Day 16 - 11th July 2025

Selection Sort:

Task 1:

Write an algorithm / steps for selection sort.

Step 1: Set minIndex to position 0

(minIndex will hold the index of the smallest number in the unsorted subarray)

Step 2: Search for the smallest element in the unsorted subarray and update minIndex

Step 3: Swap the element at the position minIndex with the first element of the unsorted subarray.

Step 4: Again set minIndex to the first position of the unsorted subarray

Step 5: Repeat steps 2 to 4 until the array gets sorted

Task 2:

Write a pseudo code for the selection sort

ALGORITHM SelectionSort(A: array of size n)

FOR i = 0 TO n-1

// Find minimum element in unsorted portion

min\_index = i

FOR j = i+1 TO n-1

IF A[j] < A[min\_index] THEN

min\_index = j

END IF

END FOR

// Swap found minimum element with first element of unsorted part

IF min\_index != i THEN

temp = A[i]

A[i] = A[min\_index]

A[min\_index] = temp

END IF

END FOR

END ALGORITHM

// Optional helper function for swapping

PROCEDURE Swap(x, y)

temp = x

x = y

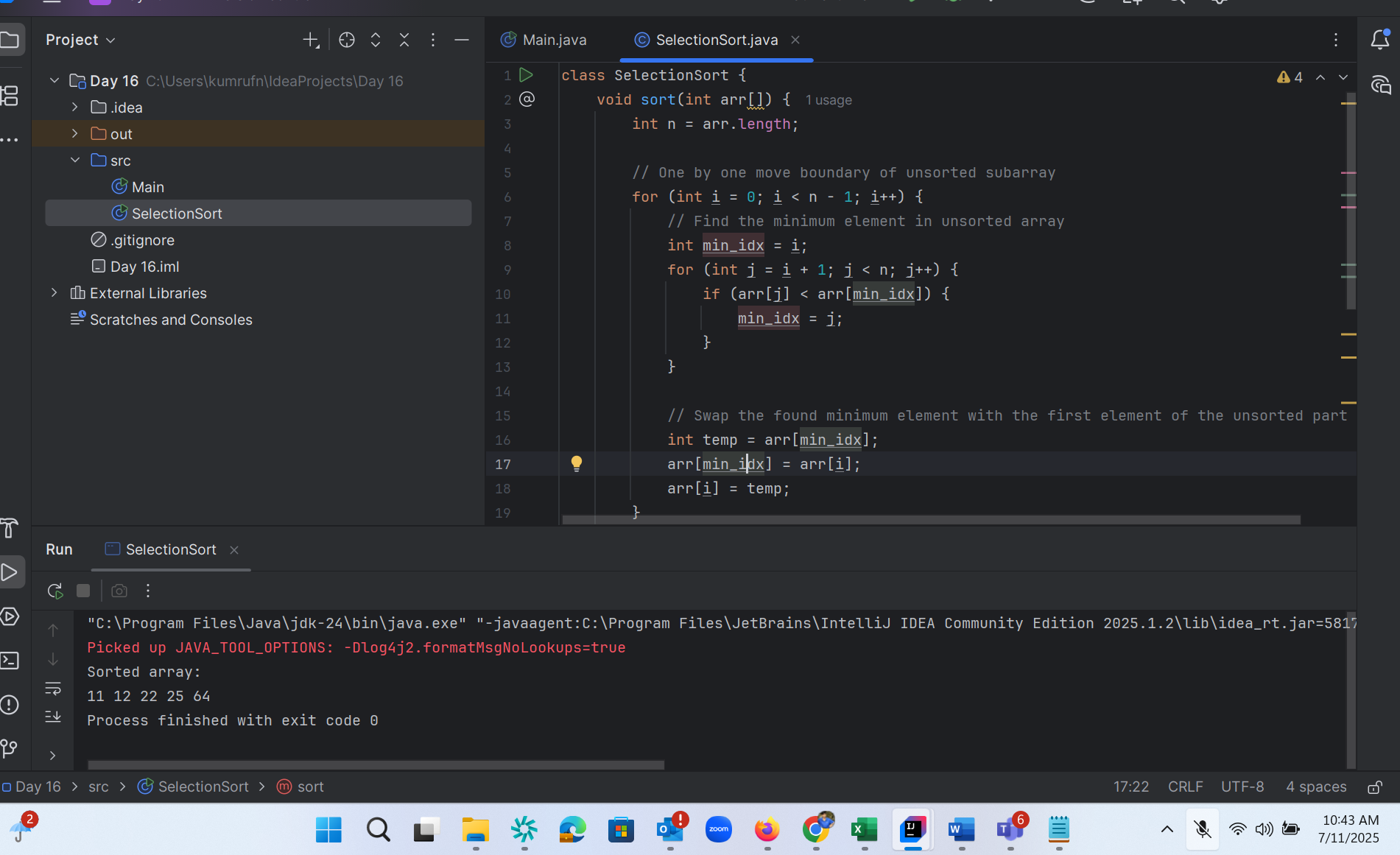
y = temp

END PROCEDURE

Task 3 :

