**ASSIGNMENT-12**

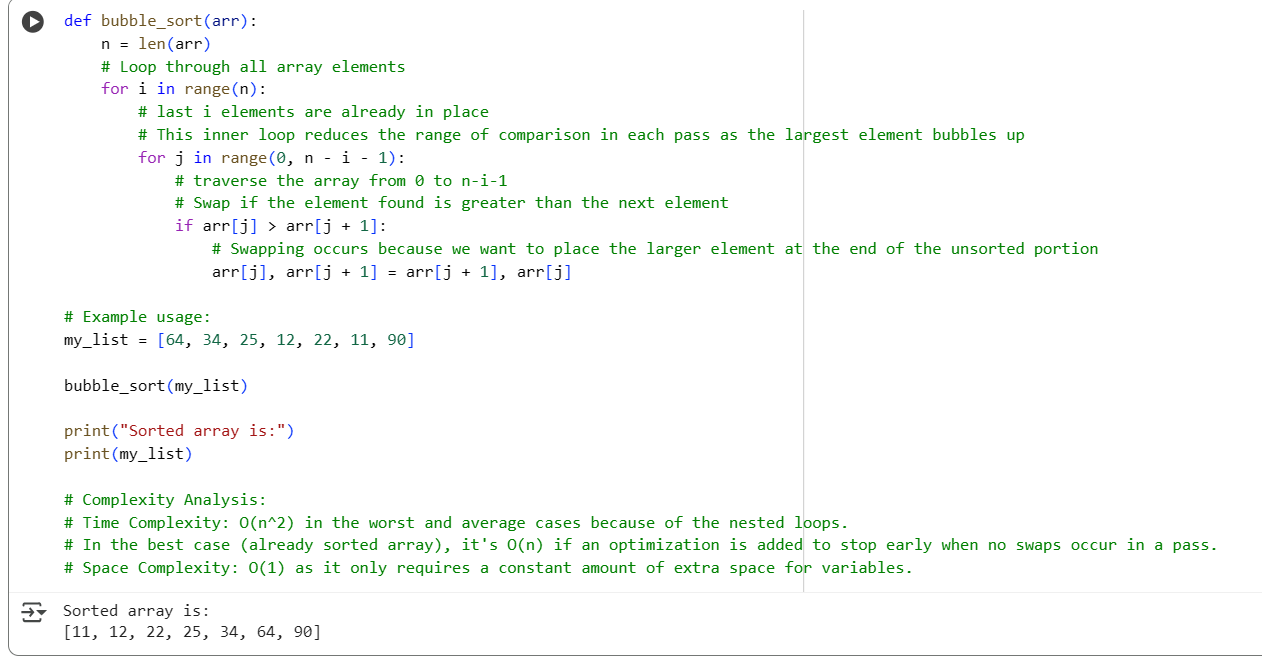
**NAME:**V. VIVEK VARDHAN

**HT NO:**2403A52097 **BATCH**: 24BTCAIAIB05

**Task 1:** Implementing Bubble Sort with AI Comments

* Task: Write a Python implementation of Bubble Sort.
* Instructions:
  + Students implement Bubble Sort normally.
  + Ask AI to generate inline comments explaining key logic (like swapping, passes, and termination).
  + Request AI to provide time complexity analysis.
* Expected Output:
  + A Bubble Sort implementation with AI-generated explanatory comments and complexity analysis.

