Task01:

Algorithm

Start from index i = 0 to n-2

Find the smallest element from i to n-1

Swap it with element at index i

Repeat until entire array is sorted

Task02:

Pseudo Code

for i = 0 to n - 2 do

minIndex = i

for j = i + 1 to n - 1 do

if A[j] < A[minIndex] then

minIndex = j

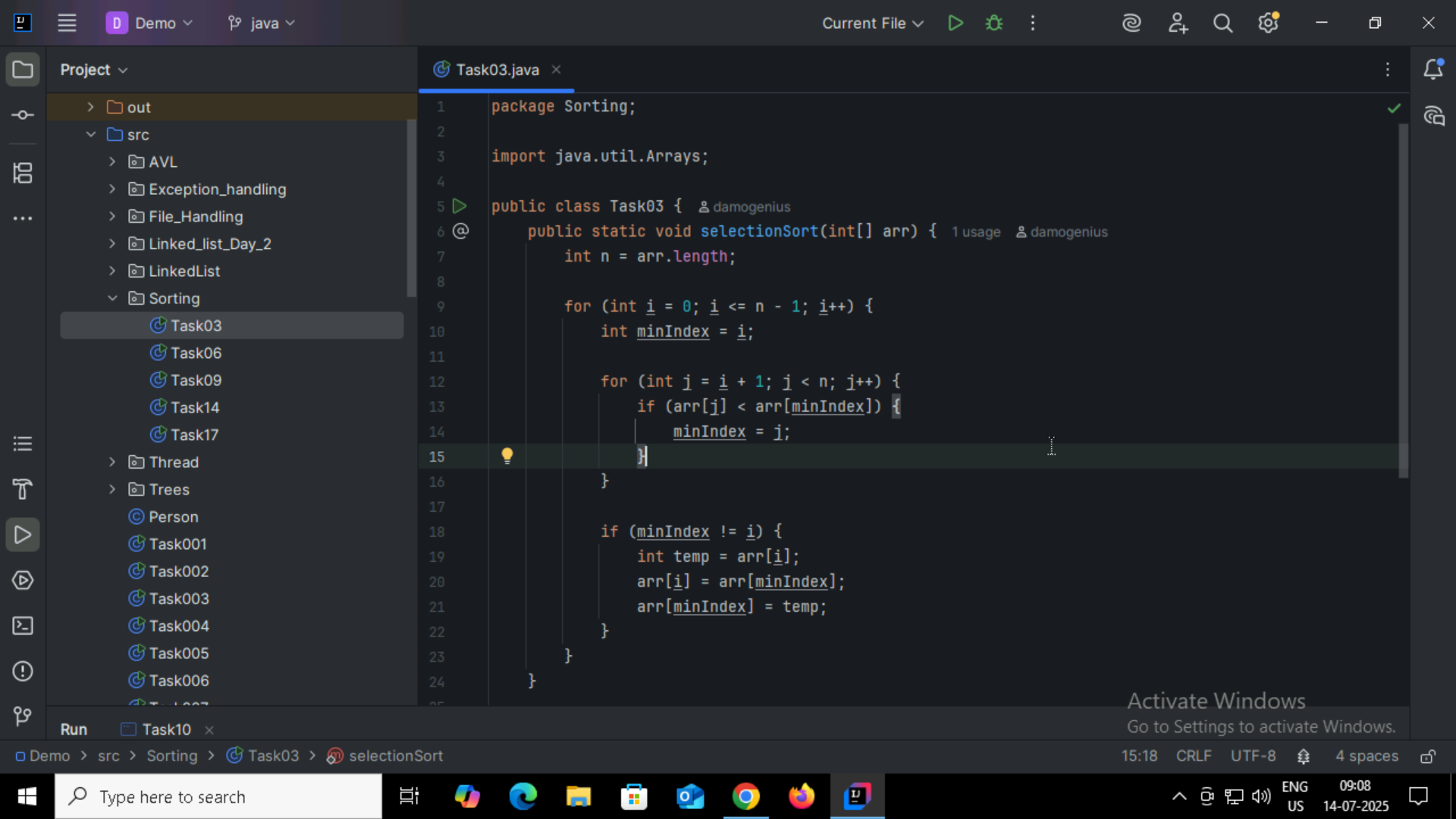
end for

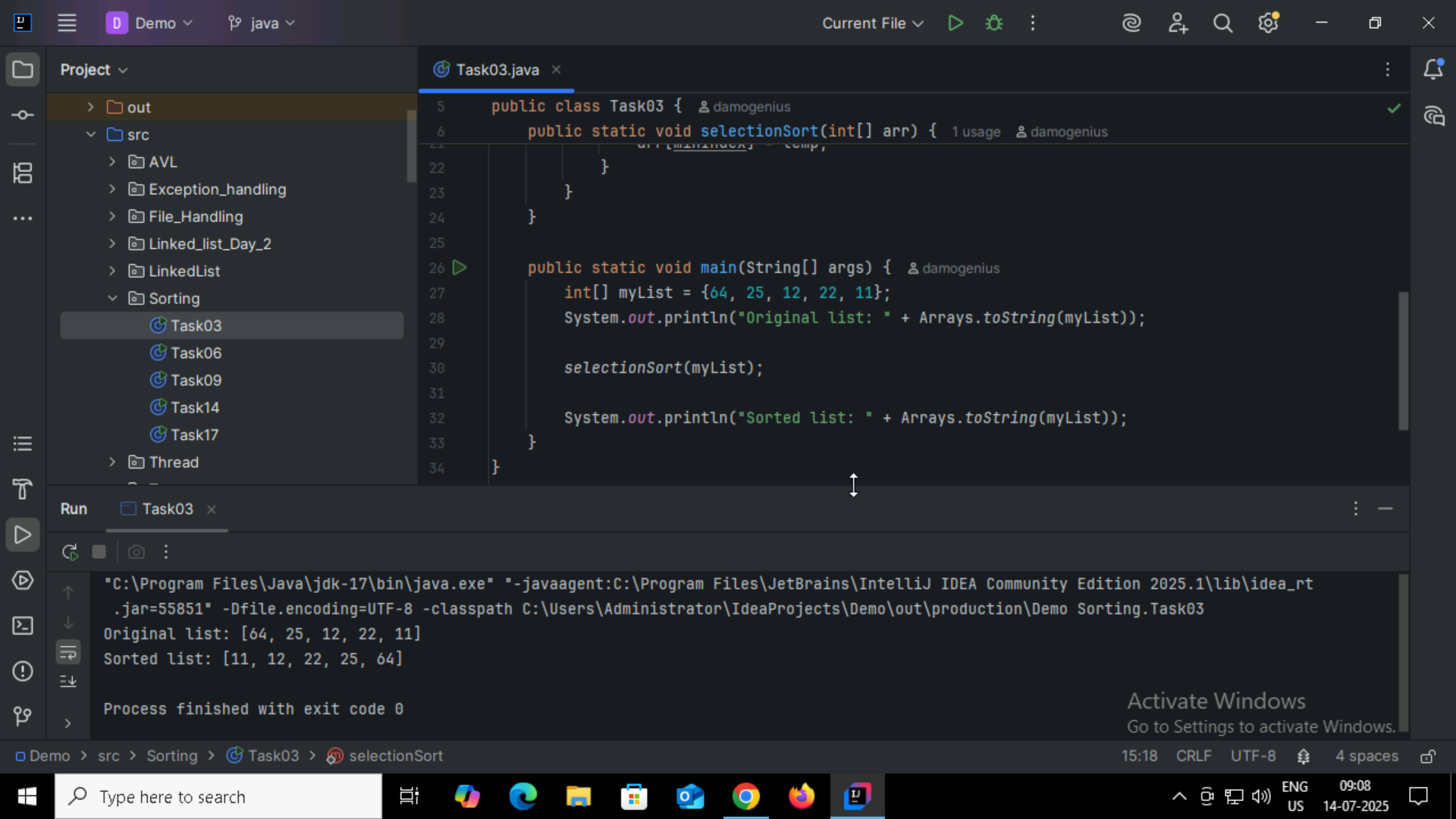
if minIndex != i then

swap A[i] and A[minIndex]

end for

Task03:





Task04:

**Step 1:** Start

**Step 2:** Repeat the following steps for all elements in the array, except the last one.