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

class SelectionSort {  
 void sort(int arr[]) {  
 int n = arr.length;  
  
 // One by one move boundary of unsorted subarray  
 for (int i = 0; i < n - 1; i++) {  
 // Find the minimum element in unsorted array  
 int min\_idx = i;  
 for (int j = i + 1; j < n; j++) {  
 if (arr[j] < arr[min\_idx]) {  
 min\_idx = j;  
 }  
 }  
  
 // Swap the found minimum element with the first element of the unsorted part  
 int temp = arr[min\_idx];  
 arr[min\_idx] = arr[i];  
 arr[i] = temp;  
 }  
 }  
  
 public static void main(String args[]) {  
 SelectionSort ob = new SelectionSort();  
 int arr[] = {64, 25, 12, 22, 11};  
 ob.sort(arr);  
  
 System.*out*.println("Sorted array:");  
 for (int i = 0; i < arr.length; ++i) {  
 System.*out*.print(arr[i] + " ");  
 }  
 }  
}



Task 4 Write an algorithm / steps for bubble sort.

Start at the beginning of the list.

Compare each pair of adjacent elements.

If the elements are in the wrong order, swap them.

Move to the next pair and repeat step 3 until the end of the list.

After each pass through the list, the largest element moves to its correct position.

Task 5:

Bubble Sort pseudocode

data\_set = [9,2,5,23,34,56]

last\_exam\_position = data\_set.length - 2.

swap = true.

WHILE swap == true.

swap = false.

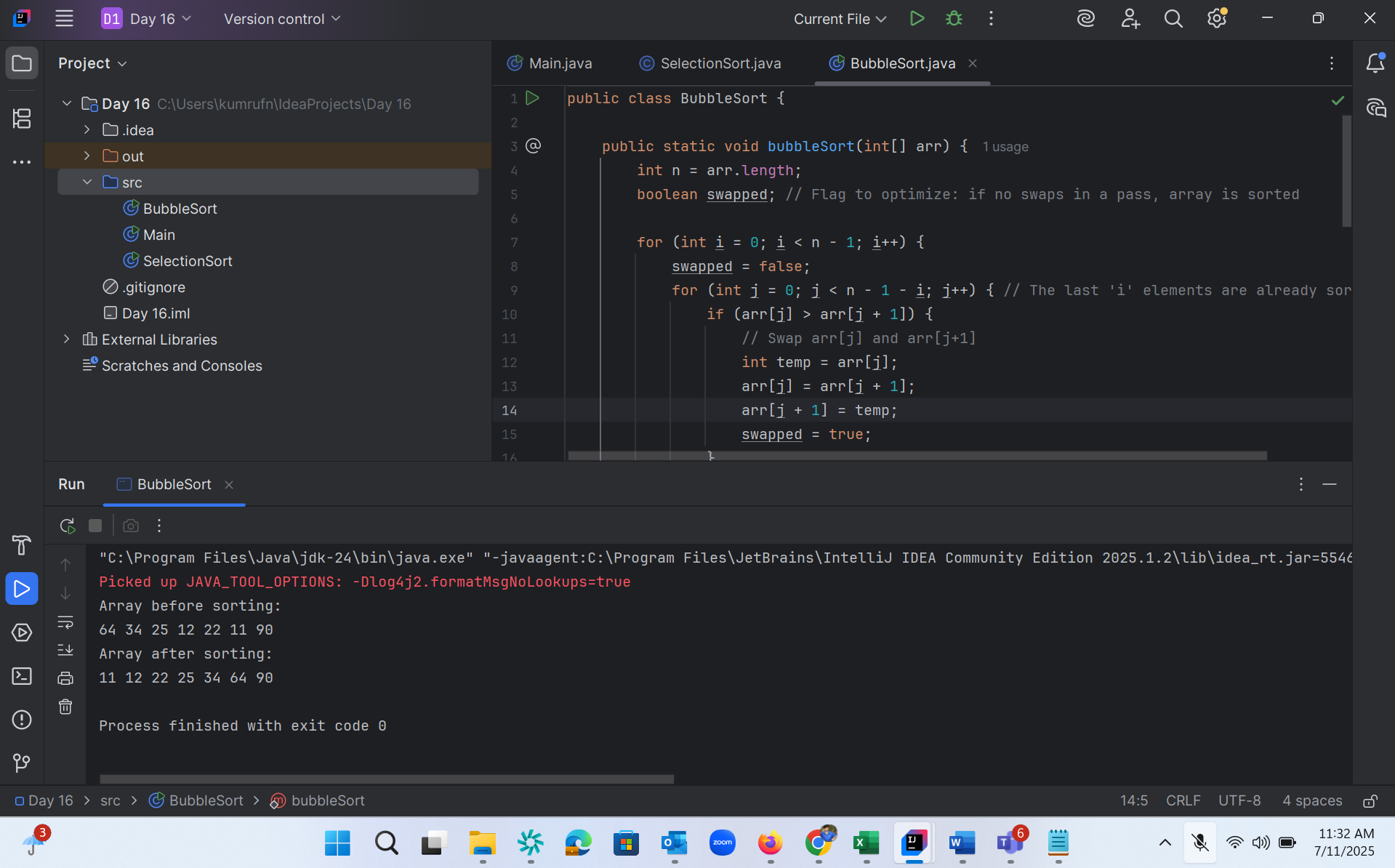
FOR i = 0 to last\_exam\_position. Remember that lists start at the 0th position, not the 1st.

