Selection Sort

Task 1 : Write an algorithm / steps for selection sort.

Solution :

1. Start with the first element in the array (index 0).

2. Find the smallest element in the unsorted part of the array.

3. Swap this smallest element with the first element of the unsorted part.

4. Move the boundary of the sorted part one step to the right.

5. Repeat steps 2-4 for the rest of the array until the whole array is sorted.

6. End.

Example (for array [5, 2, 9, 1, 5]):

- First pass: Find smallest (1), swap with 5 → [1, 2, 9, 5, 5]

- Second pass: Find next smallest (2), already in place → [1, 2, 9, 5, 5]

- Third pass: Find next smallest (5), swap with 9 → [1, 2, 5, 9, 5]

- Fourth pass: Find next smallest (5), swap with 9 → [1, 2, 5, 5, 9]

- Now sorted!

Task 2: Write a pseudo code for the selection sort.

Solution :

SelectionSort(A, n):

for i from 0 to n-2 do

minIndex = i

for j from i+1 to n-1 do

if A[j] < A[minIndex] then

minIndex = j

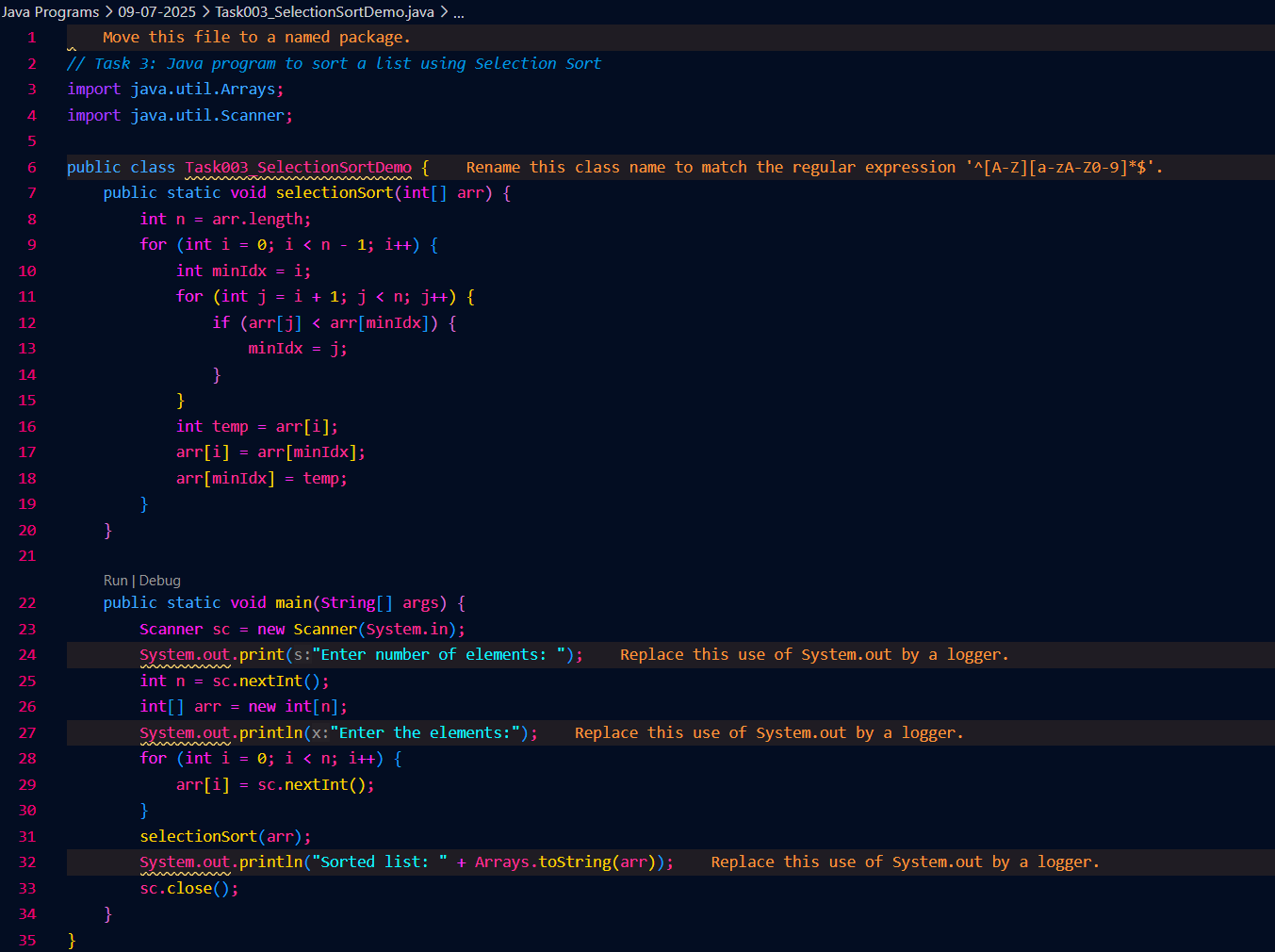
end for

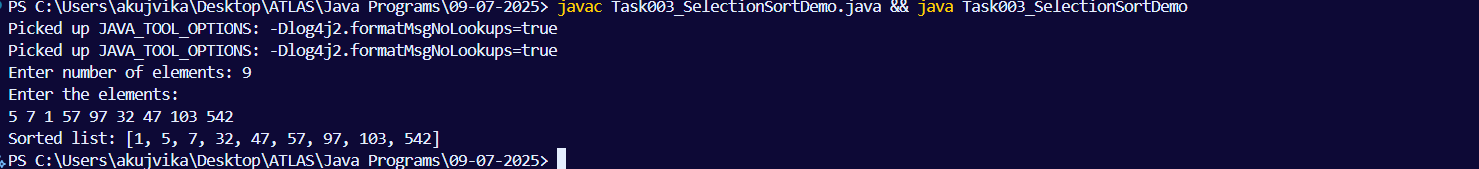
swap A[i] and A[minIndex]

end for

end

Task 3: Wap to make sure your list is sorted using selection sort.

Solution : 



Bubble Sort

Task 4: Write algorithm for the Bubble sort.

Solution :

1. Start at the beginning of the list.

2. Compare each pair of adjacent elements.

3. If the left element is bigger than the right, swap them.

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

5. After each pass, the largest element will have "bubbled up" to its correct position at the end.

6. Repeat steps 1-5 for the remaining unsorted part of the list (ignore the last sorted elements).

7. Keep repeating until no swaps are needed (the list is sorted).

8. End.

Example (for array [5, 2, 9, 1, 5]):

- First pass: [2, 5, 1, 5, 9]

- Second pass: [2, 1, 5, 5, 9]

- Third pass: [1, 2, 5, 5, 9]

- Now sorted!

Task 5: Write pseudo code for the bubble sort

Solution : BubbleSort(A, n):