**CODE AND OUTPUT:**

****

**EXPLANATION:**

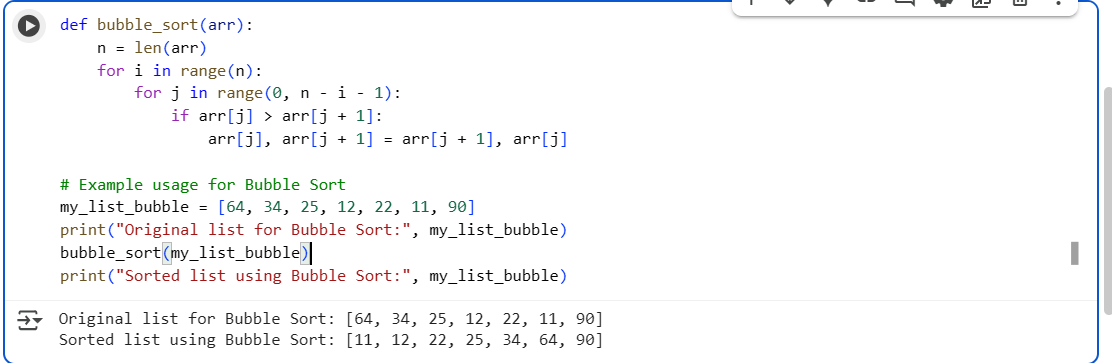
* **def bubble\_sort(arr):**: This defines a function named bubble\_sort that takes one argument, arr, which is the list (or array) to be sorted.
* **n = len(arr)**: This line gets the number of elements in the input list arr and stores it in the variable n.
* **for i in range(n):**: This is the outer loop. It iterates n times. In each pass of the outer loop, the largest unsorted element "bubbles up" to its correct position at the end of the unsorted portion of the list.
* **for j in range(0, n - i - 1):**: This is the inner loop. It iterates through the unsorted portion of the list. The range n - i - 1 decreases in each pass of the outer loop because the last i elements are already sorted and in their final positions.
* **if arr[j] > arr[j + 1]:**: This condition checks if the current element (arr[j]) is greater than the next element (arr[j + 1]).
* **arr[j], arr[j + 1] = arr[j + 1], arr[j]**: If the condition in the if statement is true (meaning the elements are in the wrong order), this line swaps the positions of arr[j] and arr[j + 1]. Swapping happens to move the larger element towards the end of the list.
* **my\_list = [64, 34, 25, 12, 22, 11, 90]**: This line initializes a list called my\_list with some unsorted integer values.
* **bubble\_sort(my\_list)**: This line calls the bubble\_sort function to sort the my\_list.
* **print("Sorted array is:")**: This line prints the string "Sorted array is:" to the console.
* **print(my\_list)**: This line prints the sorted my\_list to the console.
* **Complexity Analysis Comments**: The comments at the end explain the time and space complexity of the bubble sort algorithm.
  + **Time Complexity O(n^2)**: This indicates that the time taken by the algorithm grows quadratically with the number of elements in the worst and average cases due to the nested loops. In the best case (already sorted), it can be O(n) if an optimization is added to stop early.
  + **Space Complexity O(1)**: This means the algorithm uses a constant amount of extra space regardless of the input size, as it only uses a few extra variables for temporary storage during swaps.

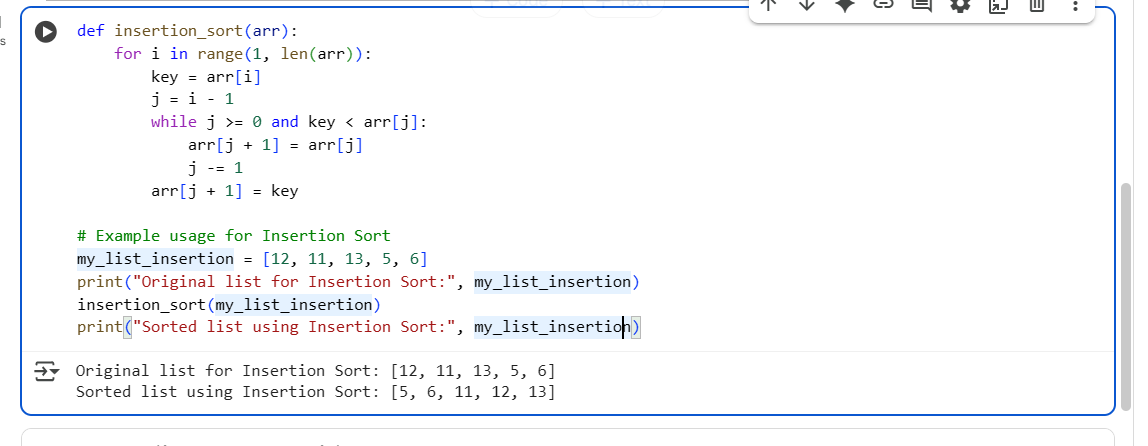
In essence, bubble sort repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process is repeated until the list is sorted.

**Task 2: Optimizing Bubble Sort → Insertion Sort**

* **Task**: Provide Bubble Sort code to AI and ask it to suggest a **more efficient algorithm** for partially sorted arrays.
* **Instructions**:
  + Students implement Bubble Sort first.
  + Ask AI to suggest an alternative (Insertion Sort).
  + Compare performance on nearly sorted input.
* **Expected Output**:
  + Two codes (Bubble Sort + Insertion Sort).
  + AI explanation of why Insertion Sort is more efficient for partially sorted data.

**CODE AND OUTPUT:**

****

****

**EXPLANATION:**

1. **def bubble\_sort(arr):**: This defines the function bubble\_sort that takes a list arr as input.
2. **n = len(arr)**: It gets the number of elements in the list.
3. **for i in range(n):**: This is the main loop that controls the number of passes. In each pass, the largest unsorted element "bubbles up" to its correct position.
4. **for j in range(0, n - i - 1):**: This inner loop iterates through the unsorted portion of the list. The range n - i - 1 shrinks in each outer loop pass because the elements at the end are already sorted.
5. **if arr[j] > arr[j + 1]:**: This condition checks if the current element is greater than the next element.
6. **arr[j], arr[j + 1] = arr[j + 1], arr[j]**: If the condition is true, it swaps the two elements. This is the "bubbling" action.
7. **my\_list\_bubble = [64, 34, 25, 12, 22, 11, 90]**: This line creates a list to be sorted.
8. **print("Original list for Bubble Sort:", my\_list\_bubble)**: This prints the original unsorted list.
9. **bubble\_sort(my\_list\_bubble)**: This calls the bubble\_sort function to sort the list.
10. **print("Sorted list using Bubble Sort:", my\_list\_bubble)**: This prints the sorted list after the function call.