**Step 3:** For each pair of adjacent elements in the array:  
    **a.** Compare the current element with the next one.  
    **b.** If the current element is greater than the next one, swap them.

**Step 4:** Repeat the process for all remaining unsorted elements.

**Step 5:** Continue the process until the array is sorted (i.e., no swaps are needed in a full pass).

**Step 6:** Stop

Task05:

1. repeat

2. swapped = false

3. for i = 0 to n - 2 do

4. if A[i] > A[i + 1] then

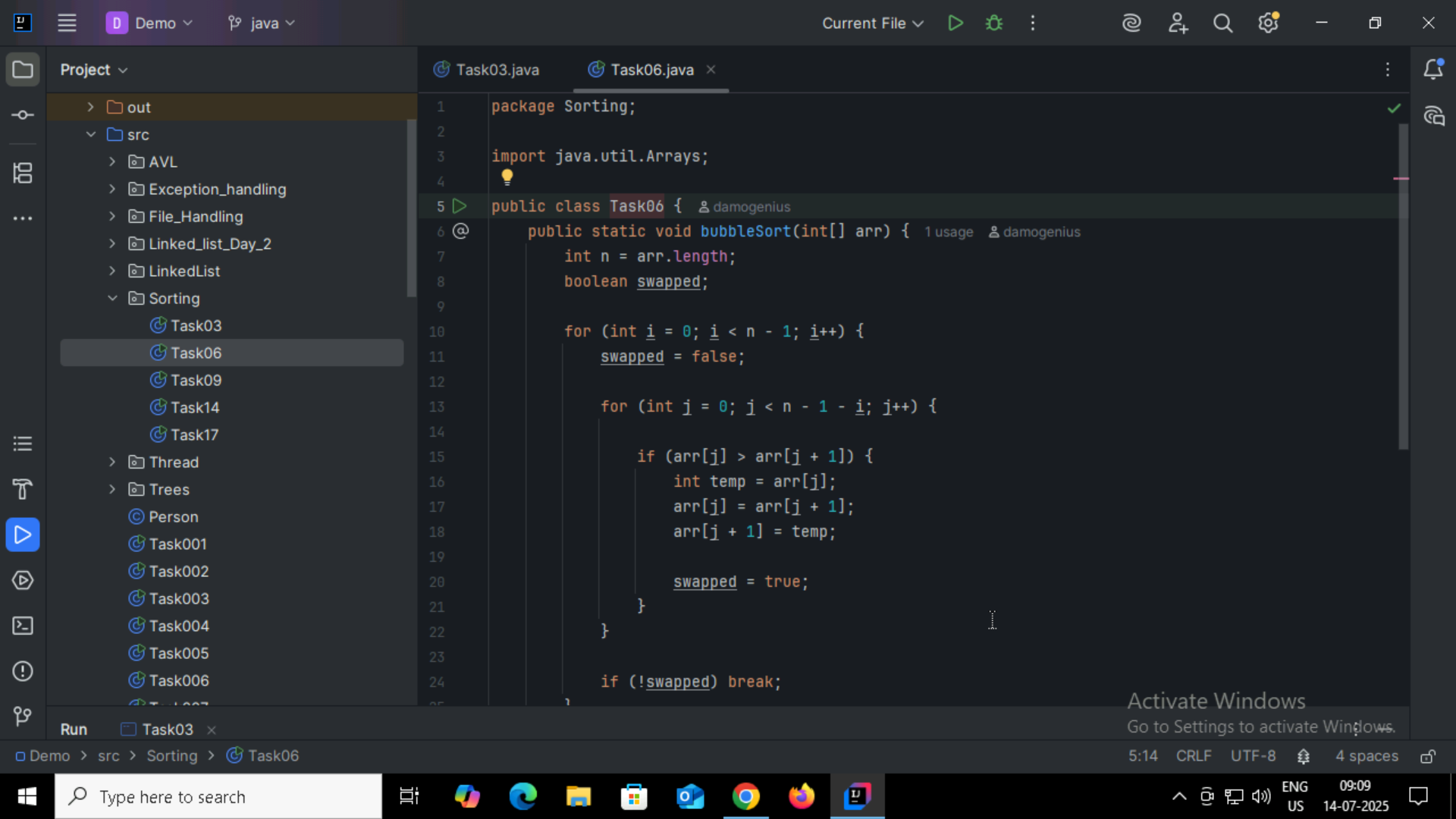
5. swap A[i] and A[i + 1]

