Task 1:

Write an algorithm / steps for selection sort?

First Spot:

Look at our whole messy pile: [6, 4, 1, 8, 3]

Find the smallest: Scan through it. "Is 6 the smallest? No, 4 is smaller. Is 4 the smallest? No, 1 is smaller. Is 1 the smallest? Yes, compared to 8 and 3. So, 1 is the smallest!"

Move it to the first spot: Take that 1 and swap it with whatever is currently in the first spot (6).

Your pile now looks like: [1, 4, 6, 8, 3] (Notice 1 is now in its correct place).

Second Spot:

Now, forget about the 1 (it's sorted). Look only at the remaining messy part of your pile: [4, 6, 8, 3]

Find the smallest: Scan this part. "Is 4 the smallest? No, 3 is smaller. Is 3 the smallest? Yes."

Move it to the second spot: Take that 3 and swap it with whatever is currently in the second spot (4).Your pile now looks like: [1, 3, 6, 8, 4] (Notice 1 and 3 are in their correct places).

Third Spot:

Ignore 1 and 3. Look at the remaining messy part: [6, 8, 4]

Find the smallest: Scan this part. "Is 6 the smallest? No, 4 is smaller. Is 4 the smallest? Yes."

Move it to the third spot: Take that 4 and swap it with whatever is currently in the third spot (6).

Your pile now looks like: [1, 3, 4, 8, 6] (Notice 1, 3, and 4 are in their correct places).

Fourth Spot:

Ignore 1, 3, 4. Look at the remaining messy part: [8, 6]

Find the smallest: Scan this part. "Is 8 the smallest? No, 6 is smaller. Is 6 the smallest? Yes."

Move it to the fourth spot: Take that 6 and swap it with whatever is currently in the fourth spot (8).

Your pile now looks like: [1, 3, 4, 6, 8] (Notice 1, 3, 4, and 6 are in their correct places).

When we've put N-1 items in their correct places, the last item automatically has to be in its correct place too.

Task 2:

Write a pseudo code for the selection sort?

Algorithm SelectionSort(Array arr, N)