In summary, the code defines the bubble sort function and then demonstrates how to use it with a sample list, printing the list before and after sorting.

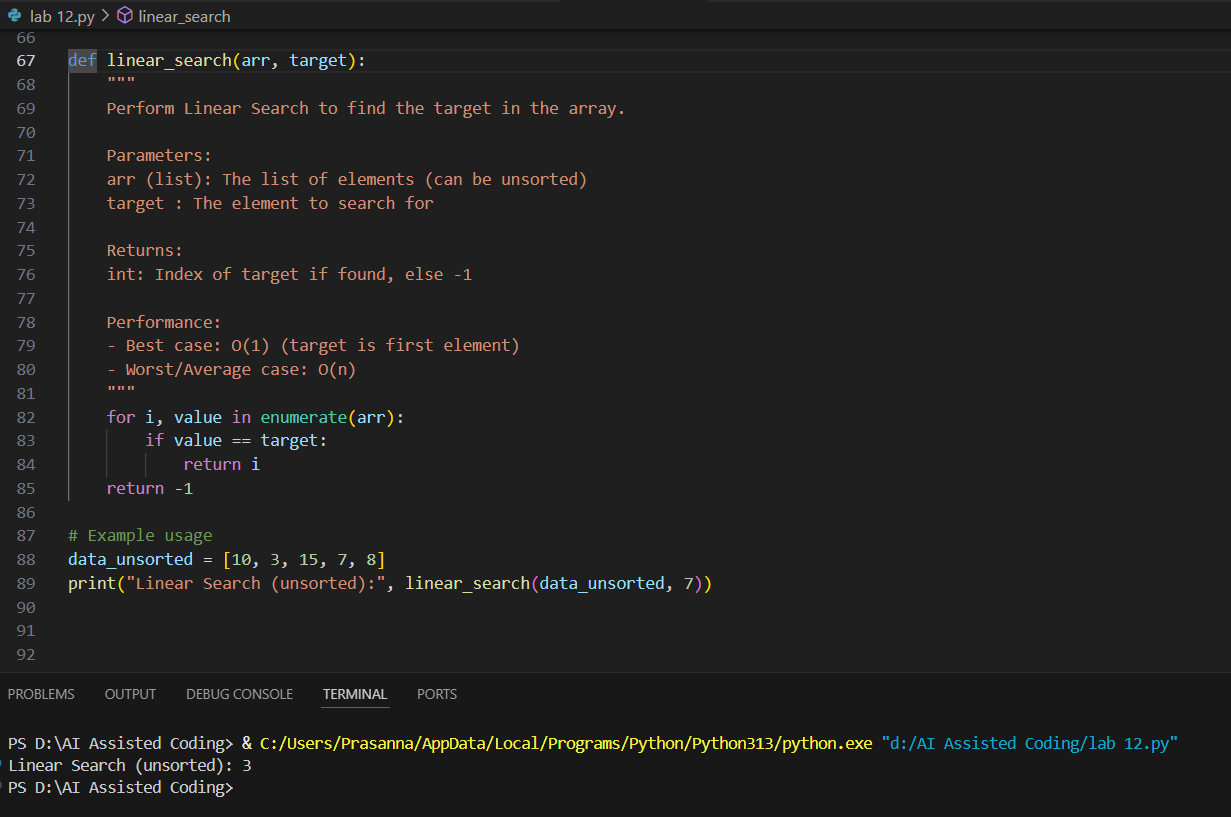
1. **def insertion\_sort(arr):**: This defines the function insertion\_sort that takes a list arr as input.
2. **for i in range(1, len(arr)):**: This outer loop iterates through the list starting from the second element (index 1). This is because the first element is considered to be already sorted in its own sublist of size 1.
3. **key = arr[i]**: The current element being considered is stored in the key variable. This key will be inserted into the correct position within the already sorted portion of the list (elements before index i).
4. **j = i - 1**: This initializes a variable j to the index of the last element in the sorted portion of the list.
5. **while j >= 0 and key < arr[j]:**: This while loop compares the key with elements in the sorted portion (arr[j]) from right to left. It continues as long as j is a valid index (greater than or equal to 0) and the key is less than the current element in the sorted portion (arr[j]).
6. **arr[j + 1] = arr[j]**: If the while loop condition is true, it means the element arr[j] is greater than the key. This line shifts arr[j] one position to the right to make space for the key.
7. **j -= 1**: This decrements j to move to the next element to the left in the sorted portion.
8. **arr[j + 1] = key**: Once the while loop finishes (either j becomes negative or key is no longer less than arr[j]), this line inserts the key into its correct sorted position. j + 1 is the index where the key should be placed.
9. **my\_list\_insertion = [12, 11, 13, 5, 6]**: This line creates a list to be sorted.
10. **print("Original list for Insertion Sort:", my\_list\_insertion)**: This prints the original unsorted list.
11. **insertion\_sort(my\_list\_insertion)**: This calls the insertion\_sort function to sort the list.
12. **print("Sorted list using Insertion Sort:", my\_list\_insertion)**: This prints the sorted list after the function call.