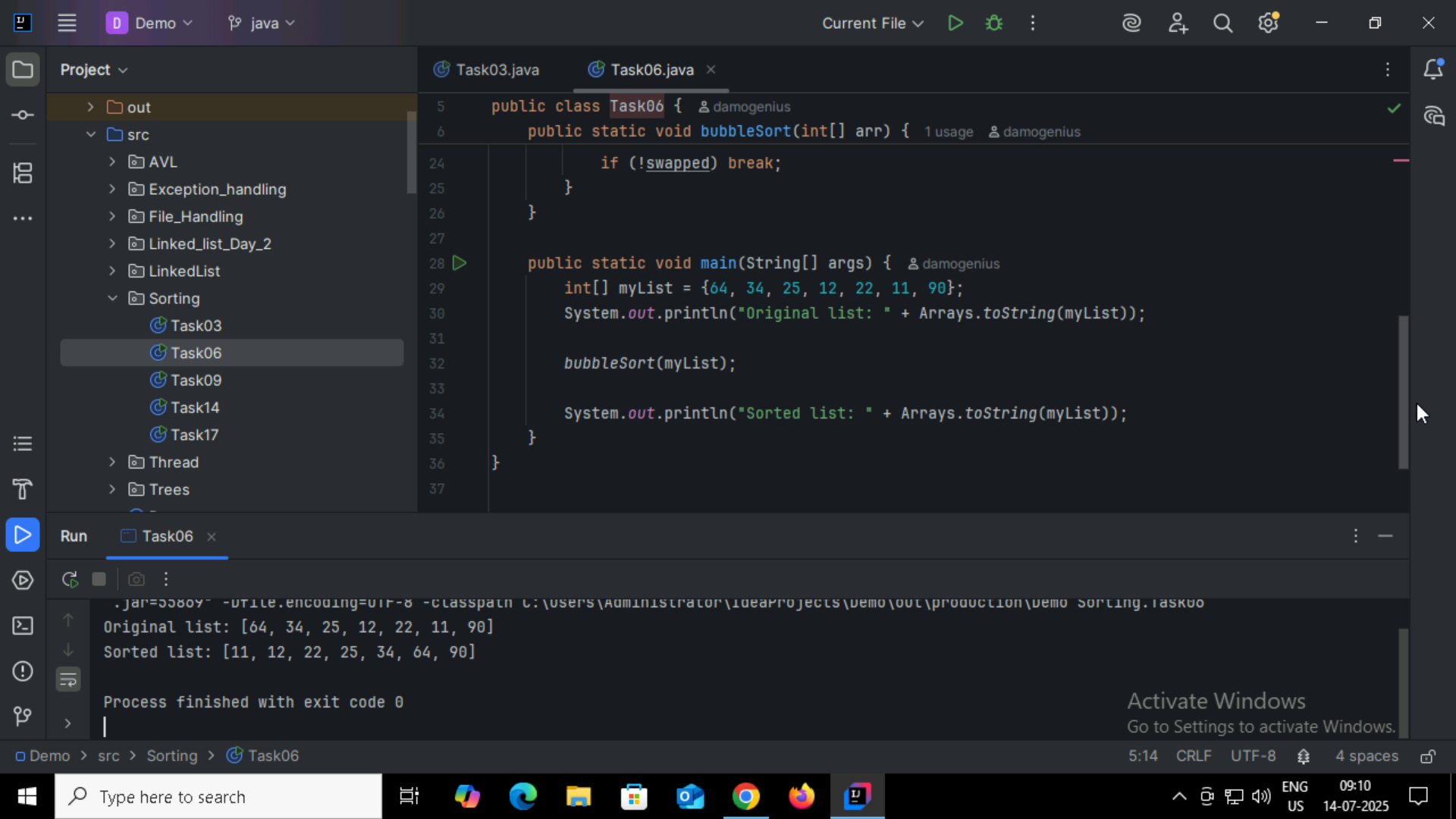
6. swapped = true

7. end for

8. until swapped = false

Task06:





Task07:

Algorithm: Insertion Sort (Ascending Order)

**Step 1:** Start

**Step 2:** Assume the first element of the array is already sorted.

**Step 3:** Take the next element from the unsorted part of the array.

**Step 4:** Compare it with elements in the sorted part of the array (from right to left).

**Step 5:** Shift all larger elements in the sorted part one position to the right.

**Step 6:** Insert the current element into its correct position in the sorted part.

**Step 7:** Repeat steps 3 to 6 until all elements are sorted.

**Step 8:** Stop

Task08:

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

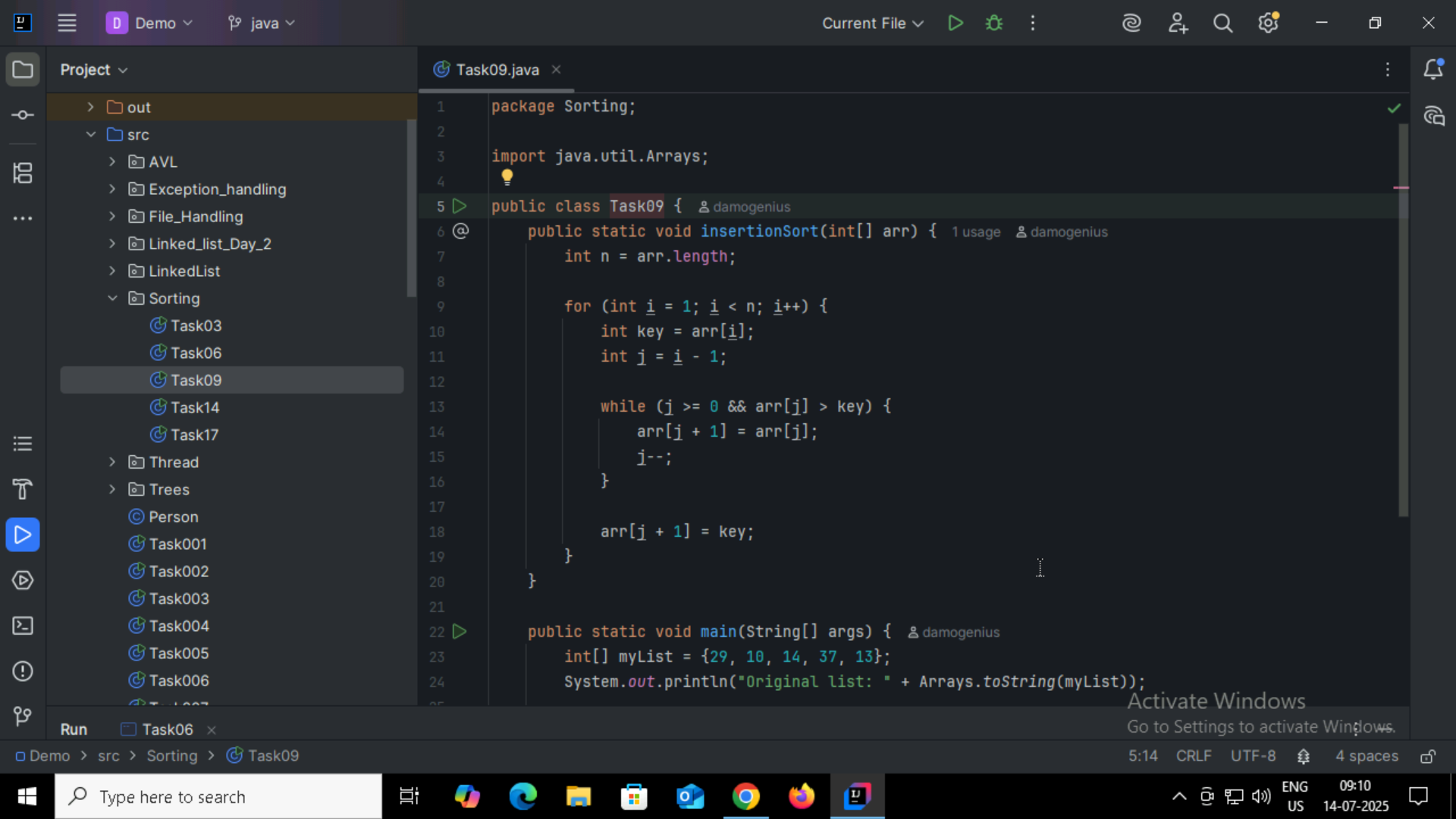
end while

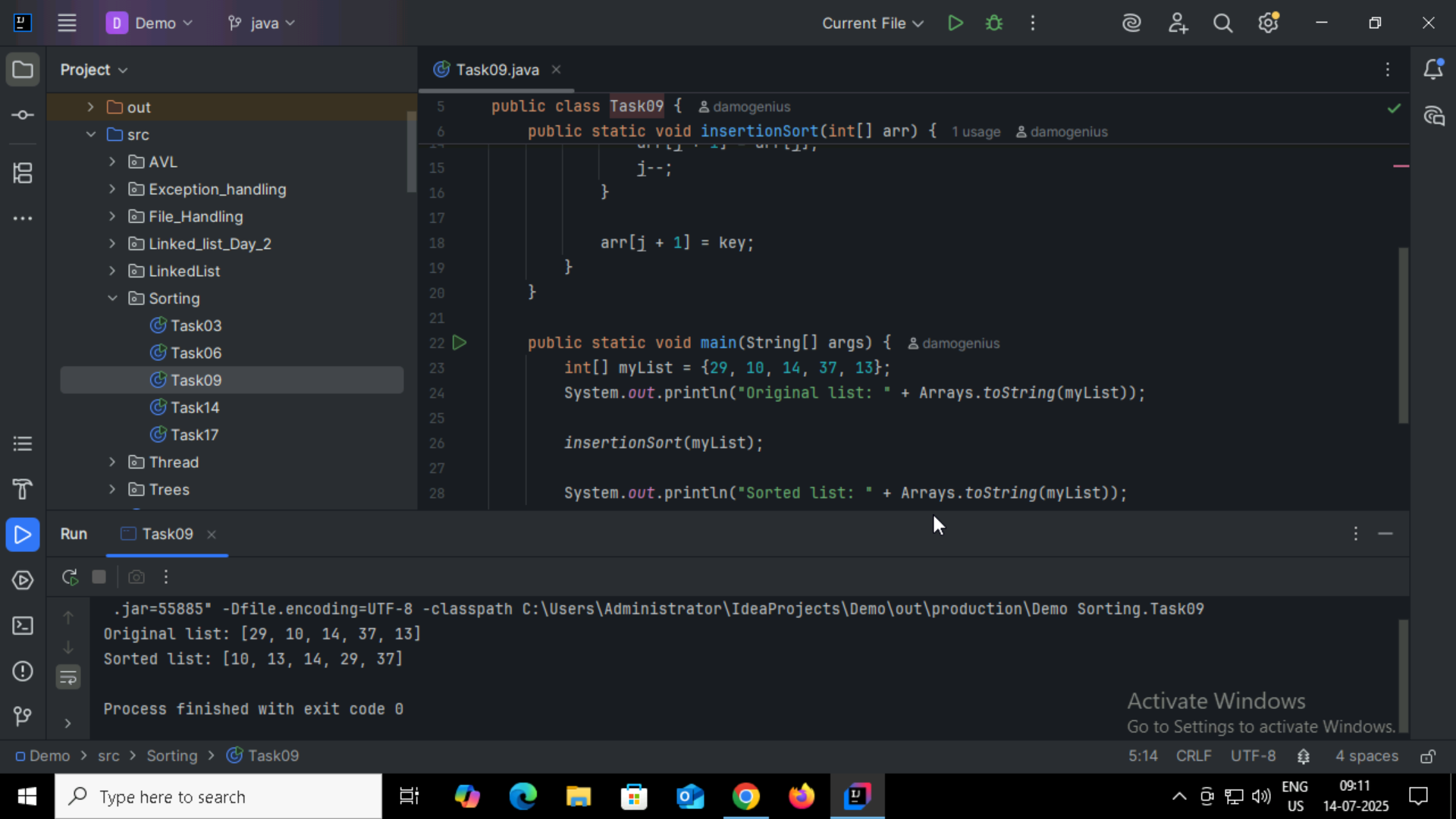
A[j + 1] = key

end for

end procedure

Task09:





Task10:

**Advantages:**

Simple and Easy to Understand

The algorithm is straightforward and easy to implement for beginners.