IF data\_set[i] > data\_set[i +1] THEN.

temp = data\_set[i+1]

Task 6:

public class BubbleSort {  
  
 public static void bubbleSort(int[] arr) {  
 int n = arr.length;  
 boolean swapped; // Flag to optimize: if no swaps in a pass, array is sorted  
  
 for (int i = 0; i < n - 1; i++) {  
 swapped = false;  
 for (int j = 0; j < n - 1 - i; j++) { // The last 'i' elements are already sorted  
 if (arr[j] > arr[j + 1]) {  
 // Swap arr[j] and arr[j+1]  
 int temp = arr[j];  
 arr[j] = arr[j + 1];  
 arr[j + 1] = temp;  
 swapped = true;  
 }  
 }  
 // If no two elements were swapped by inner loop, then break  
 if (!swapped) {  
 break;  
 }  
 }  
 }  
  
 public static void main(String[] args) {  
 int[] data = {64, 34, 25, 12, 22, 11, 90};  
 System.*out*.println("Array before sorting:");  
 for (int i : data) {  
 System.*out*.print(i + " ");  
 }  
 System.*out*.println();  
  
 *bubbleSort*(data);  
  
 System.*out*.println("Array after sorting:");  
 for (int i : data) {  
 System.*out*.print(i + " ");  
 }  
 System.*out*.println();  
 }  
}



Task 7 :

Write an algorithm / steps for insertion sort?

1. Start with the second element (index 1)

2. Compare it with previous elements

3. If previous element is greater, shift it right

4. Place current element in its correct position

5. Repeat for all elements in array

Task 8 :

Write pseudo code for the insertion sort?

ALGORITHM InsertionSort(array)

n = length of array

FOR i = 1 to n-1

// Store the current element to be inserted

current = array[i]

// Initialize position for comparison

j = i - 1

// Move elements greater than current

// to one position ahead of their current position

WHILE j >= 0 AND array[j] > current

array[j + 1] = array[j]