FOR i FROM 0 TO N-2

min\_idx = i

FOR j FROM i+1 TO N-1

IF arr[j] < arr[min\_idx] THEN

min\_idx = j

END IF

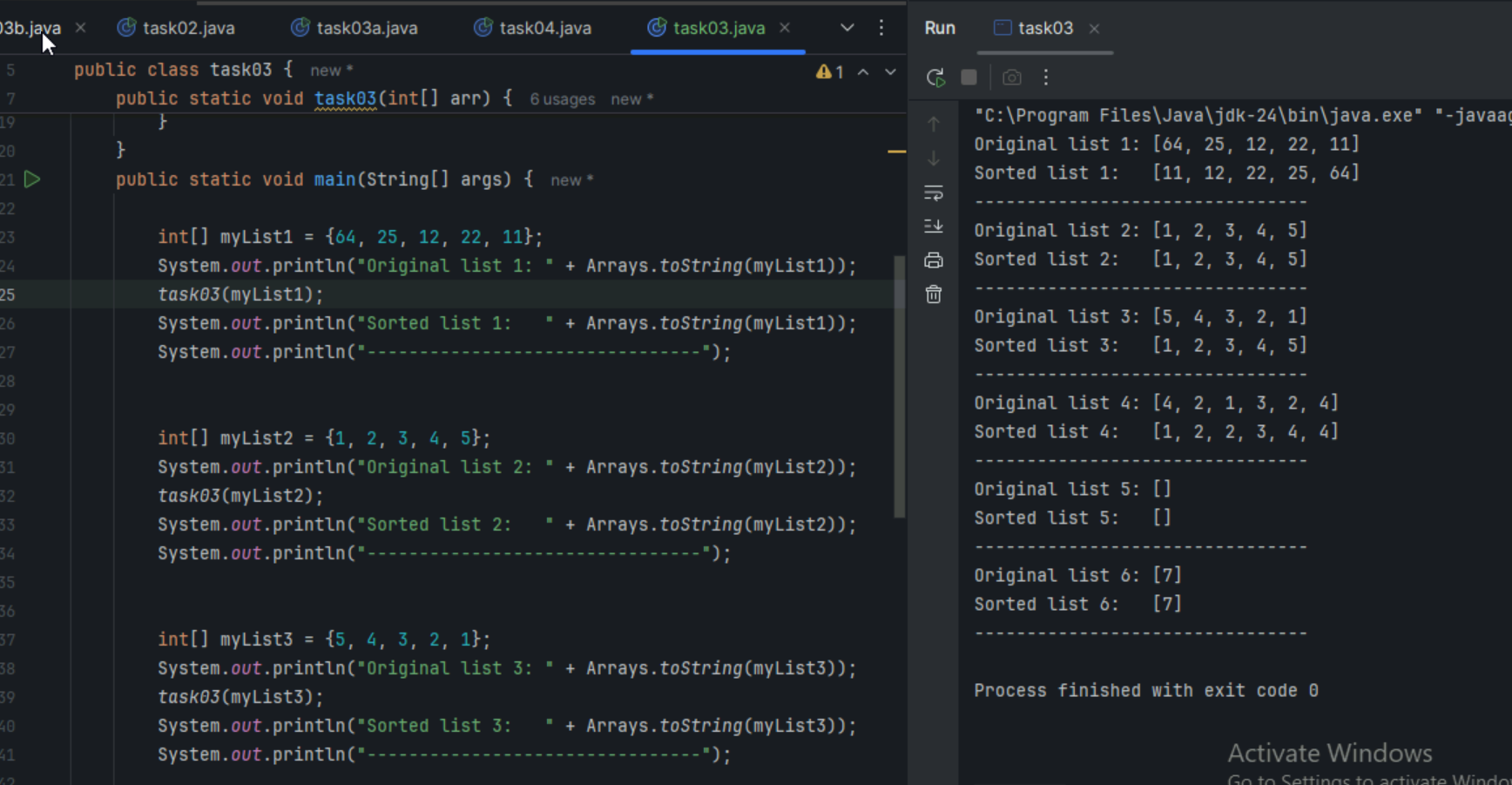
END FOR

Swap(arr[i], arr[min\_idx])

END FOR

END Algorithm

Task 03:

****

**Task 4:**

Write algorithm for the Bubble sort.

our messy line (array) of numbers is: [5, 1, 4, 2, 8] (We want them smallest to largest).

Round 1 (First "Pass" through the line):

Look at (5, 1): Is 5 bigger than 1? Yes, Swap them.

Line: [1, 5, 4, 2, 8]

Look at (5, 4): (Now 5 is in the second spot). Is 5 bigger than 4? Yes, Swap them.

Line: [1, 4, 5, 2, 8]

Look at (5, 2): (Now 5 is in the third spot). Is 5 bigger than 2? Yes, Swap them.

Line: [1, 4, 2, 5, 8]

Look at (5, 8): (Now 5 is in the fourth spot). Is 5 bigger than 8? No, so Don't swap.

Line: [1, 4, 2, 5, 8]

End of Round 1: Notice that the 8 (the biggest number) has "bubbled" all the way to the end. It's now in its correct final spot

Round 2 (Second "Pass"):

We don't need to check the last spot anymore, because 8 is already where it belongs.

Look at (1, 4): Is 1 bigger than 4? No,so Don't swap.

Line: [1, 4, 2, 5, 8]

Look at (4, 2): Is 4 bigger than 2? Yes, Swap them.

Line: [1, 2, 4, 5, 8]

Look at (4, 5): (Now 4 is in the third spot). Is 4 bigger than 5? No, so Don't swap.

Line: [1, 2, 4, 5, 8]

End of Round 2: The 5 (the next biggest) has bubbled to its correct final spot!

Round 3 (Third "Pass"):

Now we don't need to check the last two spots (8 and 5).

Look at (1, 2): Is 1 bigger than 2? No, so Don't swap.

Line: [1, 2, 4, 5, 8]

Look at (2, 4): Is 2 bigger than 4? No, so Don't swap.

Line: [1, 2, 4, 5, 8]

End of Round 3: The 4 is now in its correct spot.

Task 5:

for Bubble Sort (Ascending Order)

Algorithm BubbleSort(Array arr, N)

// Input:// arr: The list/array of elements you want to sort.

// N: The total number of elements in that list/array.

// Output: // The original array 'arr', but now sorted in ascending order (smallest to largest).

// It modifies the array directly (in-place sorting).

// Outer Loop: This loop controls how many full "passes" we make through the list.

// We need to make at most N-1 passes, because after N-1 passes, the array

// will definitely be sorted.

FOR i FROM 0 TO N-2 // 'i' goes from 0 up to (N-2)

// Flag to track if any swaps happened in this pass.

// If no swaps occur in a whole pass, it means the list is already sorted,

// and we can stop early (optimization!).

swapped\_in\_this\_pass = FALSE

// Inner Loop: This loop is where the actual comparisons and swaps happen.

// It iterates through the unsorted part of the list.

// The '-i' part is a smart optimization: After 'i' passes, the last 'i' elements

// are already in their correct sorted positions, so we don't need to check them again.

FOR j FROM 0 TO N-2-i // 'j' goes from 0 up to (N-2-i)

// Compare the current element (arr[j]) with the next element (arr[j+1]).

// If the current element is GREATER than the next one (for ascending sort),

// then they are in the wrong order.

IF arr[j] > arr[j+1] THEN

// Swap them: Move the larger element to the right, and the smaller to the left.

// This is like the "bubble" moving up.

Swap(arr[j], arr[j+1]) // Conceptual swap operation

// Since a swap occurred, set the flag to TRUE.

swapped\_in\_this\_pass = TRUE

END IF

END FOR // End of Inner Loop

// Check the flag after a full pass (inner loop) is complete.