In-Place Sorting

Requires no additional memory (space complexity is O(1)).

Stable Sort

Maintains the relative order of equal elements.

Good for Small Data Sets

Early Termination Possible

**Limitations:**

Poor time complexity (O(n²))

Too many unnecessary operations

Not scalable

Outperformed by almost all modern sorting algorithms

Task11:

If n is large ,recursion may still overflow the stack. In that case, use an iterative approach or a tail-recursive optimization.

public class RecLoop {

public int calc(int n) {

if (n == 0) return 0;

return n + calc(n);

}

Corrected Code:

public class RecLoop {

public int calc(int n) {

if (n == 0) return 0;

return n + calc(n - 1);

}

Task 12 :

**Step 1:** Start

**Step 2:** If the array has more than one element:

* Divide the array into two halves.

**Step 3:** Recursively apply Merge Sort to each half.

**Step 4:** Merge the two sorted halves into a single sorted array:

* Compare elements from both halves one by one.
* Insert the smaller element into the result array.
* Continue until all elements are merged in order.

**Step 5:** Stop when the entire array is sorted.

Task 13:

procedure MergeSort(A, left, right)

if left < right then

mid ← (left + right) / 2

MergeSort(A, left, mid)

MergeSort(A, mid + 1, right)

Merge(A, left, mid, right)

end if

end procedure

procedure Merge(A, left, mid, right)

Create temporary arrays L and R

Copy data into L and R from A[left..mid] and A[mid+1..right]

i ← 0, j ← 0, k ← left

while i < length(L) and j < length(R) do

if L[i] ≤ R[j] then

A[k] ← L[i]

i ← i + 1

else

A[k] ← R[j]