j = j - 1

END WHILE

// Place current element in its correct position

array[j + 1] = current

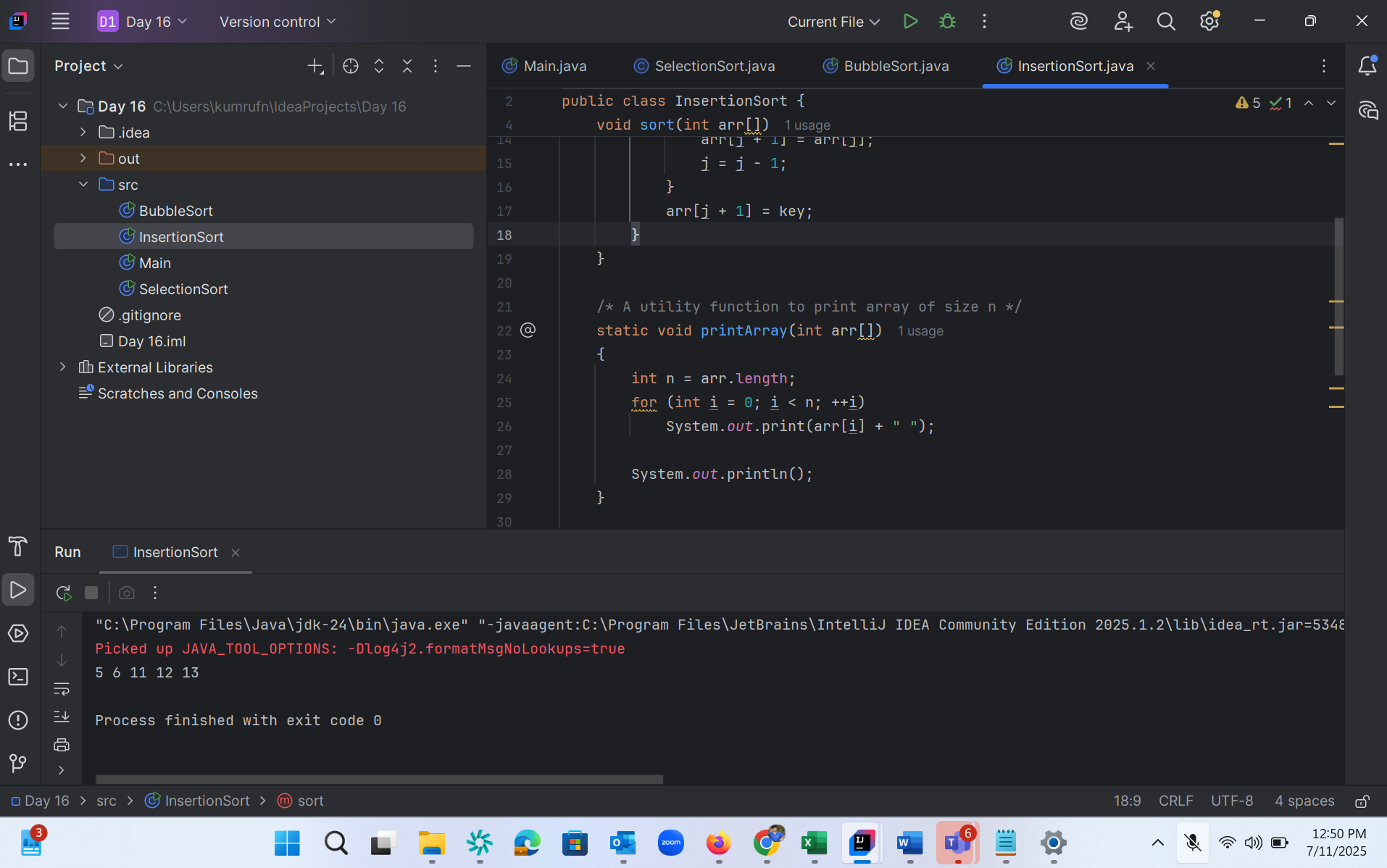
END FOR

END ALGORITHM

Task 9 :

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

// Java program for implementation of Insertion Sort  
public class InsertionSort {  
 /\* Function to sort array using insertion sort \*/  
 void sort(int arr[])  
 {  
 int n = arr.length;  
 for (int i = 1; i < n; ++i) {  
 int key = arr[i];  
 int j = i - 1;  
 /\* Move elements of arr[0..i-1], that are  
 greater than key, to one position ahead  
 of their current position \*/  
 while (j >= 0 && arr[j] > key) {  
 arr[j + 1] = arr[j];  
 j = j - 1;  
 }  
 arr[j + 1] = key;  
 }  
 }  
  
 /\* A utility function to print array of size n \*/  
 static void printArray(int arr[])  
 {  
 int n = arr.length;  
 for (int i = 0; i < n; ++i)  
 System.*out*.print(arr[i] + " ");  
  
 System.*out*.println();  
 }  
  
 // Driver method  
 public static void main(String args[])  
 {  
 int arr[] = { 12, 11, 13, 5, 6 };  
  
 InsertionSort ob = new InsertionSort();  
 ob.sort(arr);  
  
 *printArray*(arr);  
 }  
}



Task 10 :

What are the advantages and disadvantages of Bubble sort Algo?

List them

note:

Poor performance - limitations of bubble sort

Here's a list of advantages and disadvantages of the Bubble Sort algorithm:

Advantages:

1. Simplicity: Easy to understand and implement

2. In-place sorting: Requires minimal additional memory

3. Stable sort: Maintains the relative order of equal elements

4. Adaptive: Performance improves for partially sorted arrays

5. Works well with small datasets

6. Easy to detect if the list is already sorted

Disadvantages:

1. Poor performance: O(n²) time complexity for average and worst cases

2. Inefficient for large datasets

3. Excessive number of comparisons and swaps

4. Slow compared to more advanced sorting algorithms like Quick Sort, Merge Sort, or Heap Sort

5. Does not take advantage of existing order in the input

6. Lack of adaptability to different data types or structures

7. Performance degrades quickly as the number of elements increases

8. Not suitable for systems where time is a critical factor

9. Inefficient use of CPU cache due to many array accesses

10. No improvement in best-case scenario (already sorted array) without additional checks

Limitations of Bubble Sort:

1. Scalability issues: Performance becomes unacceptable for large datasets

2. Time-consuming for reversely sorted arrays

3. Not practical for most real-world applications due to inefficiency

4. Requires passing through the entire list multiple times, even if sorted early

5. Cannot handle complex data structures efficiently without significant modifications

6. Not parallelizable, limiting its use in multi-core processor environments

7. Excessive element movements can lead to increased power consumption in embedded systems

8. Not suitable for external sorting of large files that don't fit in memory

In summary, while Bubble Sort is simple and can be useful for educational purposes or very small datasets, its poor performance and limitations make it impractical for most real-world sorting tasks, especially when dealing with large amounts of data.