// If 'swapped\_in\_this\_pass' is still FALSE, it means no elements were out of order,

// so the entire array is now sorted. We can stop.

IF swapped\_in\_this\_pass IS FALSE THEN

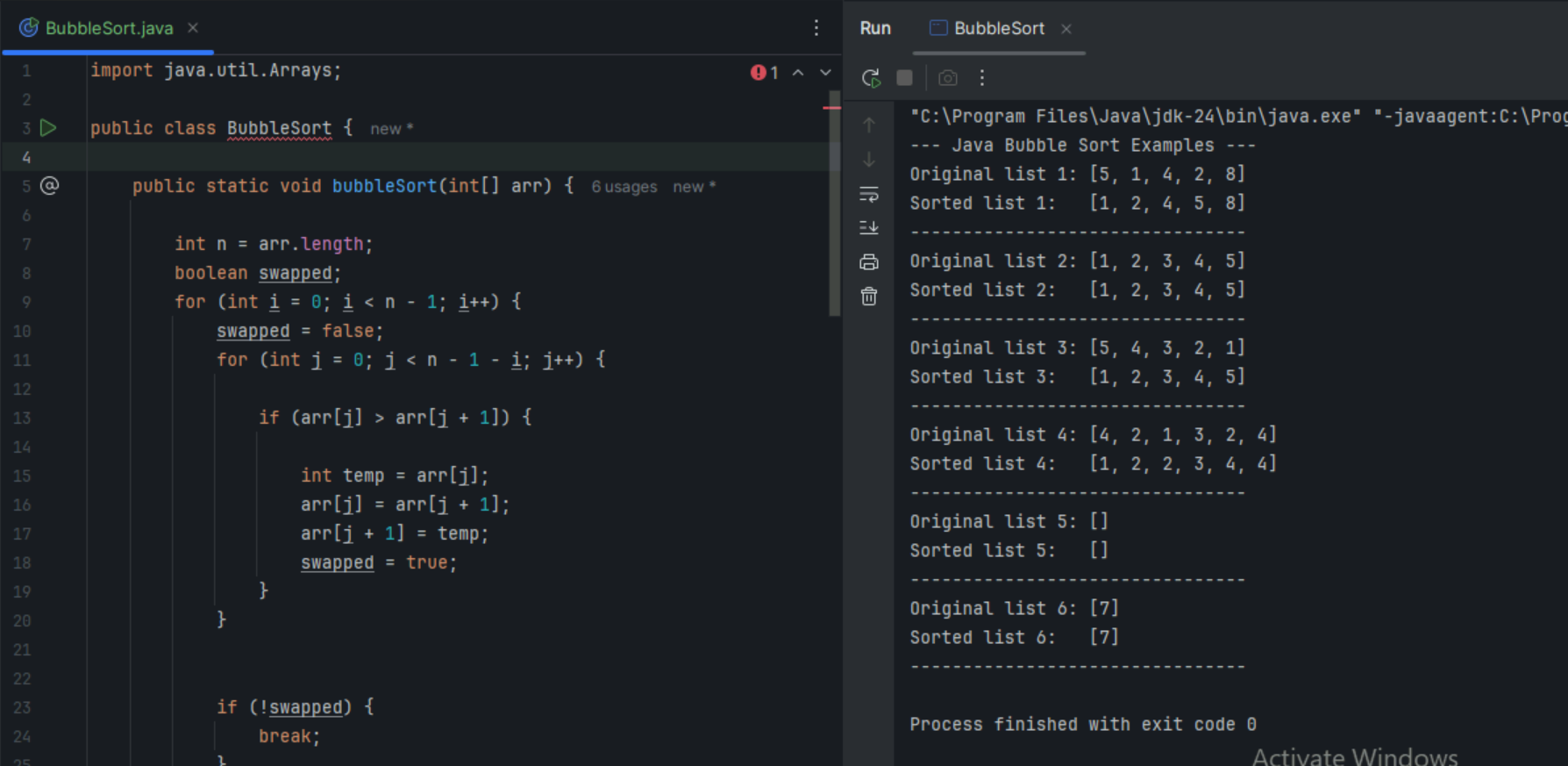
BREAK // Exit the outer loop early

END IF

END FOR // End of Outer Loop

END Algorithm

Task 6:



Task 7:

Iam playing cards, and we are picking up cards one by one. i want to keep my hand sorted as i pick them up.

Input: An array arr of N elements.

Output: The same array arr, but sorted in ascending order.

Start with a "Sorted" Section: Assume the first element (arr[0]) is already sorted. Your "sorted part" is just this single element.

Outer Loop (Iterate through Unsorted Elements)

Start a loop from the second element (i = 1) up to the last element (i = N-1).

Each arr[i] in this loop is the "current element" you want to insert into the sorted part.

Store the Current Element:

Take arr[i] and store it in a temporary variable (let's call it key). This is the card you've just picked up.