j ← j + 1

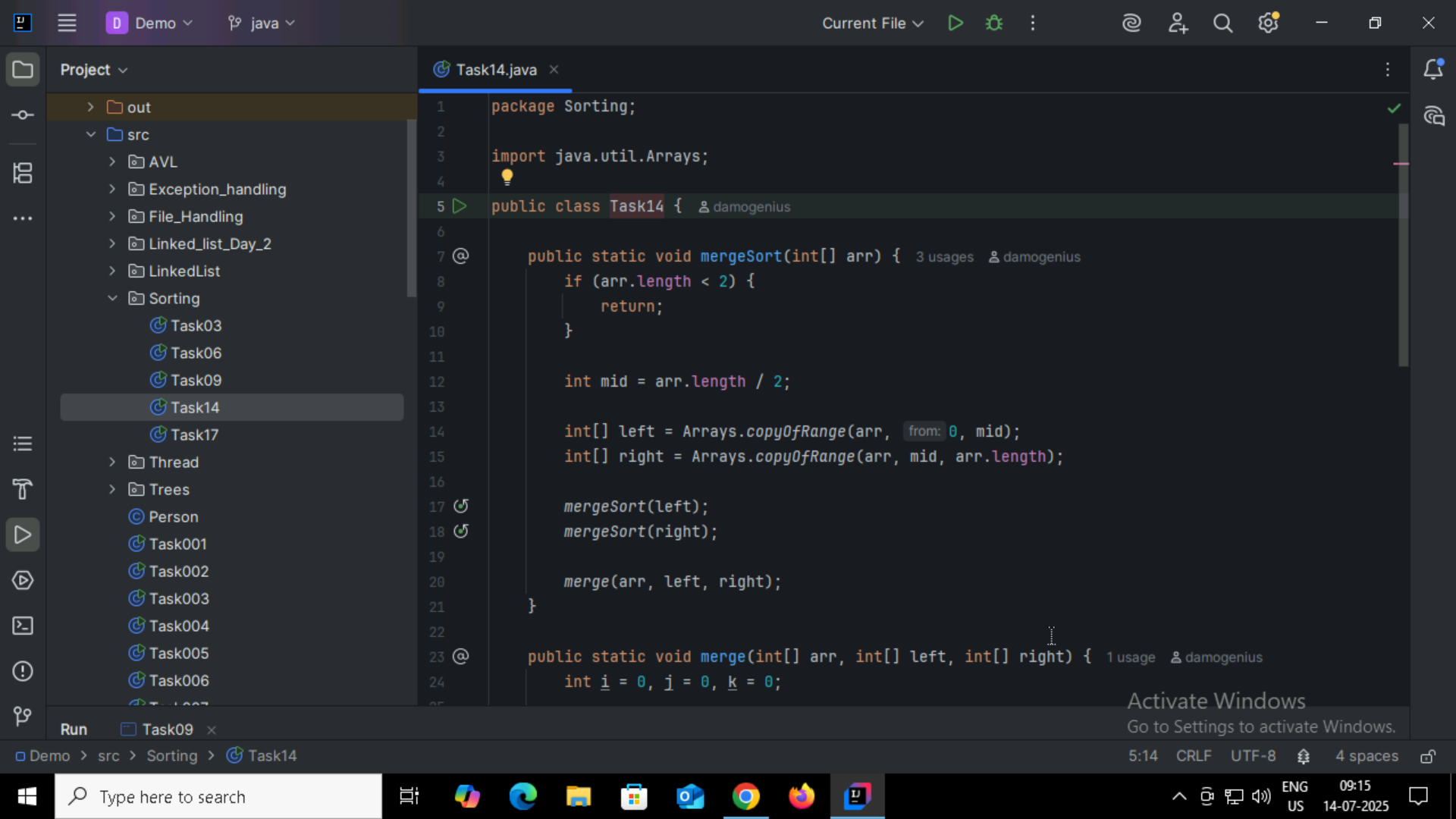
k ← k + 1

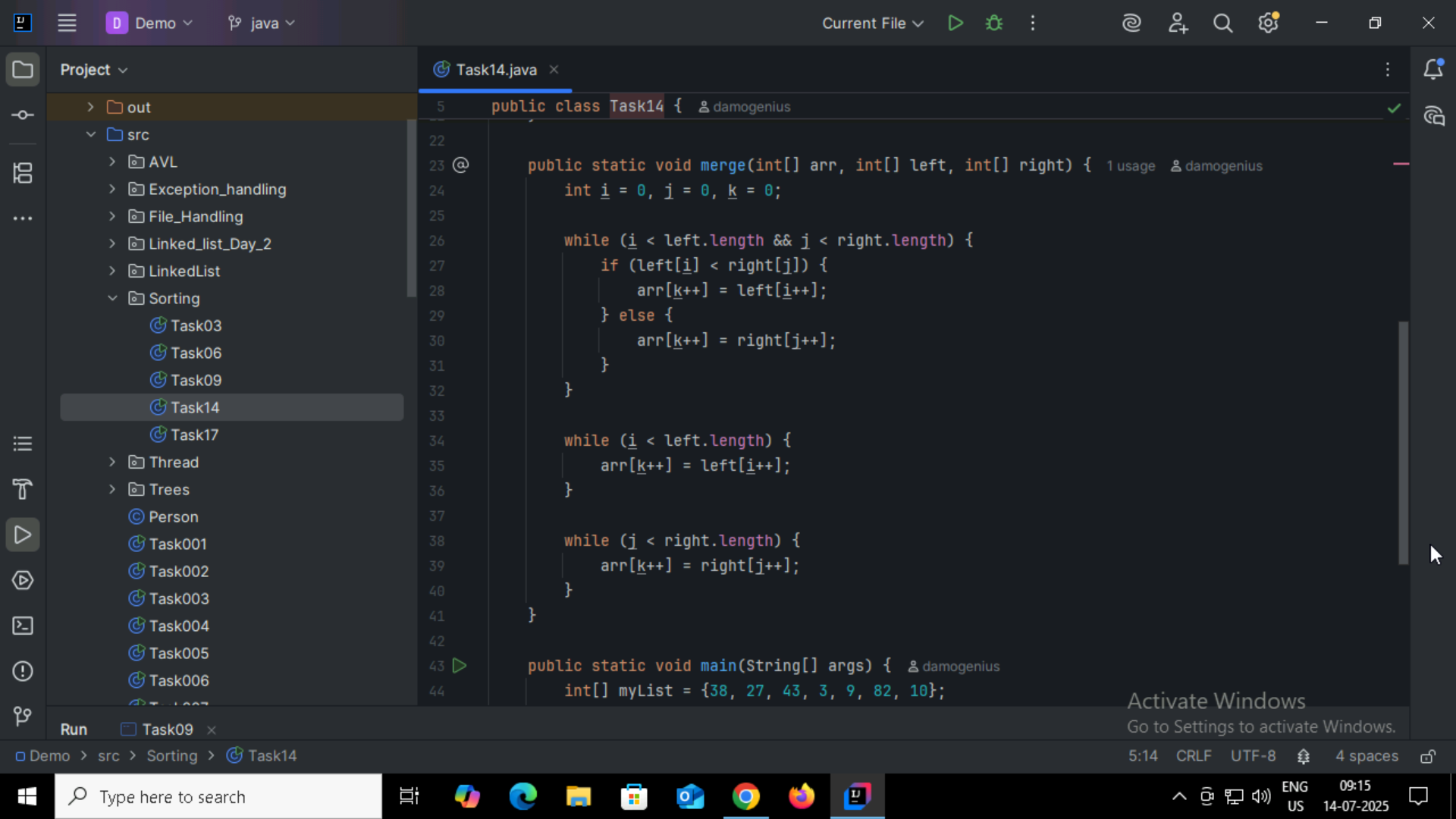
Copy any remaining elements of L

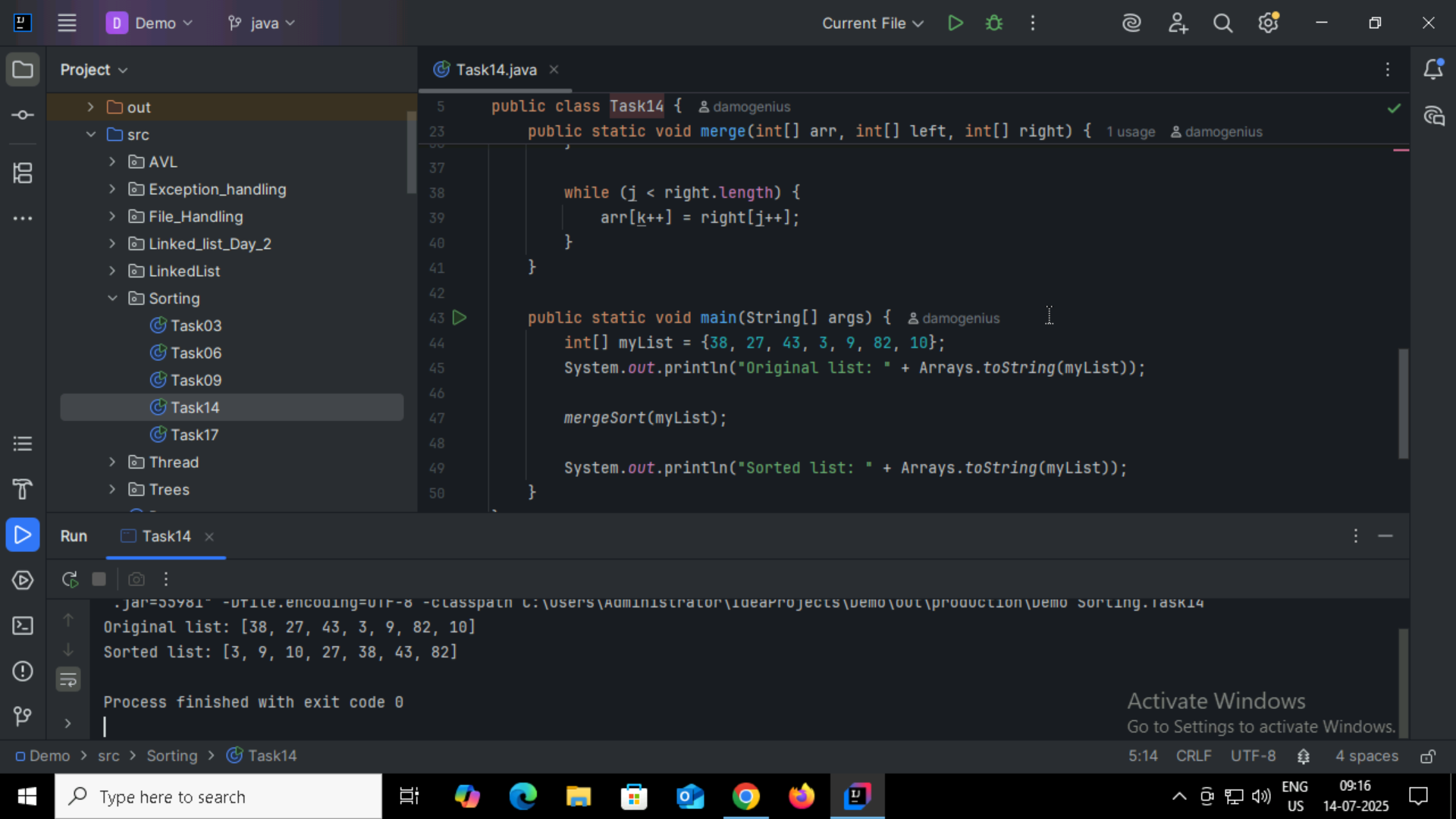
Copy any remaining elements of R

end procedure

Task14:







Task 15:

**Step 1:** Start

**Step 2:** Choose a **pivot element** from the array (commonly the first, last, or middle element).

**Step 3:** Partition the array:

* Place elements **less than the pivot** to its **left**
* Place elements **greater than the pivot** to its **right**

**Step 4:** Recursively apply Quick Sort to the subarrays:

* Left subarray (elements less than pivot)