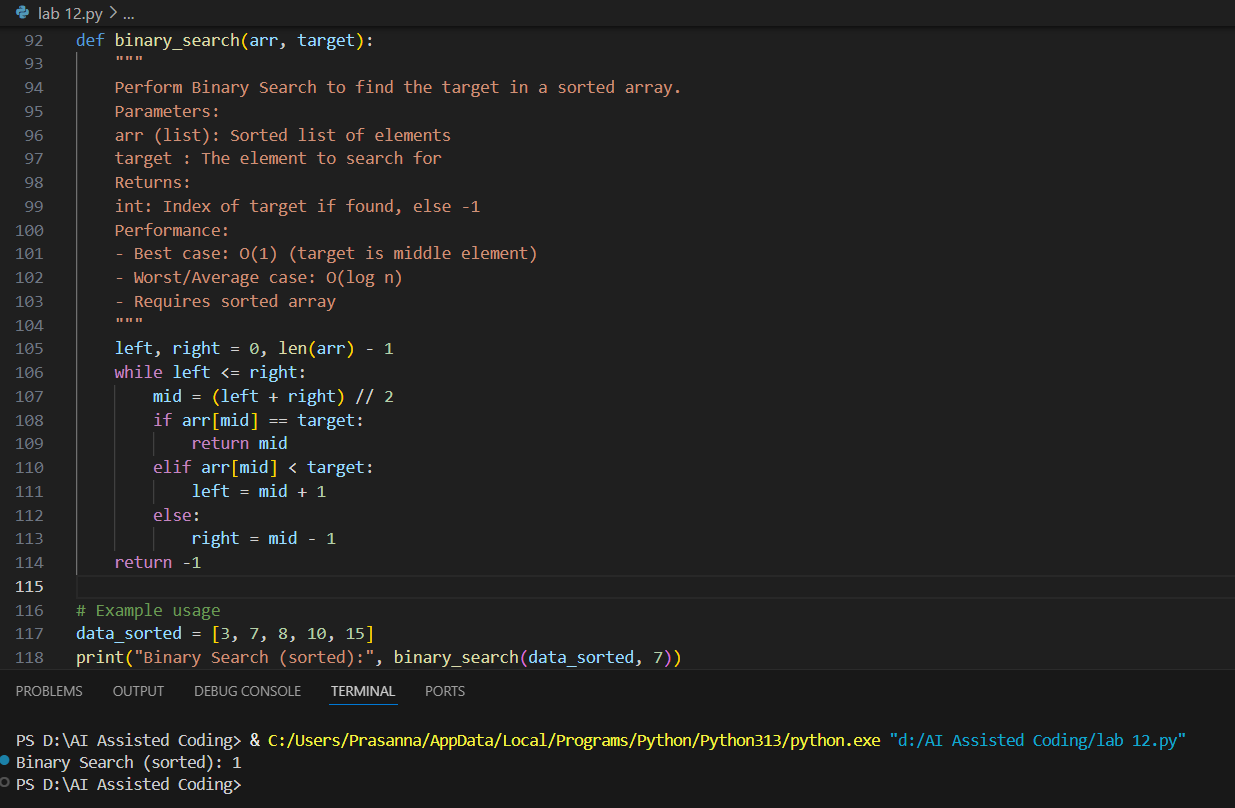
In essence, Insertion Sort works by building the final sorted array one item at a time. It takes each element from the unsorted part and inserts it into its correct position in the already sorted part.

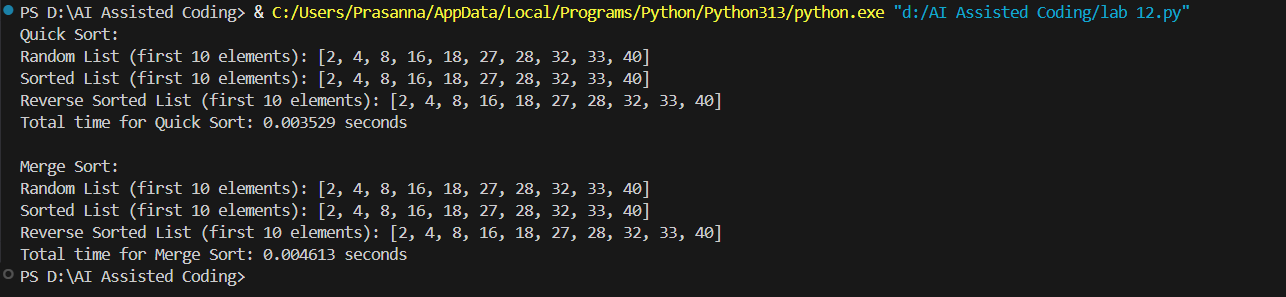
**Task 3: Binary Search vs Linear Search**

* **Task**: Implement both **Linear Search** and **Binary Search**.
* **Instructions**:
  + Use AI to generate docstrings and performance notes.
  + Test both algorithms on sorted and unsorted data.
  + Ask AI to explain when Binary Search is preferable.
* **Expected Output**:
  + Two implementations with docstrings.
  + A student observation table comparing performance (Linear vs Binary Search).