Inner Loop (Find Correct Position and Shift Elements):

Initialize another variable j to i - 1 (this j points to the last element of your current sorted section).

Start a while loop that continues as long as two conditions are met:

j is greater than or equal to 0 (meaning you haven't reached the beginning of the array).

arr[j] (an element in your sorted section) is greater than key (your new card).

Shift: Inside this while loop, if arr[j] is greater than key, shift arr[j] one position to the right: arr[j+1] = arr[j].

Move Left: Decrement j by 1 (j = j - 1) to move to the next element on the left in the sorted section and continue the comparison.

Insert the Element:

When the while loop finishes, it means you've either found an element smaller than key (or reached the beginning of the array). The spot j+1 is where key should be inserted.

Place key into arr[j+1].

Repeat:

The outer loop continues to the next element arr[i], and the process repeats until all elements have been inserted into their correct places.

Task 8:

Algorithm InsertionSort(Array arr, N)

Input:

arr: The array of elements to be sorted.

N: The number of elements in the array.

Output: The array 'arr' sorted in ascending order (modified in-place).

Outer loop: Iterate from the second element to the end of the array.

FOR i FROM 1 TO N-1

key = arr[i]

'j' points to the last element of the sorted subarray (arr[0...i-1]).

j = i - 1

WHILE j >= 0 AND arr[j] > key

arr[j+1] = arr[j]

j = j - 1

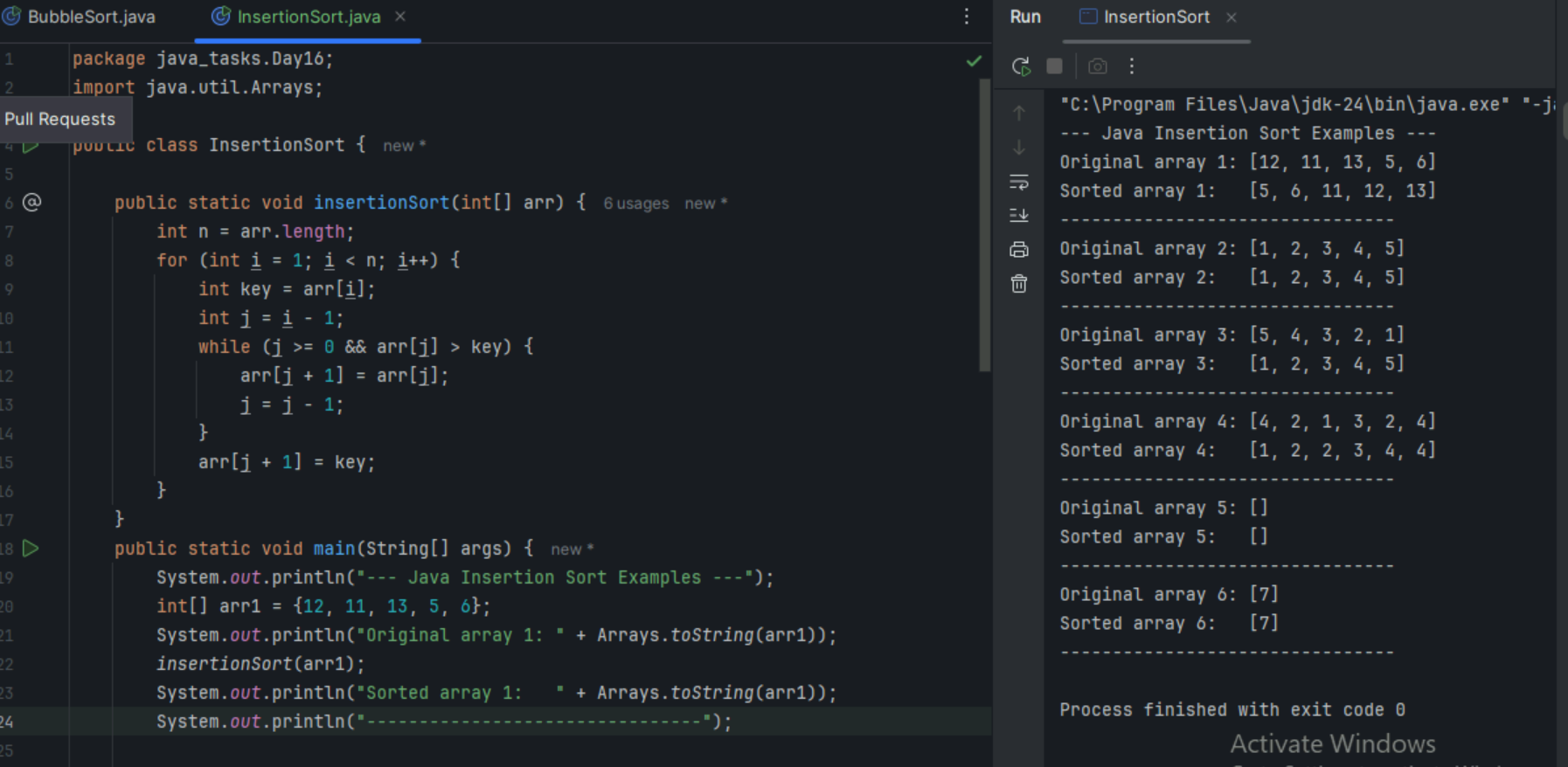
END WHILE

arr[j+1] = key

END FOR

END Algorithm

Task 9:



Task 10:

What are the advantages and disadvantages of Bubble sort Algo?