Task 12 :

Algo for merge sort,

Algorithm Steps:

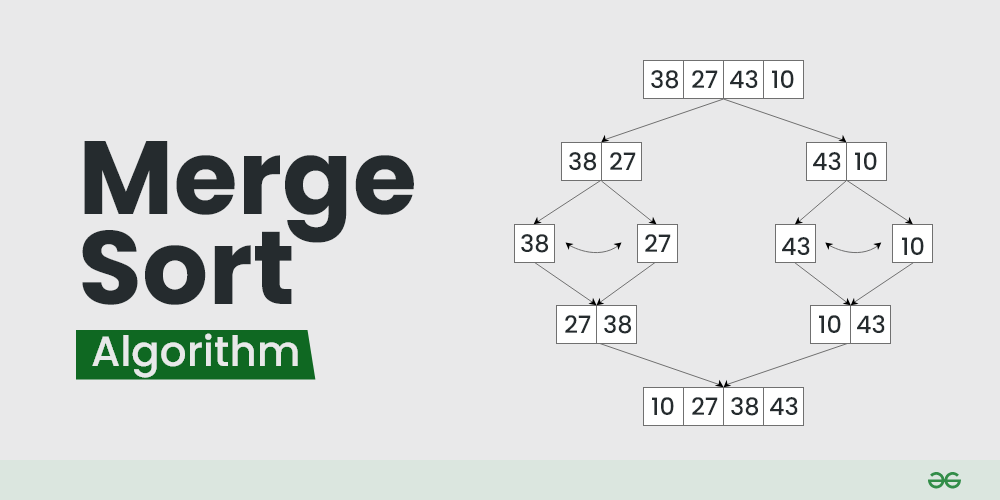
Divide Phase:

* + Find middle point of array
  + Divide array into two halves
  + Recursively sort left half
  + Recursively sort right half

Merge Phase:

* + Create temporary arrays for left and right parts
  + Compare elements from both arrays
  + Merge them back in sorted order

**Task 13**

****

**pseudo code for merge sort,**

ALGORITHM MergeSort(arr, left, right)

    IF left < right

        mid = (left + right) / 2

        MergeSort(arr, left, mid)

        MergeSort(arr, mid + 1, right)

        Merge(arr, left, mid, right)

    END IF

END ALGORITHM

ALGORITHM Merge(arr, left, mid, right)

    n1 = mid - left + 1

    n2 = right - mid

    // Create temporary arrays

    Let L[1...n1] and R[1...n2] be new arrays

    // Copy data to temporary arrays L[] and R[]

    FOR i = 1 to n1

        L[i] = arr[left + i - 1]

    END FOR

    FOR j = 1 to n2

        R[j] = arr[mid + j]

    END FOR

    // Merge the temporary arrays back into arr[left..right]

    i = 1

    j = 1

    k = left

    WHILE i ≤ n1 AND j ≤ n2

        IF L[i] ≤ R[j]

            arr[k] = L[i]

            i = i + 1

        ELSE

            arr[k] = R[j]

            j = j + 1

        END IF

        k = k + 1

    END WHILE

    // Copy the remaining elements of L[], if any

    WHILE i ≤ n1

        arr[k] = L[i]

        i = i + 1

        k = k + 1

    END WHILE

    // Copy the remaining elements of R[], if any

    WHILE j ≤ n2

        arr[k] = R[j]

        j = j + 1

        k = k + 1

    END WHILE

END ALGORITHM

// To sort an entire array, we would call:

// MergeSort(arr, 0, arr.length - 1)

**TASK 14**

**code for Merge sort**

**public class MergeSort {**

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

**if (low < high) {**

**int mid = low + (high - low) / 2;**

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

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

**merge(arr, low, mid, high);**

**}**

**}**

**private void merge(int[] arr, int low, int mid, int high) {**

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

**int n2 = high - mid;**

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

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

**for (int i = 0; i < n1; i++) {**

**leftArray[i] = arr[low + i];**

**}**

**for (int j = 0; j < n2; j++) {**

**rightArray[j] = arr[mid + 1 + j];**

**}**

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

**int k = low;**

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

**if (leftArray[i] <= rightArray[j]) {**

**arr[k] = leftArray[i];**

**i++;**

**} else {**

**arr[k] = rightArray[j];**

**j++;**

**}**

**k++;**

**}**

**while (i < n1) {**

**arr[k] = leftArray[i];**

**i++;**

**k++;**

**}**

**while (j < n2) {**

**arr[k] = rightArray[j];**

**j++;**

**k++;**

**}**

**}**

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

**int[] data = {12, 11, 13, 5, 6, 7};**

**MergeSort ms = new MergeSort();**

**ms.mergeSort(data, 0, data.length - 1);**

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

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

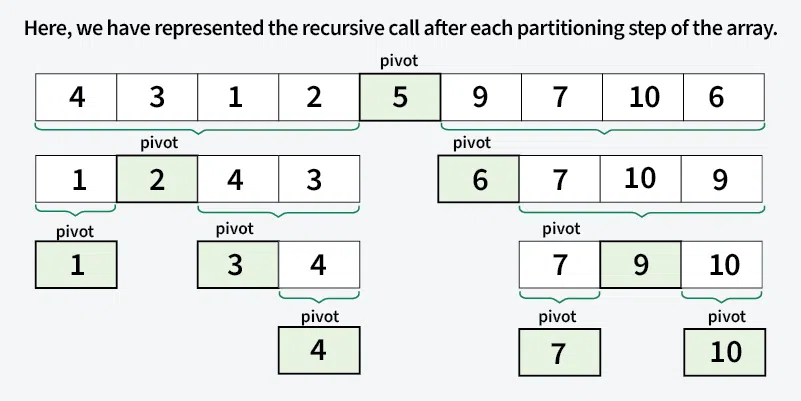
**System.*out*.print(num + " ");**

**}**

**}**

**}**

**Task 15:**

****

**Algo fro quick sort**

Steps of the Algorithm:

**pseudo code for merge sort,**

ALGORITHM MergeSort(arr, left, right)

    IF left < right

        mid = (left + right) / 2

        MergeSort(arr, left, mid)

        MergeSort(arr, mid + 1, right)

        Merge(arr, left, mid, right)

    END IF

END ALGORITHM

ALGORITHM Merge(arr, left, mid, right)

    n1 = mid - left + 1

    n2 = right - mid

    // Create temporary arrays

    Let L[1...n1] and R[1...n2] be new arrays

    // Copy data to temporary arrays L[] and R[]

    FOR i = 1 to n1

        L[i] = arr[left + i - 1]

    END FOR

    FOR j = 1 to n2

        R[j] = arr[mid + j]

    END FOR

    // Merge the temporary arrays back into arr[left..right]

    i = 1

    j = 1

    k = left

    WHILE i ≤ n1 AND j ≤ n2

        IF L[i] ≤ R[j]

            arr[k] = L[i]

            i = i + 1

        ELSE

            arr[k] = R[j]

            j = j + 1

        END IF

        k = k + 1

    END WHILE

    // Copy the remaining elements of L[], if any

    WHILE i ≤ n1

        arr[k] = L[i]

        i = i + 1

        k = k + 1

    END WHILE

    // Copy the remaining elements of R[], if any

    WHILE j ≤ n2

        arr[k] = R[j]

