push operation**

**public void push(int value) {**

**if (top == maxSize - 1) {**

**System.*out*.println("Stack Overflow! Cannot push " + value);**

**} else {**

**stack[++top] = value;**

**System.*out*.println(value + " pushed to stack.");**

**}**

**}**

**// Pop operation**

**public int pop() {**

**if (isEmpty()) {**

**System.*out*.println("Stack Underflow! Cannot pop.");**

**return -1;**

**} else {**

**return stack[top--];**

**}**

**}**

**// Peek operation**

**public int peek() {**

**if (isEmpty()) {**

**System.*out*.println("Stack is empty.");**

**return -1;**

**} else {**

**return stack[top];**

**}**

**}**

**// Check if stack is empty**

**public boolean isEmpty() {**

**return top == -1;**

**}**

**// Display elements of stack**

**public void display() {**

**if (isEmpty()) {**

**System.*out*.println("Stack is empty.");**

**} else {**

**System.*out*.print("Stack elements: ");**

**for (int i = top; i >= 0; i--) {**

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

**}**

**System.*out*.println();**

**}**

**}**

**// Main method to test stack**

**public static void main(String[] args) {**

**AddonTask05 stack = new AddonTask05(5);**

**stack.push(10);**

**stack.push(20);**

**stack.push(30);**

**stack.display();**

**System.*out*.println("Top element: " + stack.peek());**

**System.*out*.println("Popped: " + stack.pop());**

**stack.display();**

**}**

**}**

**OUTPUT:**