List them

note:

Poor performance - limitations of bubble sort

Bubble Sort is like tidying your messy room by only swapping things that are right next to each other if they're in the wrong spot.

Advantages :

🡪Super Easy to Understand and Code

🡪Space Efficient (Doesn't Need Extra Room

🡪Stable Sort (Keeps Original Order of Equal Things

🡪Can Be Fast for Already Sorted Lists (With a Clever Trick

Disadvantages :

🡪Super Slow for Most Lists (The BIGGEST Problem

🡪 Many Unnecessary Comparisons and Swaps

🡪 Inefficient for "Out-of-Place" Elements

🡪 Rarely Used in Real-World Applications

Bubble Sort is our friendly, simple sorting method for learning, but it's way too slow and takes too many steps to be useful for anything but the tiniest lists or very specific (and rare) situations where the list is almost perfectly sorted already.

Task 11:

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

TAsk 12:

Algo for merge sort?

If i have a huge stack of unsorted papers, and you want to sort them alphabetically.

i decide to split the stack into two smaller stacks. Then i tell my friend to sort one half, and i sort the other. But my friend also decides to split their stack, and so on, until everyone has just one paper, which is sorted.

Once everyone has sorted their tiny stacks, i start combining them. i and my friend take my two sorted halves and carefully merge them back into one bigger sorted stack, always picking the next paper in order from either my stack or their stack. This merging continues upwards until i have one giant, perfectly sorted stack.

Algorithm : Divide the array into two halves, recursively sort each half, and then merge the sorted halves.

mergeSort(arr, low, high) - The Dividing & Conquering Part:

Base Case: If low (start index) is greater than or equal to high (end index), it means the sub-array has 0 or 1 element, which is already considered sorted. So, stop.

Divide: Find the middle index of the current sub-array: mid = low + (high - low) / 2.

Conquer (Recursively Sort Halves):

Call mergeSort(arr, low, mid) to sort the left half.

Call mergeSort(arr, mid + 1, high) to sort the right half.

Combine (Merge): After both halves are sorted, call a merge(arr, low, mid, high) function to combine them into one sorted sub-array.

merge(arr, low, mid, high) –

The Merging Part:

This function takes two already sorted sub-arrays: arr[low...mid] and arr[mid+1...high].

Create two temporary arrays, leftTemp[] and rightTemp[], to hold elements from the two sub-arrays. Copy the elements into them.

Initialize three pointers: i for leftTemp, j for rightTemp, and k for the original arr (starting at low).

Compare and Copy: While i is within bounds of leftTemp AND j is within bounds of rightTemp:

If leftTemp[i] is less than or equal to rightTemp[j], copy leftTemp[i] to arr[k], then increment i and k.

Else (if rightTemp[j] is smaller), copy rightTemp[j] to arr[k], then increment j and k.

Copy Remaining Elements: After one of the temporary arrays runs out of elements, copy any remaining elements from the other temporary array into arr.

Task 13

pseudo code for merge sort,

Algorithm MergeSort(arr, low, high)

IF low < high THEN

mid = low + (high - low) / 2

MergeSort(arr, low, mid)

MergeSort(arr, mid + 1, high)

Merge(arr, low, mid, high)

END IF

END Algorithm

Algorithm Merge(arr, low, mid, high)

n1 = mid - low + 1

n2 = high - mid

LeftTempArray[n1]

RightTempArray[n2]

FOR i FROM 0 TO n1-1

LeftTempArray[i] = arr[low + i]

END FOR

FOR j FROM 0 TO n2-1

RightTempArray[j] = arr[mid + 1 + j]

END FOR

i = 0 // Initial index for LeftTempArray

j = 0 // Initial index for RightTempArray

k = low // Initial index for merged subarray in arr

WHILE i < n1 AND j < n2

IF LeftTempArray[i] <= RightTempArray[j] THEN

arr[k] = LeftTempArray[i]

i = i + 1

ELSE

arr[k] = RightTempArray[j]

j = j + 1

END IF

k = k + 1

END WHILE

WHILE i < n1

arr[k] = LeftTempArray[i]

i = i + 1

k = k + 1

END WHILE

WHILE j < n2

arr[k] = RightTempArray[j]