        j = j + 1

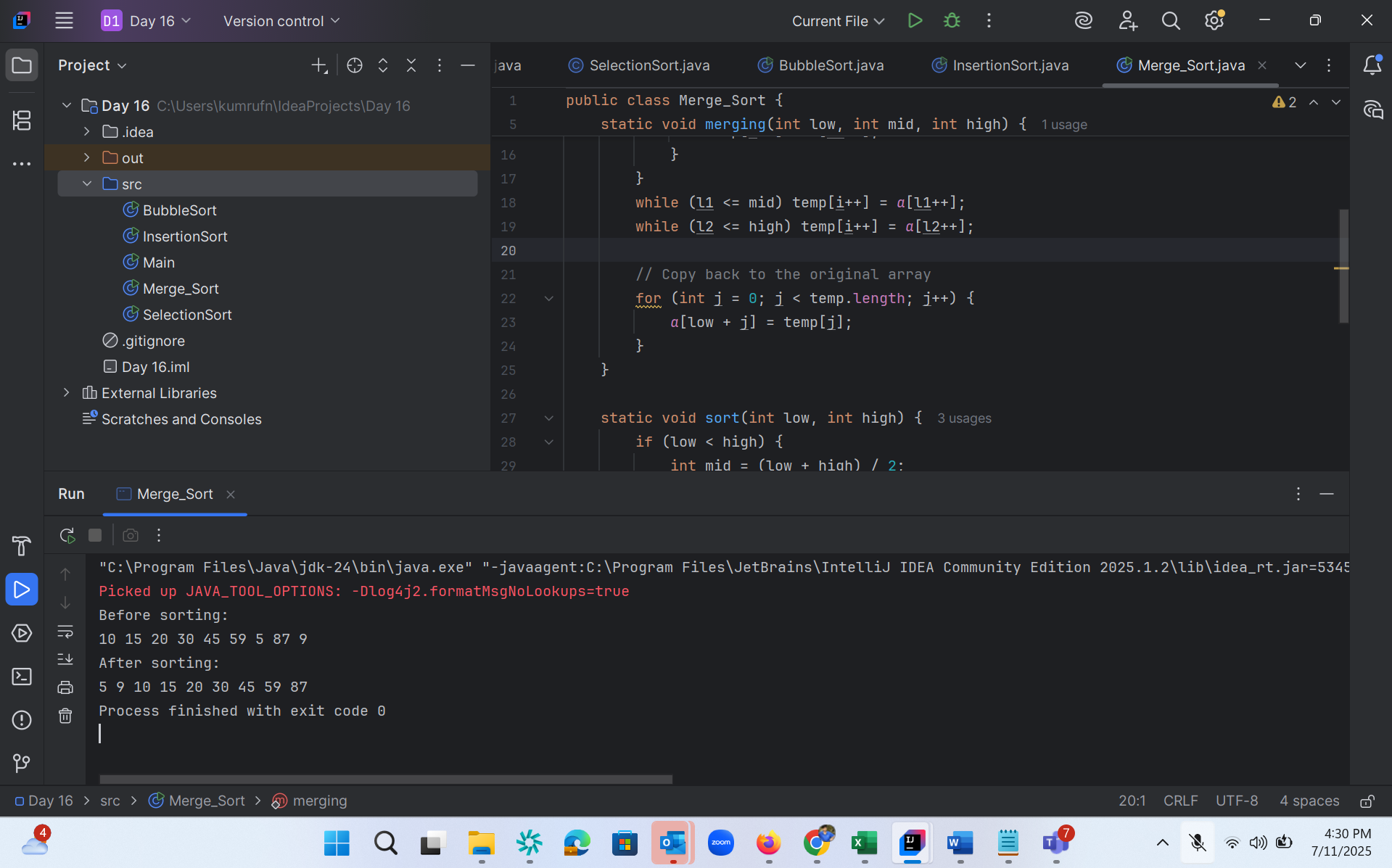
        k = k + 1

    END WHILE

END ALGORITHM

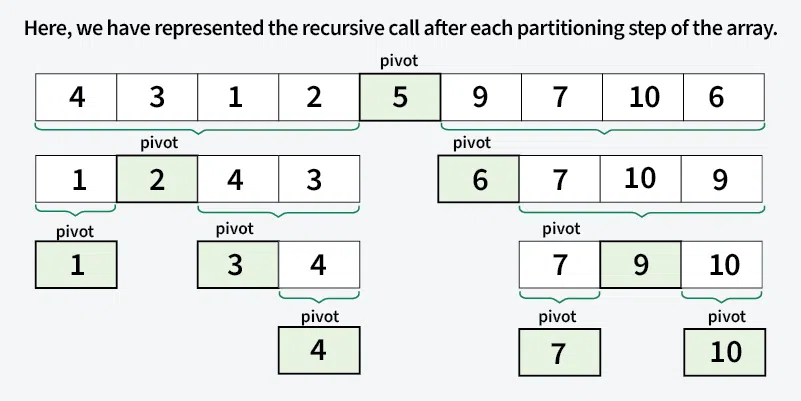
// To sort an entire array, we would call:

// MergeSort(arr, 0, arr.length - 1)



Task 15 :

**Task 15:**

****

**Algo fro quick sort**

Steps of the Algorithm:

1. Choose a 'pivot' element from the array
2. Partition the array:
   * Put elements smaller than pivot to its left
   * Put elements larger than pivot to its right
3. Recursively repeat steps 1-2 for subarrays on left and right of pivot

**Task 16:**

**Pseudo code for quick sort**

function quickSort(arr, low, high)

    if low < high then

        pivotIndex = partition(arr, low, high)

        quickSort(arr, low, pivotIndex - 1)

        quickSort(arr, pivotIndex + 1, high)

    end if

end function

function partition(arr, low, high)

    pivot = arr[high]

    i = low - 1

    for j from low to high - 1 do

        if arr[j] <= pivot then

            i = i + 1

            swap arr[i] and arr[j]

        end if

    end for

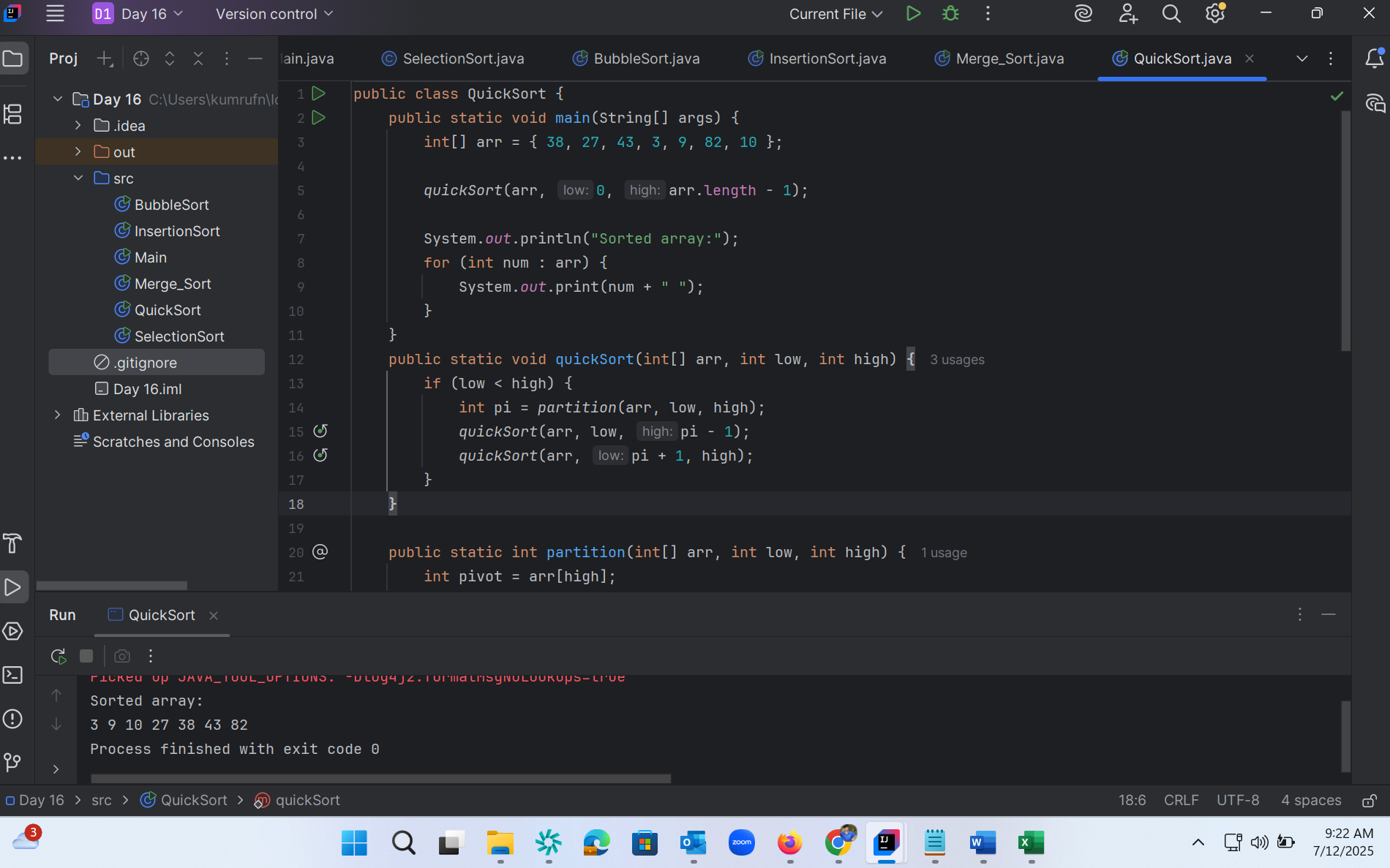
    swap arr[i + 1] and arr[high]