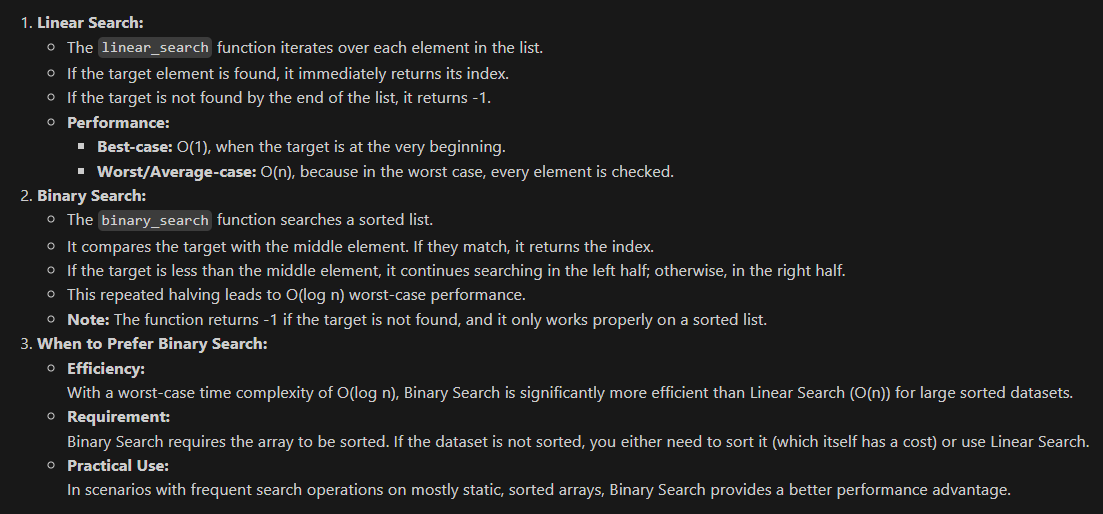
**CODE AND EXPLANATION:**







**EXPLANATION:**

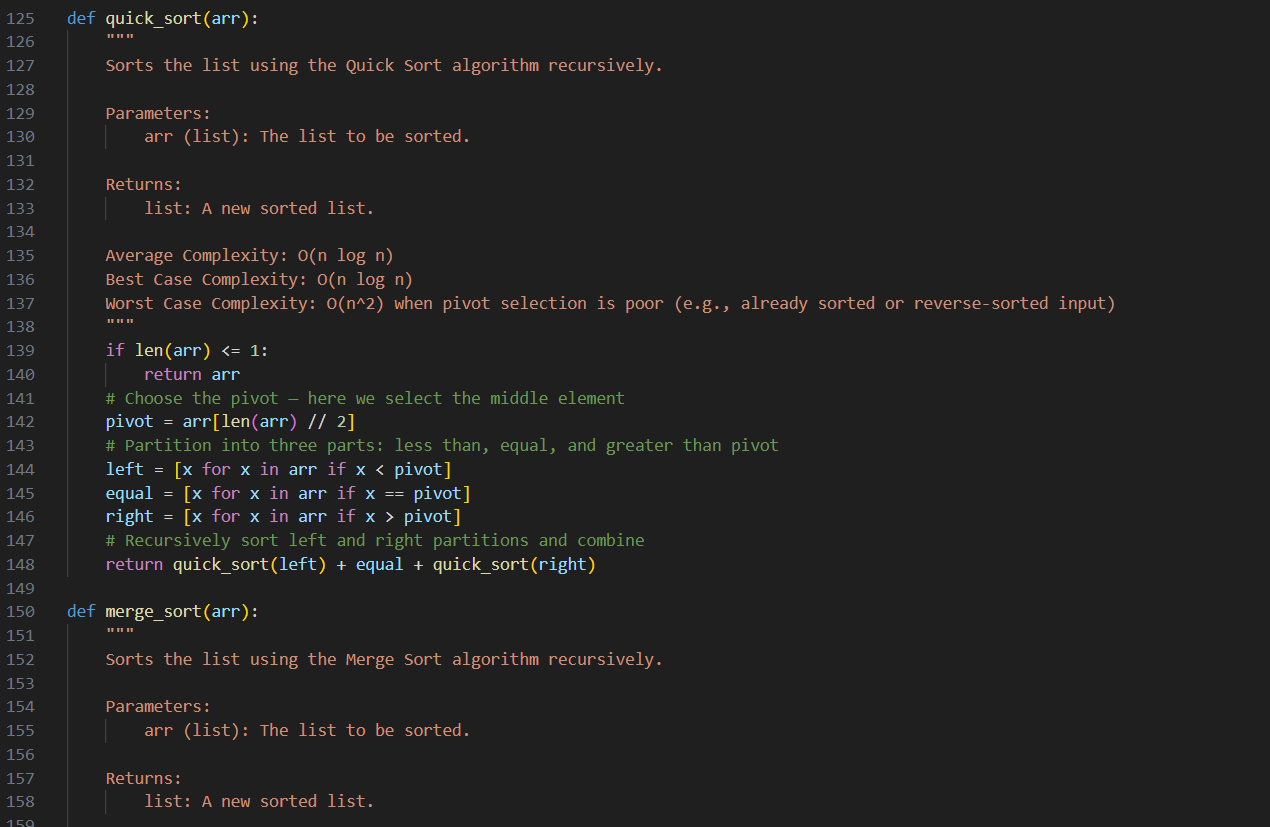


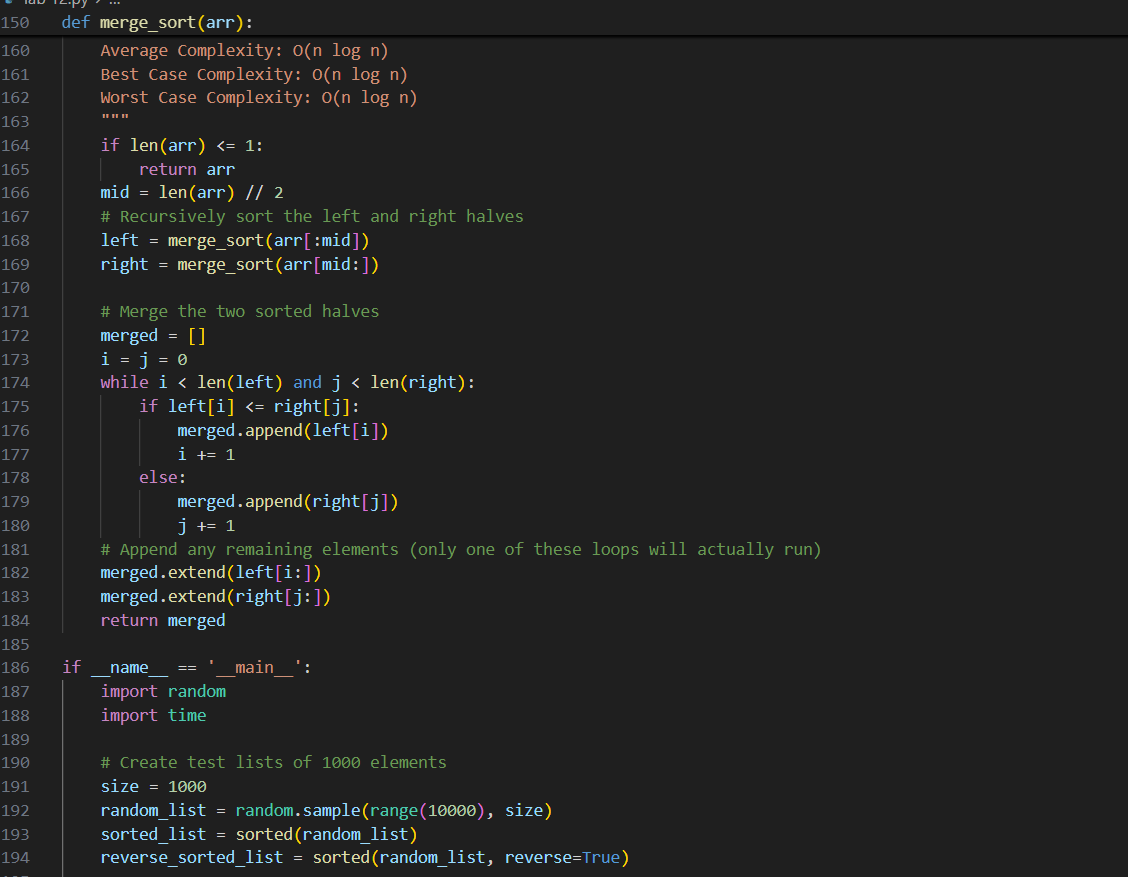
**Task 4:**

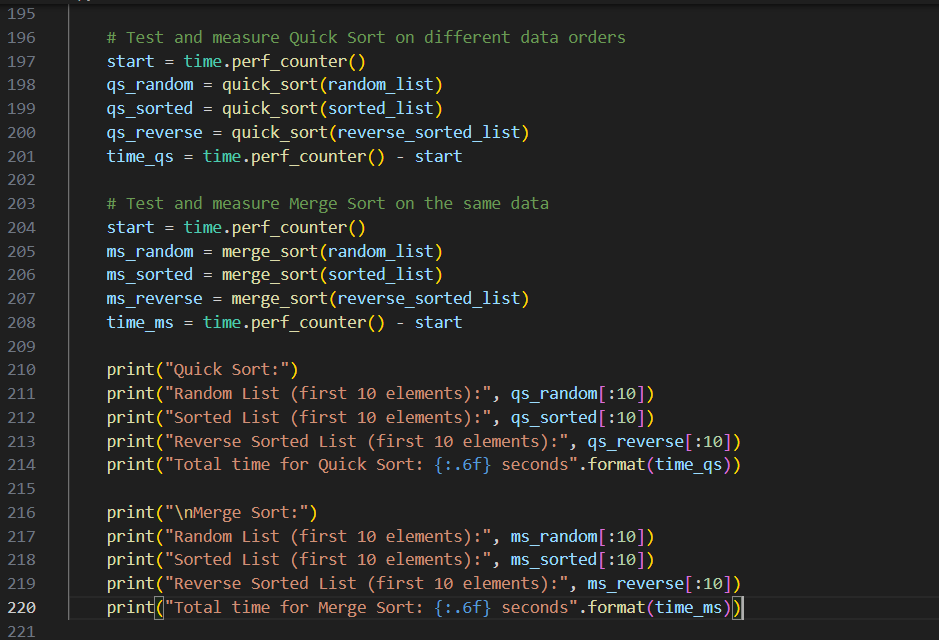
Prompt:

Implement recursive Quick Sort and Merge Sort with partially completed functions. Complete missing logic, add docstrings, and compare performance on random, sorted, and reverse-sorted lists.