j = j + 1

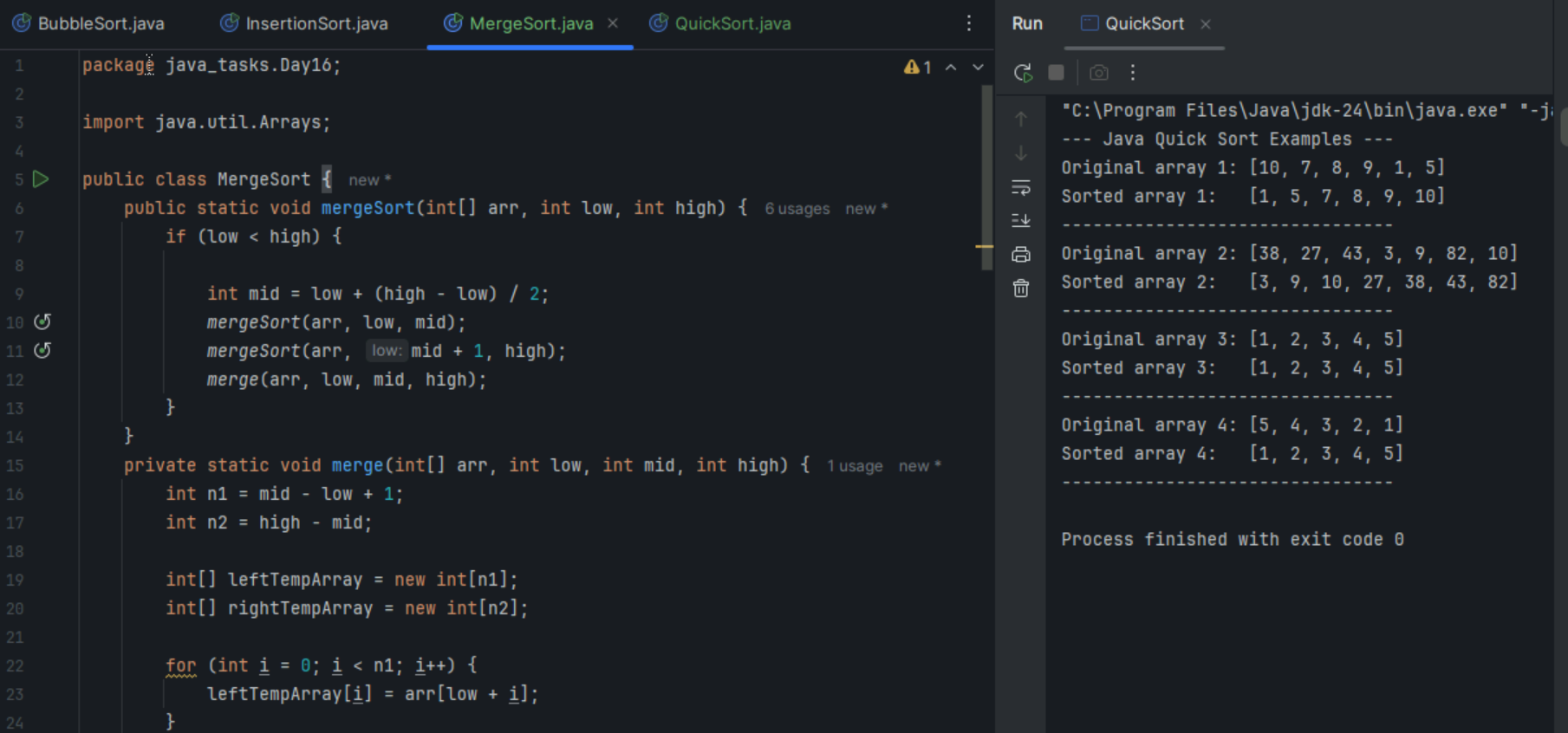
k = k + 1

END WHILE

END Algorithm

TSK 14:

code for Merge sort:



Task 15:

Algo fro quick sort:

If you have a big group of students, and you want to line them up by height.

We pick one student to be the "leader" (the pivot).

Then, you tell everyone else: "If you're shorter than the leader, go to the left side of the room. If you're taller, go to the right side."

Once everyone has moved, the leader is now in their perfectly correct spot in the line (everyone to their left is shorter, everyone to their right is taller).

Now you have two smaller, unorganized groups (left side and right side). You pick a new leader for the left group and repeat the process. You do the same for the right group.

You keep doing this until each group is so small it only has one person (which is sorted!). Then, because each "leader" was placed correctly, the whole line magically comes together sorted.

Algorithm:

Core Idea: Pick an element (pivot), partition the array around it, and recursively sort the sub-arrays.

quickSort(arr, low, high) - The Main Recursive Sorting Part:

Base Case: If low (start index) is greater than or equal to high (end index), it means the sub-array has 0 or 1 element, which is already sorted. So, stop.

Partition: Call a partition(arr, low, high) function. This function will pick a pivot, rearrange elements so that all elements smaller than the pivot are to its left, and all elements greater are to its right. It returns the final index of the pivot (pivotIndex).