    return i + 1

end function

task 17 :

public class QuickSort {  
 public static void main(String[] args) {  
 int[] arr = { 38, 27, 43, 3, 9, 82, 10 };  
  
 *quickSort*(arr, 0, arr.length - 1);  
  
 System.*out*.println("Sorted array:");  
 for (int num : arr) {  
 System.*out*.print(num + " ");  
 }  
 }  
 public static void quickSort(int[] arr, int low, int high) {  
 if (low < high) {  
 int pi = *partition*(arr, low, high);  
 *quickSort*(arr, low, pi - 1);  
 *quickSort*(arr, pi + 1, high);  
 }  
 }  
  
 public static int partition(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;  
 }  
}



**Add ons**

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

**Binary Tree:**

1. A tree data structure where each node has at most two children (left and right)
2. No specific ordering of nodes is required
3. Left and right child nodes can have any values
4. Tree can be unbalanced or balanced
5. Used for: hierarchical data representation, expression trees, etc.

**Binary Search Tree (BST):**

1. A specialized binary tree that follows specific ordering rules
2. For any node N:
   * All nodes in left subtree must be less than N
   * All nodes in right subtree must be greater than N
3. No duplicate values are typically allowed (though some implementations may handle duplicates)
4. Enables efficient searching, insertion, and deletion operations
5. Used for: searching, sorting, maintaining ordered data

**2. In sorted array why do you think binary search tree is best than linear search.. Can you ecplain plz**

Binary Search Tree (BST) is better than linear search for sorted arrays because of its efficiency and time complexity. Here's a detailed explanation:

1. Time Complexity Comparison:

* Linear Search: O(n) - must check every element
* Binary Search: O(log n) - eliminates half the elements with each comparison
* Example with 16 elements:

   Array: [1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31]

Looking for 23:

Linear Search:

- Checks: 1→3→5→7→9→11→13→15→17→19→21→23

- Takes 12 comparisons

Binary Search:

1. Check middle (15)

   - 23 > 15, eliminate left half

   [17, 19, 21, 23, 25, 27, 29, 31]

2. Check middle (23)

   - Found in just 2 comparisons!

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

Static Arrays:

1. Fixed size defined at declaration time
2. Memory allocated at compile time
3. Cannot grow or shrink during runtime
4. Faster access time due to continuous memory allocation
5. Memory allocated on stack (unless explicitly allocated on heap)
6. No memory overhead
7. Risk of overflow if size exceeded
8. Better memory locality
9. No additional functions needed for size management

**int[] staticArray = new int[5];**

Dynamic Arrays:

1. Size can change during runtime
2. Memory allocated at runtime
3. Can grow or shrink as needed
4. Slightly slower access time due to potential resizing
5. Memory allocated on heap
6. Has memory overhead for growth capacity
7. Automatically handles overflow by resizing
8. May have fragmented memory
9. Includes functions for size management (add, remove, resize)

**ArrayList<Integer> dynamicArray = new ArrayList<>(); // Size not needed**

**dynamicArray.add(1);**

**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.**

MCQ useful :

Bubble Sort

Best Case: O(n) - when array is already sorted

Worst Case: O(n²) - when array is reverse sorted

Selection Sort

Best Case: O(n²) - same as worst case, always performs same number of comparisons

Worst Case: O(n²) - needs to scan all elements for each position

Insertion Sort

Best Case: O(n) - when array is already sorted

Worst Case: O(n²) - when array is reverse sorted