for i from 0 to n-2 do

for j from 0 to n-2-i do

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

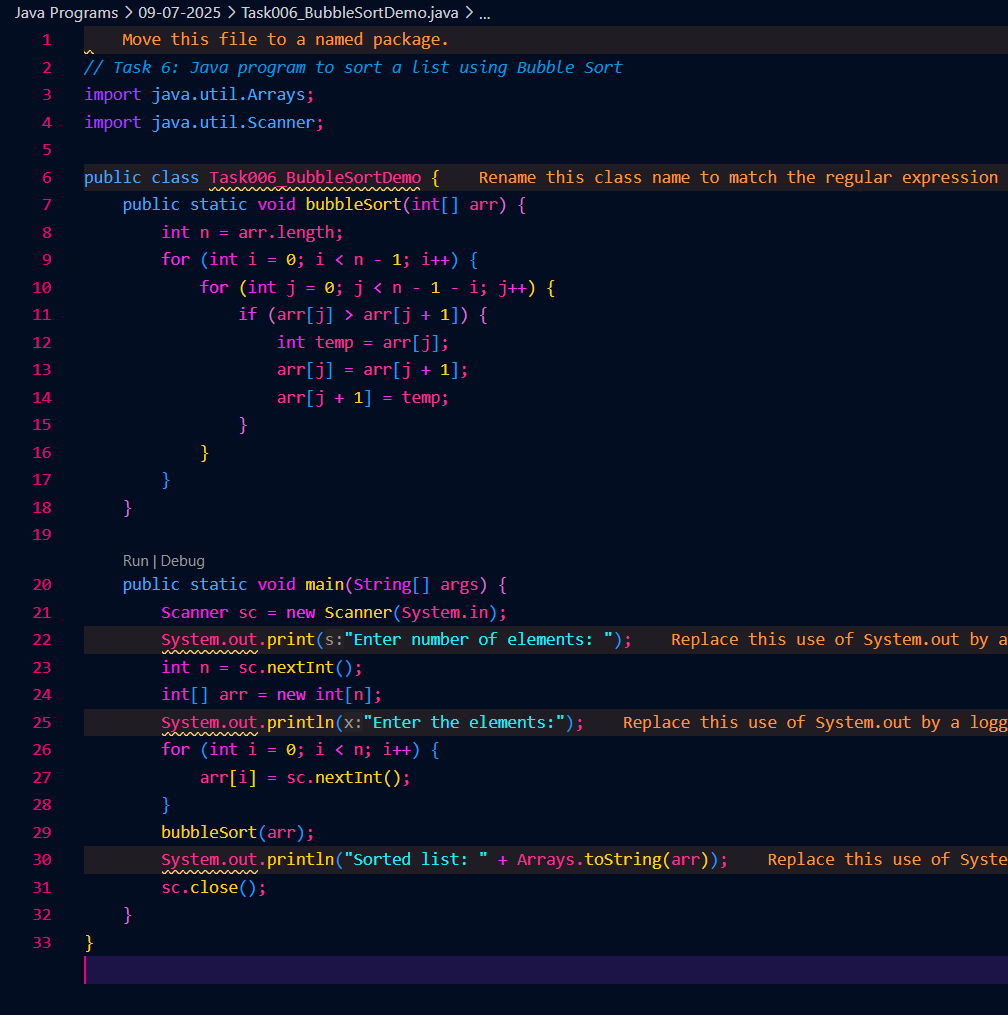
swap A[j] and A[j+1]

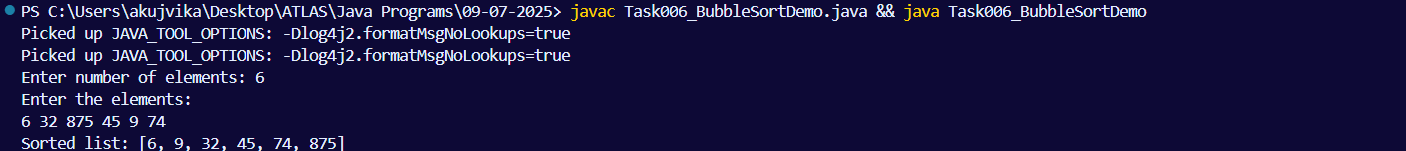
end for

end for

end

Task 6: Wap to make sure your list is sorted using Bubble sort.

Solution : 



INSERTION SORT

Task 7: Write algorithm for the Insertion sort.

Solution :

1. Start with the second element (index 1) in the list.

2. Compare it with the elements before it.

3. Move all elements that are bigger one position to the right.

4. Insert the current element into its correct position.

5. Move to the next element and repeat steps 2-4 until the end of the list.

6. End.

Example (for array [5, 2, 9, 1, 5]):

- First pass: [2, 5, 9, 1, 5]

- Second pass: [2, 5, 9, 1, 5] (no change)

- Third pass: [1, 2, 5, 9, 5]

- Fourth pass: [1, 2, 5, 5, 9]

- Now sorted!