Conquer (Recursively Sort Partitions):

Call quickSort(arr, low, pivotIndex - 1) to sort the sub-array to the left of the pivot.

Call quickSort(arr, pivotIndex + 1, high) to sort the sub-array to the right of the pivot.

partition(arr, low, high) - The Rearranging Part:

Choose Pivot: Select an element as the pivot. A common simple choice is the last element: pivot = arr[high]. (Better strategies exist, but this is easiest to explain).

Initialize Pointer for Smaller Elements: Set i = low - 1. This i will track the position where the next element smaller than or equal to the pivot should be placed.

Iterate and Swap: Loop j from low to high - 1 (i.e., through all elements before the pivot):

If arr[j] is less than or equal to pivot:

Increment i (i = i + 1).

Swap arr[i] and arr[j]. This moves the smaller element (arr[j]) into the "smaller than pivot" section.

Place Pivot: After the loop, swap arr[i + 1] with arr[high] (the original pivot element). This places the pivot in its correct sorted position.

Return Pivot's Final Index: Return i + 1.

Task 16:

Pseudo code for quick sort:

Algorithm 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 Algorithm

Algorithm Partition(arr, low, high)

pivot = arr[high]

i = low - 1

FOR j FROM low TO high - 1

IF arr[j] <= pivot THEN

i = i + 1

Swap(arr[i], arr[j]) // Swap arr[i] and arr[j]

END IF

END FOR

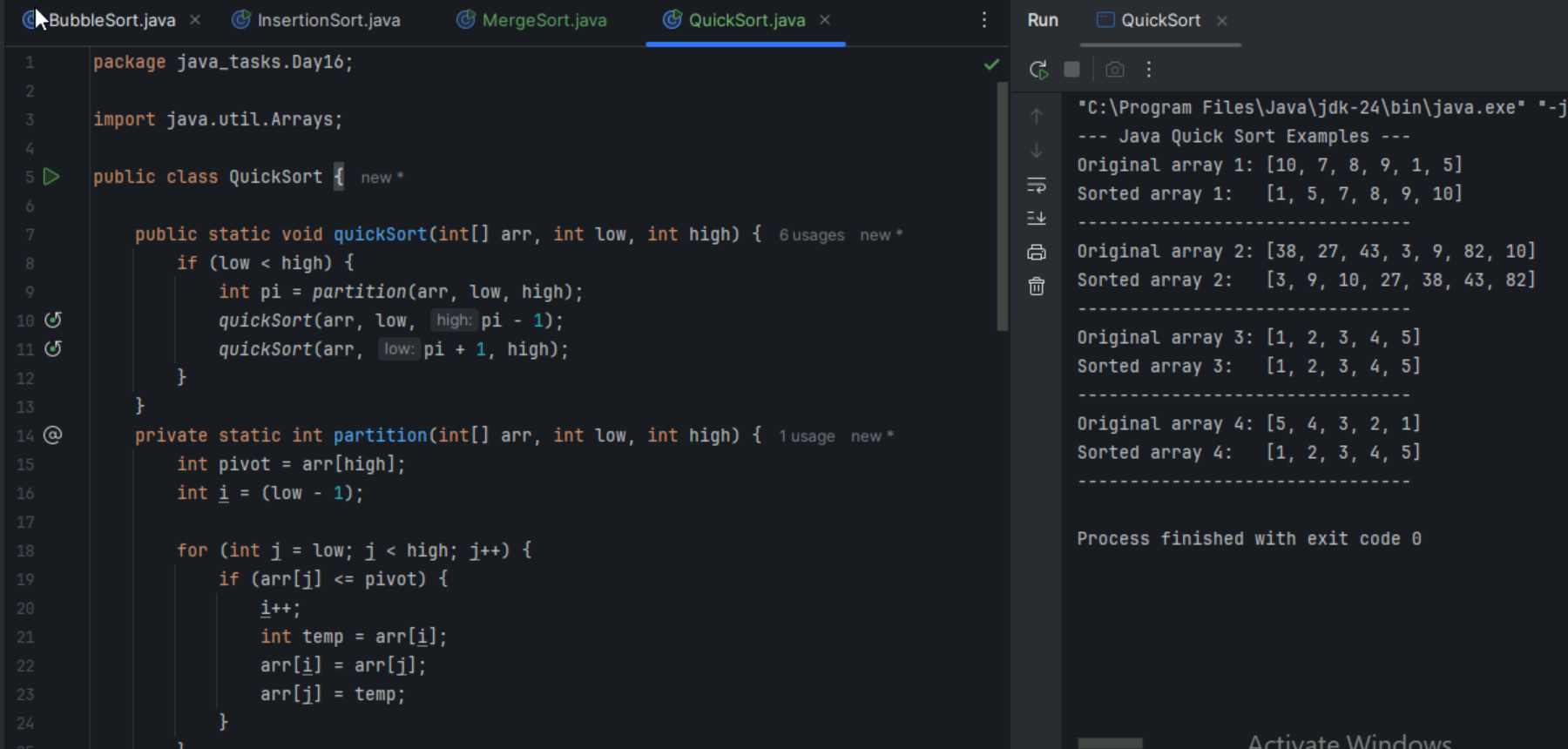
Swap(arr[i + 1], arr[high])

RETURN i + 1

END Algorithm

Task 17:

Code for Quick sort



Graph - A complex road network with one-way streets, two-way streets, roundabouts (cycles), and multiple routes between cities. Some cities might be isolated. and as  
Tree - A very specific road network where every city is connected to every other city, but there are absolutely no roundabouts or alternative routes between any two cities. If you remove just one road, the network becomes disconnected.