* Right subarray (elements greater than pivot)

**Step 5:** Continue until each subarray has 0 or 1 element (base case)

**Step 6:** Stop – the array is now sorted

Task 16:

procedure QuickSort(A, low, high)

if low < high then

pivotIndex ← Partition(A, low, high)

QuickSort(A, low, pivotIndex - 1)

QuickSort(A, pivotIndex + 1, high)

end if

end procedure

procedure Partition(A, low, high)

pivot ← A[high]

i ← low - 1

for j ← low to high - 1 do

if A[j] < pivot then

i ← i + 1

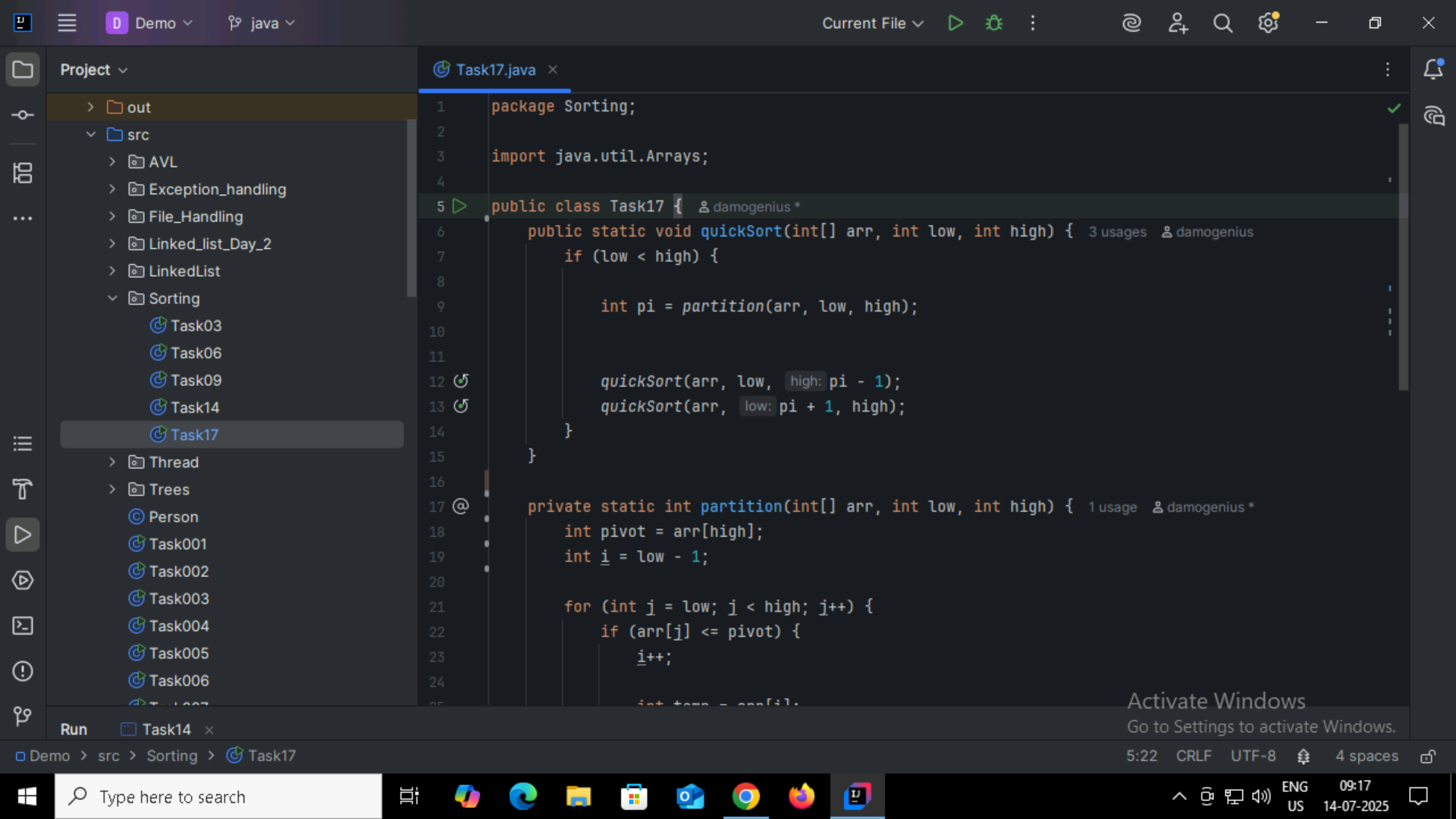
swap A[i] and A[j]