Task 8: Write pseudo code for the Insertion sort

Solution :

InsertionSort(A, n):

for i from 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

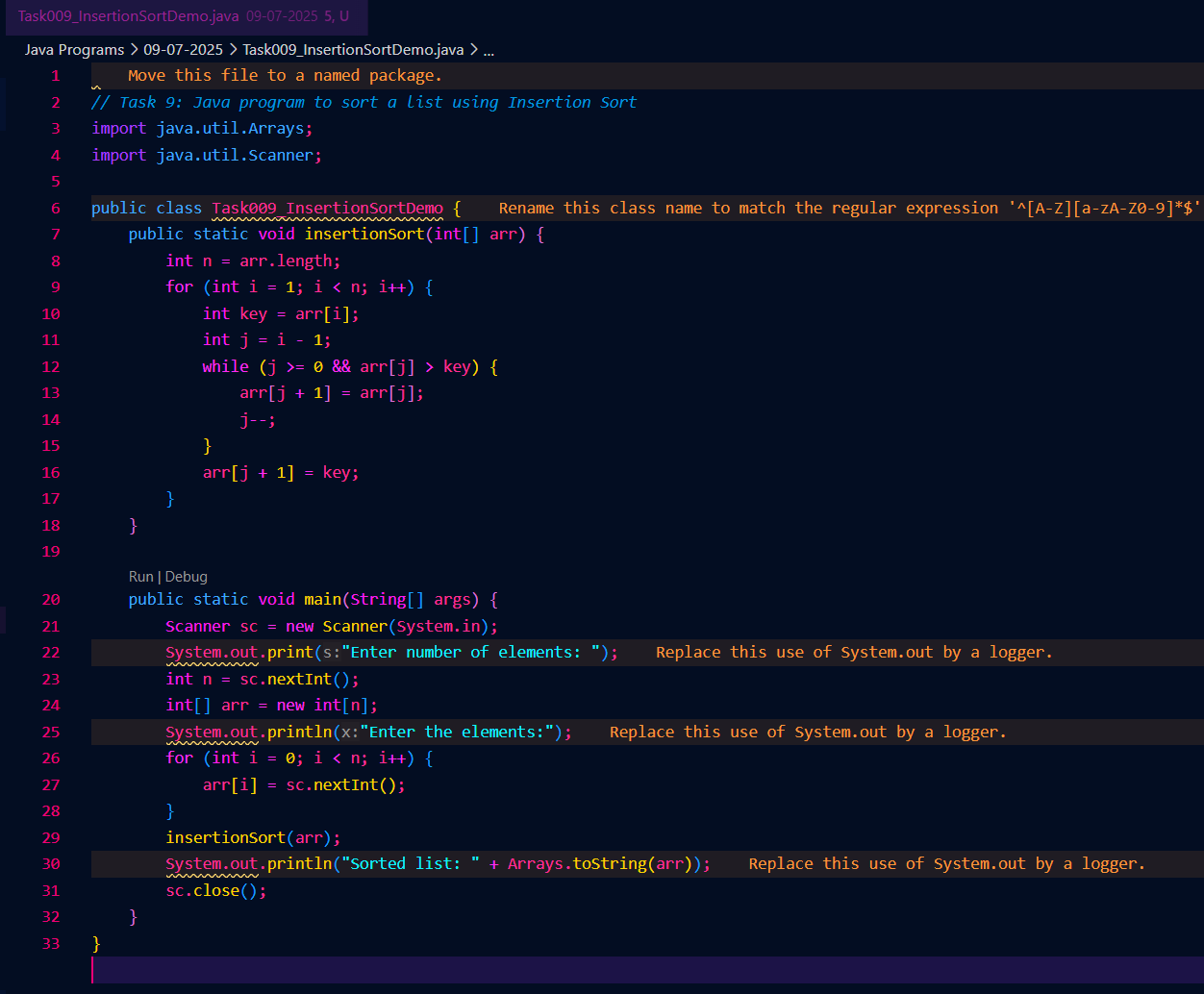
A[j + 1] = key

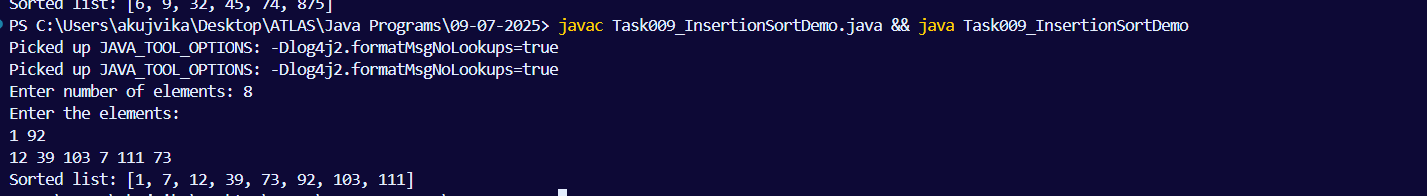
end for

end

Task 9: Wap to make sure your list is sorted using Insertion sort.

Solution :





Task 10: What are the advantages and disadvantages of Bubble sort Algo? List them

Solution :

Advantages:

1. Simple to understand and easy to implement.

2. Does not require extra memory (works in-place).

3. Good for small or nearly sorted lists.

4. Can detect if the list is already sorted (with an optimized version).

Disadvantages:

1. Very slow for large lists (O(n^2) time complexity).

2. Not suitable for real-world, large datasets.

3. Performs many unnecessary comparisons and swaps.

4. Less efficient than other sorting algorithms like quick sort or merge sort.

Task 11: Algorithm for merge sort

Solution:

1. If the list has 1 or 0 elements, it is already sorted.

2. Divide the list into two halves.

3. Recursively sort each half using merge sort.

4. Merge the two sorted halves into one sorted list.

5. End.

Task 12 : Pseudo code for merge sort.

Solution:

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

Merge(A, left, mid, right):

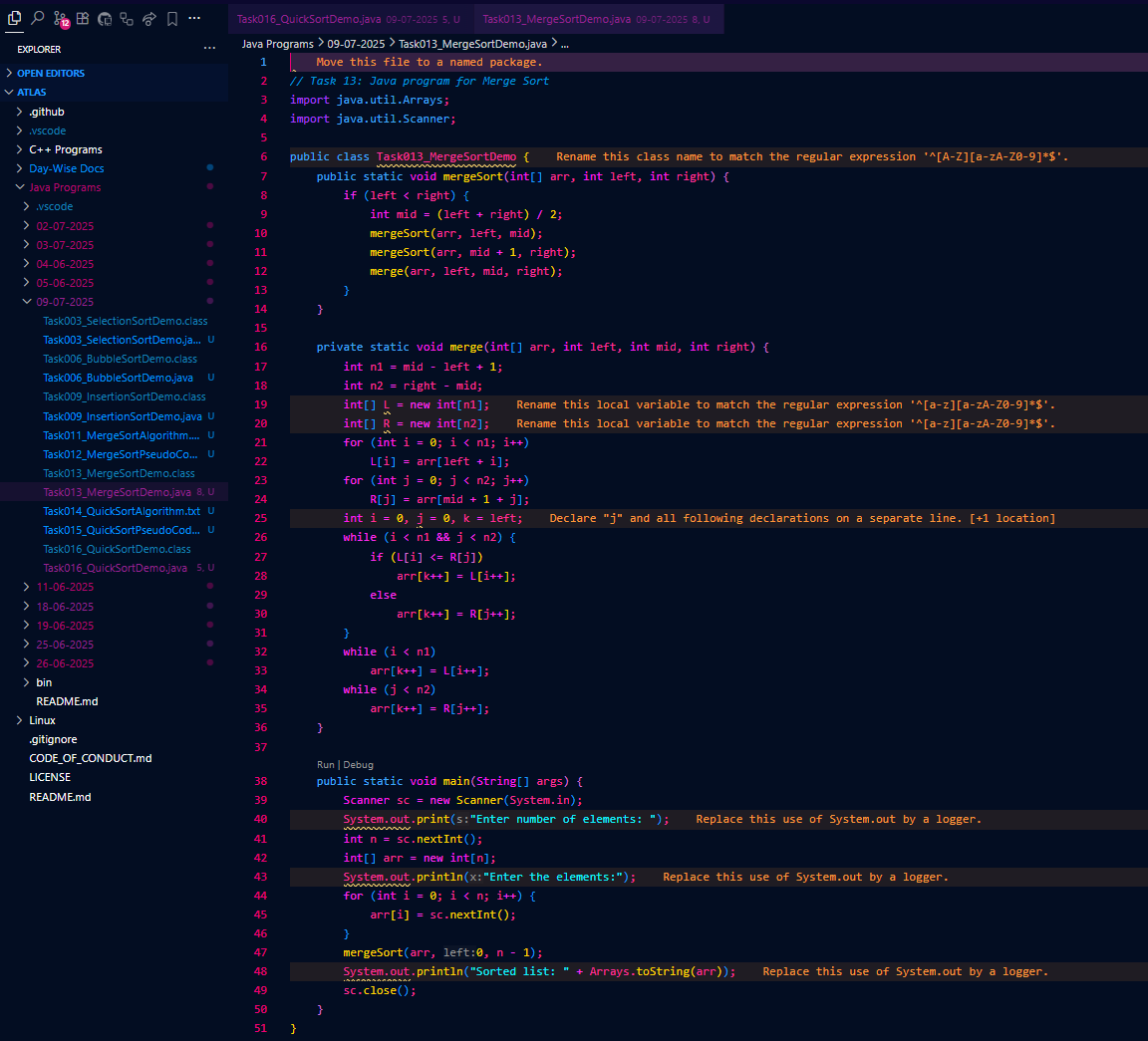
create temp arrays L and R

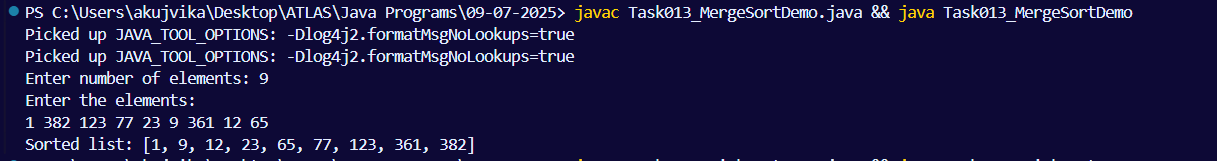
copy data to L and R

merge L and R back into A[left..right]

end

Task 13 : Write a program for Merge sort

Solution: 



Task 14: Algo for quick sort

Solution :

1. If the list has 1 or 0 elements, it is already sorted.

2. Choose a pivot element from the list.

3. Partition the list so that:

- All elements less than the pivot come before it.

- All elements greater than the pivot come after it.

4. Recursively apply quick sort to the left and right parts.

5. End.

Task 15: Pseudo code for quick sort

Solution :

QuickSort(A, low, high):

if low < high then

pi = Partition(A, low, high)

QuickSort(A, low, pi-1)

QuickSort(A, pi+1, high)

end if

end

Partition(A, low, high):

pivot = A[high]

i = low - 1

for j from low to high-1 do

if A[j] < pivot then

i = i + 1

swap A[i] and A[j]