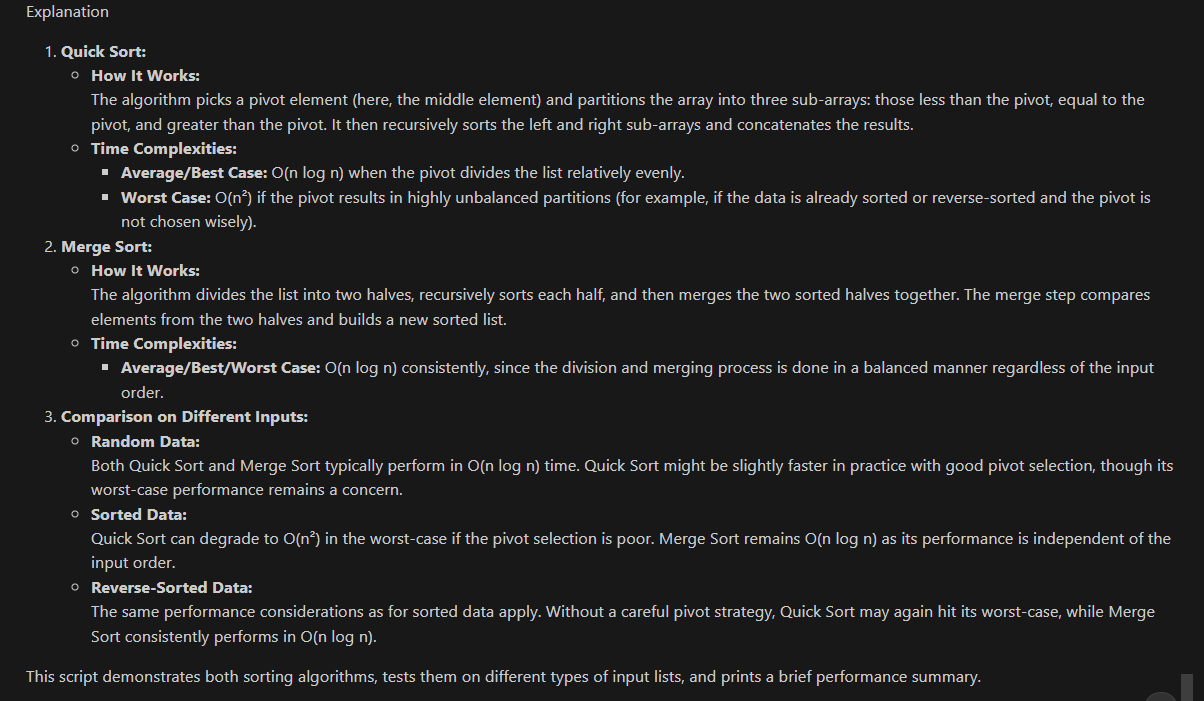
**CODE AND OUTPUT:**







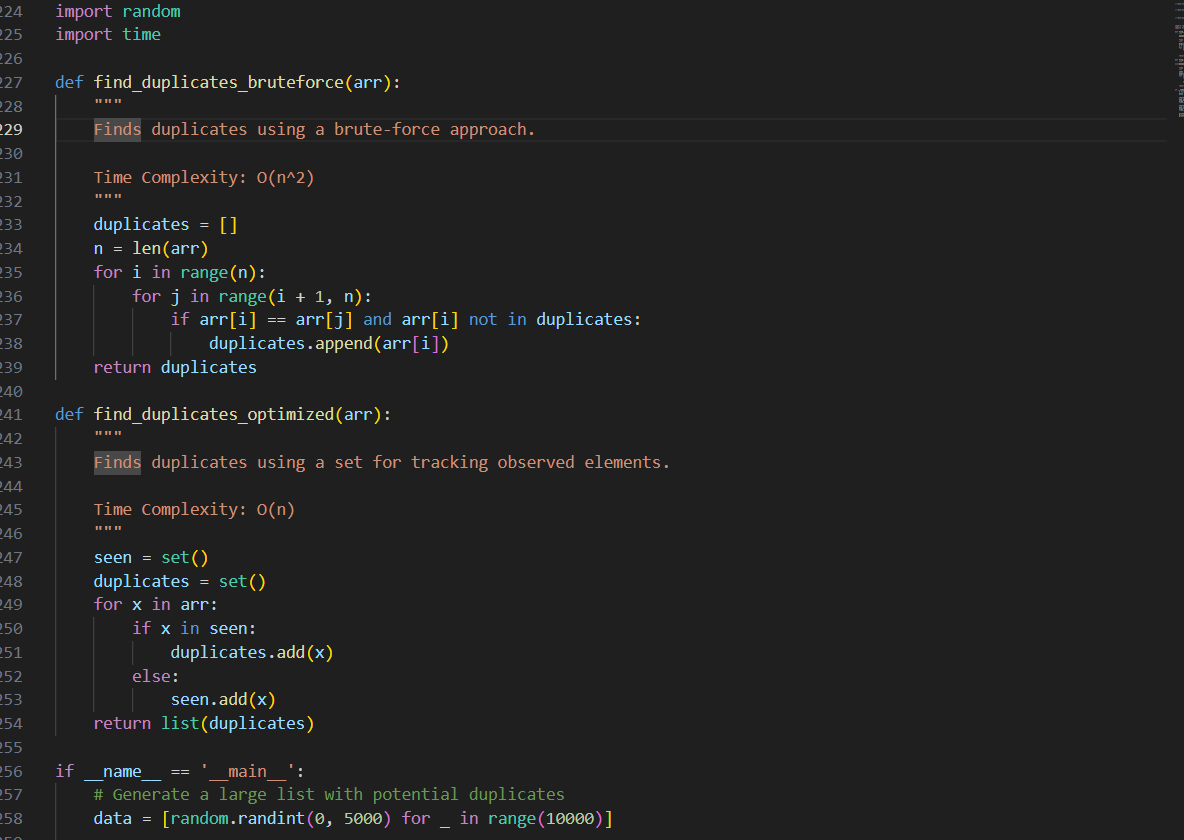
**EXPLANATION:**

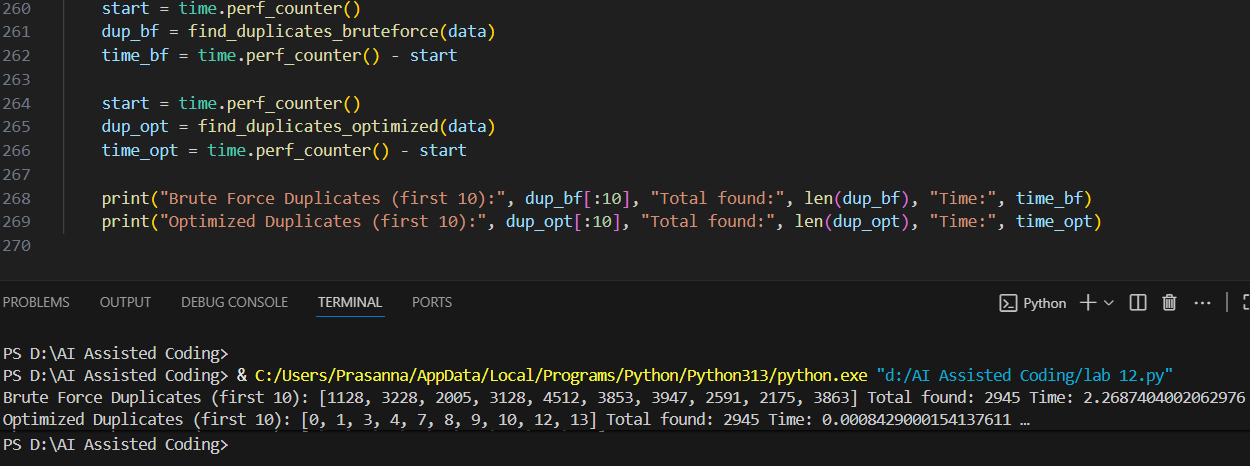


**Task 5:**

Prompt:

I've written a naive duplicate-finder function that checks each element against every other (O(n²) time). Please optimize it using a more efficient approach like sets or dictionaries to achieve O(n) time complexity. Also, compare the execution times of both versions on large input lists and explain how the complexity was improved.





**EXPLANATION:**