Add ons:

1.

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

All BSTs are Binary Trees, but Not All Binary Trees are BSTs.

-->A Binary Tree is like a family tree without any specific rules about who goes where, as long as each person has at most two children.

-->A Binary Search Tree (BST) is like a highly organized library where books are shelved based on their title (or a numerical value). You know exactly where to look for a specific book.

1. Binary Tree (BT)

Structure: "Just Two Kids, Anywhere"

At its most basic, a Binary Tree is a tree where each "node" ,a circle with data in it, can have at most two children: a left child and a right child. That's pretty much the only rule for its structure.

There's no rule about what values go on the left or right side. You could have a node with 10, its left child could be 100, and its right child could be 5. It doesn't matter for a basic Binary Tree.

It can be any shape – tall and skinny, short and fat, completely lop-sided, or perfectly balanced. It just has the "at most two children" rule.

There are Duplicates, we can have multiple nodes with the same value anywhere in the tree.

we can insert data based on various strategies (e.g., fill level by level like a "complete binary tree," or just add to the next available spot). It's not about the value of the data itself for placing it.

If you want to find a specific piece of data, you have to literally search the entire tree (or at least a large portion of it). You don't know if the data is on the left or right, so you might have to check every branch. This is slow, especially for large trees (worst case is O(N), meaning you might check every node).

More complex, as you need to remove the node while keeping the "at most two children" rule intact, but there's no value-based reordering.

Traversal (Visiting All Data): You can visit all nodes in different orders (like "inorder," "preorder," "postorder"), but the order in which you visit them isn't inherently sorted by value.

2. Binary Search Tree (BST)

Two Kids, But They Have Rules like structure.

A BST is a special kind of Binary Tree that adds a very important rule about the values in its nodes.

For every single node in a BST:

All values in its left child and entire left subtree are LESS THAN the node's own value.

All values in its right child and entire right subtree are GREATER THAN the node's own value.

Shape is Like a regular Binary Tree, a BST can still be tall and skinny or balanced. However, if you insert data in a specific order (like already sorted data), it can become very "unbalanced" and look like a linked list, which hurts its performance.

Operations: "Go Straight to the Right Shelf!"

To insert a new value:

Start at the top (the "root").

If the new value is less than the current node's value, go left.

If it's greater, go right.

Repeat until you find an empty spot where the new value fits according to the rule. This makes insertion much faster on average (O(log N)).

This is where BSTs shine - To find a value:

Start at the root.

If the target value is less than the current node's value, you know it can only be in the left subtree, so you go left.

If the target value is greater, you go right.

You keep doing this, quickly cutting down the search area by half with each step. This makes searching very fast on average (O(log N)).

Also follows the rules. It's more complex than insertion because you have to maintain the BST property and handle cases where the node has 0, 1, or 2 children (often by replacing with its "inorder successor" or "predecessor").

Traversal (Visiting All Data):

An "inorder traversal" (left child, then node, then right child) of a BST will always give you all the data back in sorted order. This is a powerful operational advantage.

The fundamental difference lies in the ordering property that the BST imposes on its structure, which then directly enables its highly efficient search, insertion, and deletion operations.

Can you explain diff between structure and operation of Binary tree and BST.

|  |  |  |
| --- | --- | --- |
| Feature | Binary Tree (BT) | Binary Search Tree (BST) |
| Structural Rule | Each node has at most two children. | BT Rule + Ordering Property: Left < Root < Right. |
| Node Value Order | No inherent order (chaotic). | Values are strictly ordered. |
| Duplicates | Generally allowed anywhere. | Usually not allowed, or specific rules apply. |
| Primary Purpose | Representing hierarchies, expressions, heaps. | Efficient Searching, Insertion, Deletion. |
| Search Operation | Can be very slow (O(N) worst case - must scan). | Very fast (O(log N) average case - cuts search space). |
| Insert Operation | Simple placement, no value-based rule. | Guided by value, moves down specific path (faster). |
| Inorder Traversal | Arbitrary order of values. | Produces sorted data. |

2.

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

Binary Search is "best" on a sorted array because it's like having a magical ability to instantly jump to the middle of your problem and then intelligently throw away half of it, over and over again. Linear Search, even on a sorted array, is still stuck plodding along one by one.