end for

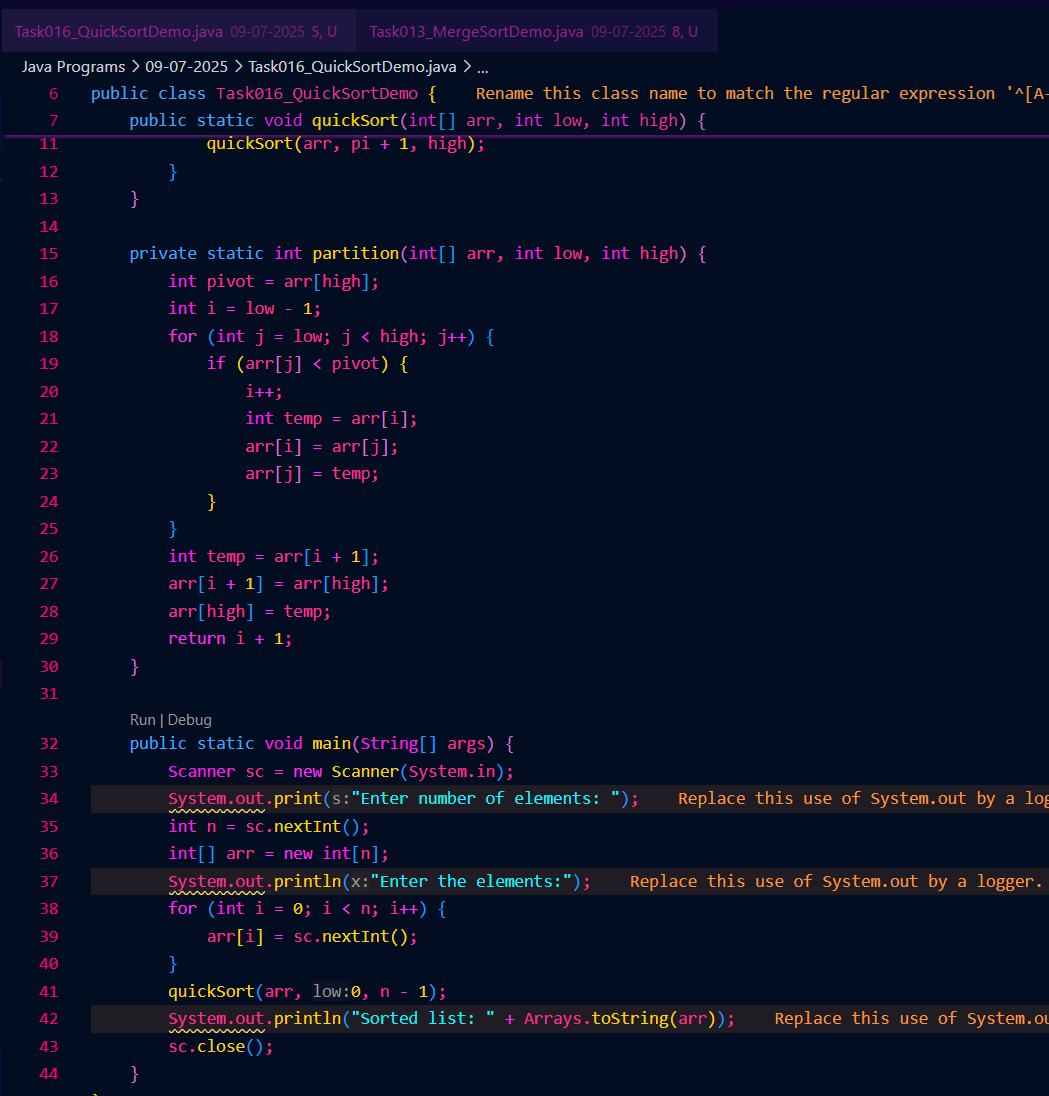
swap A[i+1] and A[high]

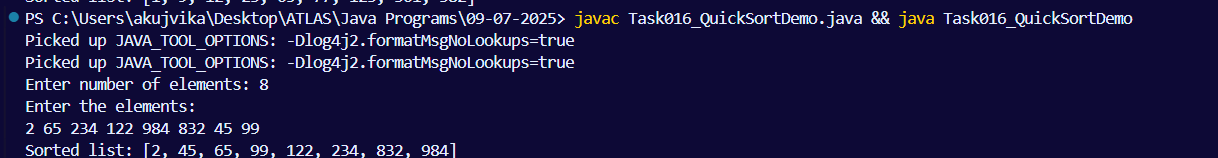
return i+1

end

Task 16: Code for Quick sort

Solution :





Task 17: Which one is better merge or bubble in terms of time complexity. Note: industrial applications like huge data bases … prefer merge sort

Merge : O(n log n ) & Bubble : O (n square)

Solution :

1. Merge sort is much better than bubble sort in terms of time complexity, especially for large data sets.

* Merge Sort: O(n log n) — much faster for big lists.
* Bubble Sort: O(n²) — very slow for big lists.

1. Merge sort splits the list and sorts parts efficiently, so it works quickly even for huge databases.
2. Bubble sort compares every pair again and again, which takes a lot more time as the list grows.
3. Merge sort (or other O(n log n) algorithms) is always preferred for large data, like in databases, because it is much faster and more efficient than bubble sort.

Task 18: (Leet code question) Find the time complexity of the given merge operation between two sorted array. Note: O(n + m) as we traverse both arrays once, n and m are size of both the arrays

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

Solution : The time complexity of the merge operation between two sorted arrays (as in the LeetCode problem “Merge Sorted Array”) is O(n + m).

Explanation:

You have two sorted arrays of sizes n and m.

You look at each element of both arrays exactly once to merge them into a single sorted array.

So, the total number of steps is proportional to n + m.

Why O(n + m)? :

You never go back or repeat elements.

You only move forward through both arrays, comparing and copying elements.

Summary:

The merge operation is very efficient—linear in the total number of elements.

Time Complexity: O(n + m)

Task 19: This code is going overflow of stack.. Can you plz help me fix it guys.. ☹️

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

public class RecLoop {

public int calc(int n) {

if (n == 0) return 0;

return n + calc(n);

}

Solution :

The above code causes a stack overflow because the recursive call never reduces the value of n, so it never reaches the base case (n == 0). This creates infinite recursion.

How to fix:

We should decrease n in each recursive call, so it eventually reaches 0

Now, calc(5) will compute 5 + 4 + 3 + 2 + 1 + 0 and stop, preventing stack overflow.

Task 20: binary search function can we use in unsorted list? Will it give you correct index no?

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;

}

}

Solution:

No, you cannot use binary search on an unsorted list and expect correct results.

Why?

- Binary search only works on sorted arrays/lists.

- It relies on the order to decide which half to search next.

- If the list is not sorted, the algorithm may skip over the correct value or return the wrong index.

Example:

Suppose your array is `[7, 2, 9, 4, 1]` (unsorted) and you search for `4`:

- Binary search might not find it, or might return the wrong index, because it assumes the array is sorted.

Summary:

- search on unsorted list:\*\* ❌ Not correct, do not use.

- Always sort the list first\*\* if you want to use binary search and get the correct index.

Add Ons:

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

A) Binary Tree:

Every node can have at most two children (left and right).

There are no rules about how the values are arranged.

The left child can be bigger or smaller than the parent—no restrictions.

Binary Search Tree (BST):

It is a special kind of binary tree.

For every node:

* All values in the left subtree are less than the node.
* All values in the right subtree are greater than the node.

This rule makes searching, inserting, and deleting fast.

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

A) Structure:

* Binary Tree: No order; any value can go anywhere.
* BST: Ordered; left < parent < right for every node.

Operation:

* Binary Tree:

Searching for a value means you may have to check every node (slow).

Inserting is just finding the next empty spot.

* BST:

Searching is fast: you can skip half the tree at each step (like a phone book).

Inserting keeps the order, so you always know where to put the new value.

3. In sorted array , why do you think binary search tree is better than linear search. Can you explain?

A)

Linear Search:

* You check each element one by one, from start to end.
* If the number is at the end, you have to look at every element.
* Time: Slow for big lists (O(n) time).

Binary Search (or BST search):

* You look at the middle element first.
* If your number is smaller, you only look at the left half; if bigger, only the right half.
* You keep cutting the search space in half.
* Time: Much faster for big lists (O(log n) time).

4. Difference between static and dynamic arrays. Please list them.

A)

Static Array :

* Fixed Size: Size is set when the array is created and cannot change.
* Memory Allocation: Allocated at compile time (in languages like C/C++) or when created (Java).
* Faster Access: Accessing elements is very fast.
* No Resizing: Cannot add or remove elements after creation.
* Example: int[] arr = new int[5]; // Always has 5 elements

Dynamic Array :

* Flexible Size: Can grow or shrink as needed.
* Memory Allocation: Allocated at runtime; can change size as the program runs.
* Slower for Some Operations: May need to copy elements to a new array when resizing.
* Can Add/Remove Elements: Easily add or remove elements.
* Example: ArrayList<Integer> list = new ArrayList<>(); // Can add/remove elements list.add(10);