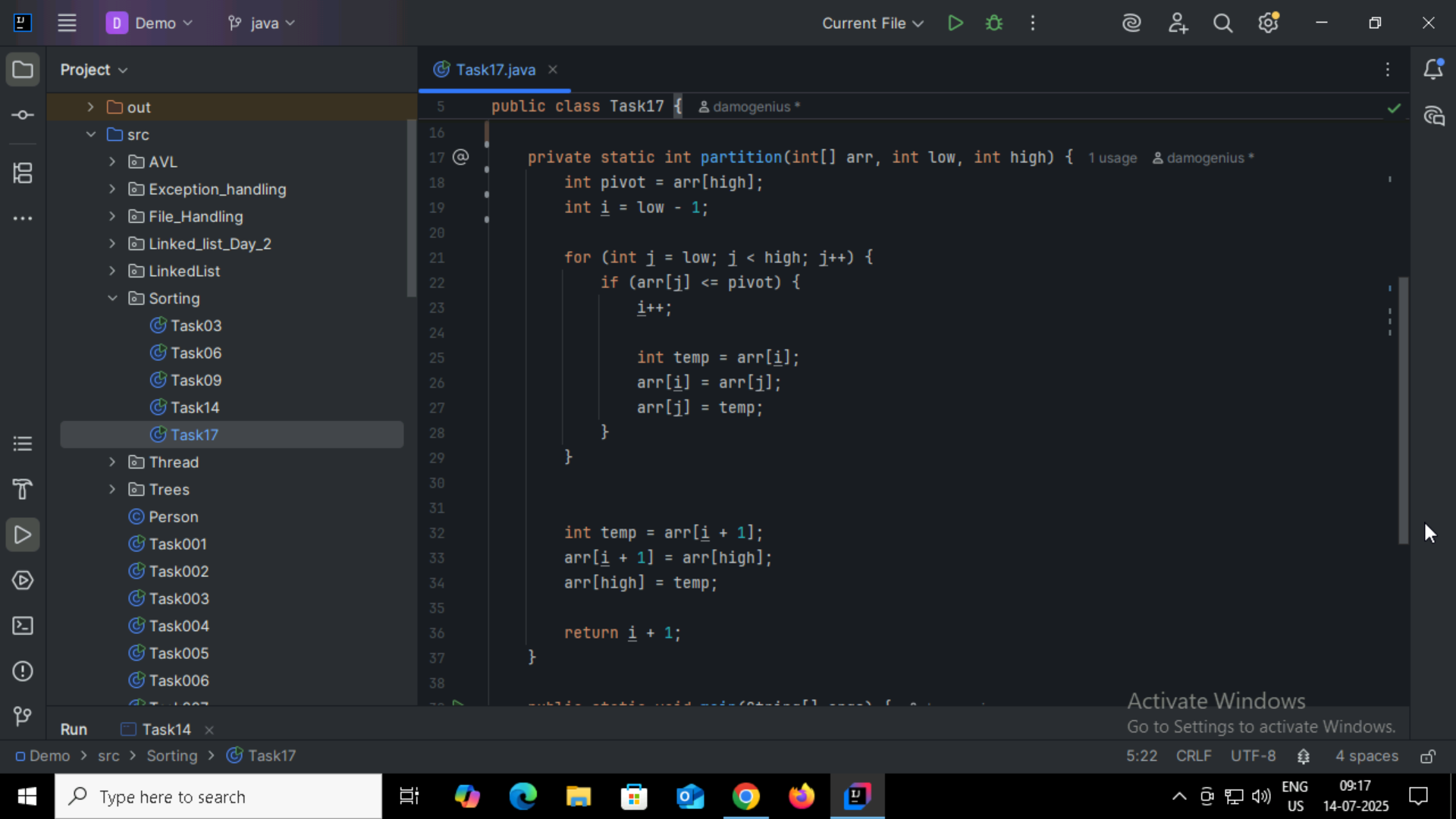
swap A[i + 1] and A[high]

return i + 1

end procedure

Task17:







Home Task:  
1:

| **Feature** | **Binary Tree** | **Binary Search Tree (BST)** |
| --- | --- | --- |
| **Definition** | A tree in which each node has at most 2 children. | A binary tree with elements arranged in sorted order. |
| **Structure** | No specific ordering of nodes. | Left child < Parent < Right child. |
| **Data Order** | Random or application-specific. | Follows strict ordering rules. |
| **Search Efficiency** | O(n) in worst case (no ordering). | O(log n) on average (if balanced). |
| **Insertion Rule** | No fixed rule. | Insert smaller values to the left, larger to the right. |
| **Use Case** | General tree problems (e.g., parsing expressions). | Fast searching, insertion, and deletion of ordered data. |
| **Traversal Type** | Preorder, Inorder, Postorder, Level order. | Same traversals apply, but **inorder** gives sorted data. |
| **Duplicates Allowed?** | Yes, depends on implementation. | Usually not allowed (or placed in consistent way). |

2:

If you already have a sorted array, then BST is more efficient than linear search.

**Binary Search Tree (BST)**

* **Approach**: Tree-based structure; values organized so that left < root < right.
* **Search Time**: O(log n) on average (if balanced)
* **Insert/Delete Time**: Also O(log n) on average
* **Advantage**: Good for **dynamic** data — where frequent **insertions/deletions** happen.

**Linear Search in Array**

* **Approach**: Go through every element one by one.
* **Time Complexity**: O(n)
* **Efficiency**: Slow for large data.
* **Advantage**: Works on unsorted data.

3:

| **Feature** | **Static Array** | **Dynamic Array** |
| --- | --- | --- |
| **Size** | Fixed at compile-time | Can grow or shrink at runtime |
| **Memory Allocation** | Done at compile-time | Done at runtime (heap) |
| **Resizing** | Not possible | Possible (automatically or manually) |
| **Memory Usage** | May waste space if not fully used | Efficient use of memory (adjustable size) |
| **Performance** | Faster (no resizing overhead) | Slight overhead during resizing |
| **Ease of Use** | Simple to declare and use | Needs memory management (in low-level) |
| **Storage Location** | Stack (usually) | Heap (usually) |
| **Flexibility** | Less flexible | More flexible for dynamic data needs |

4:

Which is more preferred for shortest path in an unweighted graph?

BFS (Breadth-First Search) is preferred.

Reason:

* In unweighted graphs, the shortest path is the one with the fewest number of edges.
* BFS explores nodes level by level:
  + It visits all nodes at distance 1 from the source, then distance 2, and so on.
  + As soon as it reaches the destination node, it has found the shortest path.

5:

