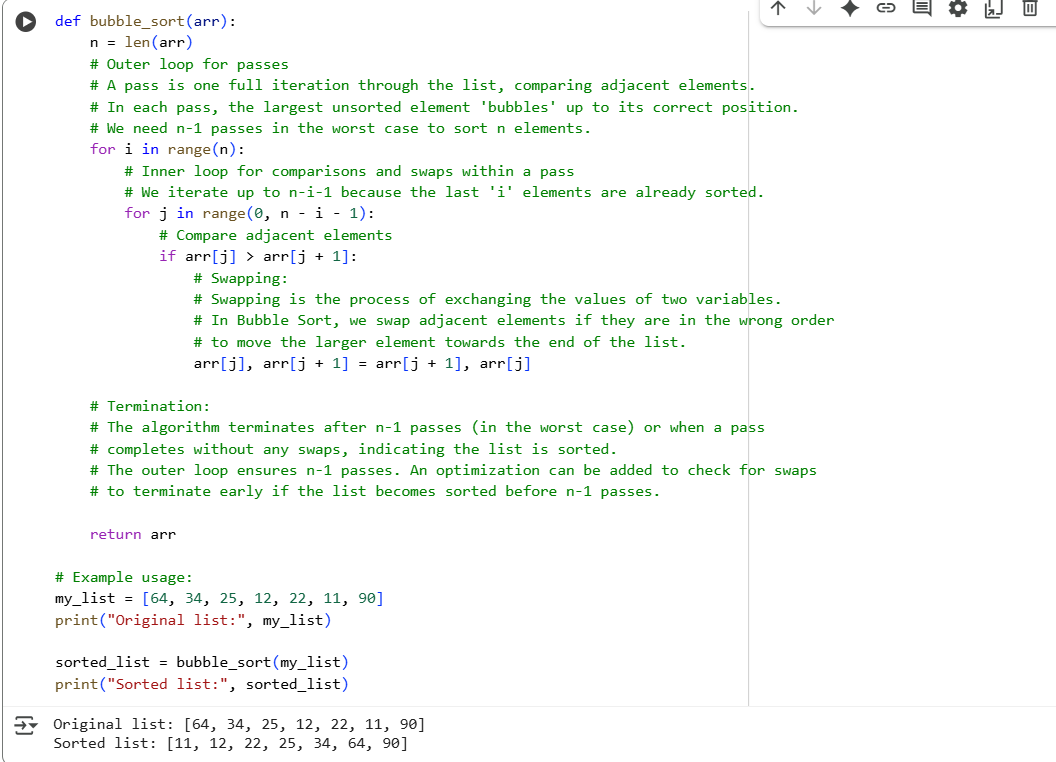
**ASSIGNMENT-12**

**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**

****

**EXPLAINATION:**

* Passes: Imagine repeatedly going through the list. Each complete run through the list, comparing adjacent elements and swapping them if they're in the wrong order, is called a "pass." In each pass, the largest unsorted element "bubbles" up to its correct position at the end of the unsorted portion of the list. For a list of n elements, in the worst case, we need n-1 passes to ensure the list is fully sorted.
* Swapping: This is the fundamental operation in Bubble Sort. When you compare two adjacent elements and find they are in the wrong order (e.g., the element on the left is larger than the element on the right in an ascending sort), you swap their positions. The code arr[j], arr[j + 1] = arr[j + 1], arr[j] is the standard Python way to perform this swap efficiently.
* Termination: The algorithm finishes when the list is sorted. The outer loop is designed to run for n-1 passes, guaranteeing sorting in the worst case. An optimization could be added to stop early if a pass completes without any swaps, indicating the list is already sorted.

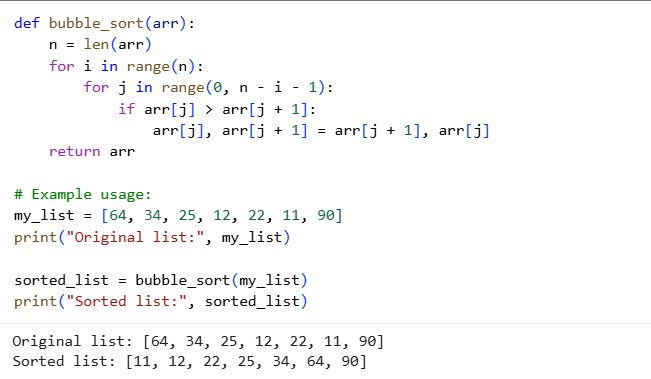
Essentially, Bubble Sort works by repeatedly stepping through the list, comparing adjacent elements and swapping them if they are in the wrong order. This process is repeated until the list is sorted.

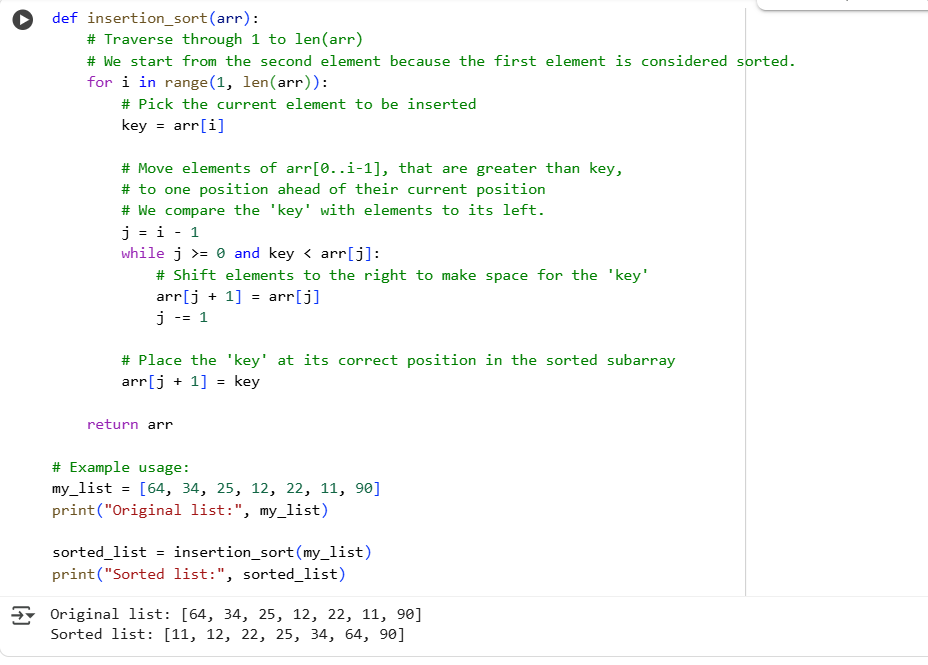
**TASK-2**

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.

**CODES OF BOBBLE SORT AND INSERTION SORT**

****

****

**EXPLANATION:**

Insertion Sort works by building a sorted subarray one element at a time. It takes each unsorted element and "inserts" it into its correct position within the already sorted portion of the array.

When the data is partially sorted, elements are often already close to their final sorted positions. In Insertion Sort, the inner while loop, which shifts elements to make space for the current element (key), will perform fewer iterations if the element is already close to its correct place. If an element is already in its sorted position, the while loop condition key < arr[j] will immediately be false, and no shifting or comparisons with elements further to the left will be needed.

In contrast, algorithms like Bubble Sort always perform comparisons and potentially swaps across the entire unsorted portion of the array in each pass, regardless of whether the data is partially sorted. This means they still do a lot of unnecessary work even if the data is nearly sorted.

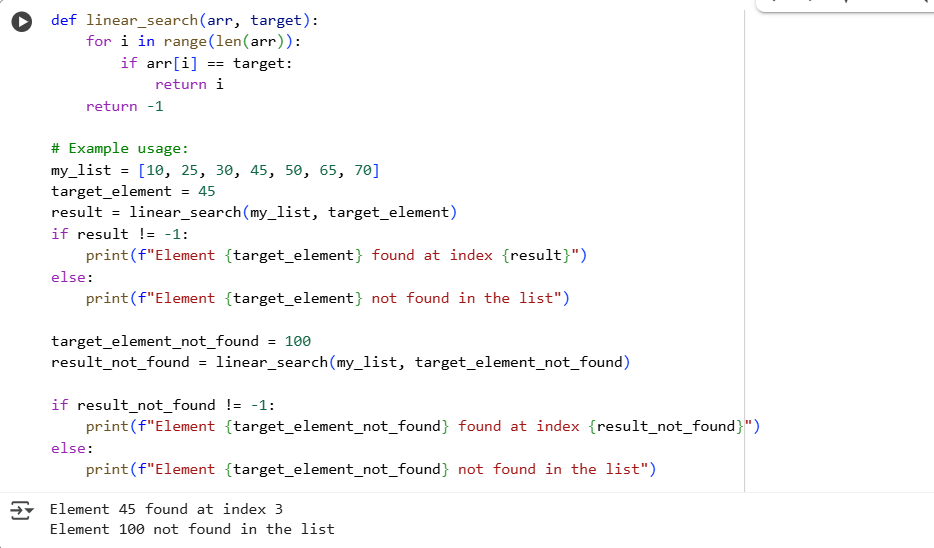
Therefore, because Insertion Sort's operations (comparisons and shifts) are directly tied to how far an element is from its sorted position, it benefits significantly from data that is already partially sorted, leading to fewer operations and better performance.

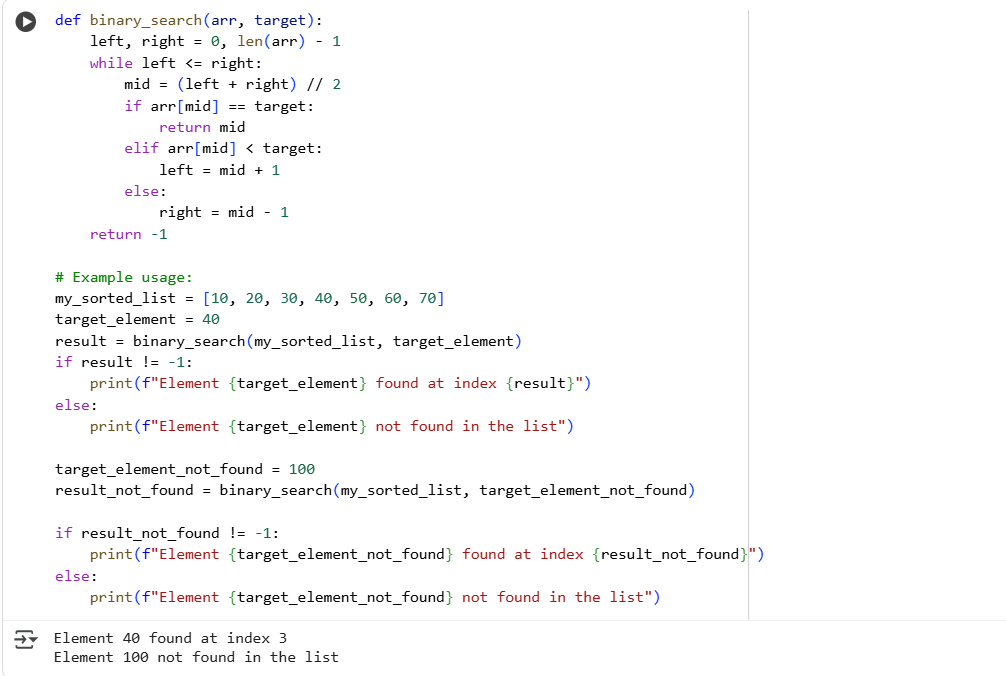
**TASK-3**

Binary Search vs Linear Search

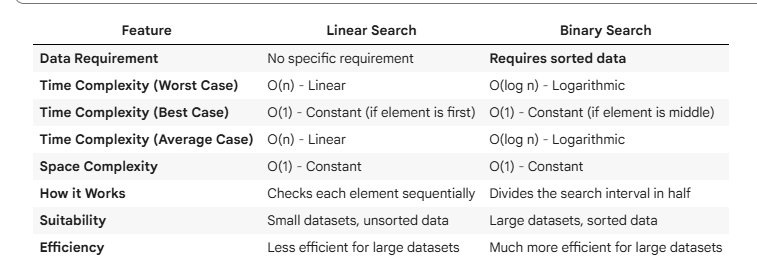
* Task: Implement both Linear Search and Binary Search.
* Instructions:
  + Use AI to generate docstrings and performance notes.
  + Test both algorithms on sorted and unsorted data.
  + Ask AI to explain when Binary Search is preferable.
* Expected Output:
  + Two implementations with docstrings.
 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

**CODE:**





**EXPLAINATION WITH TABLE:**

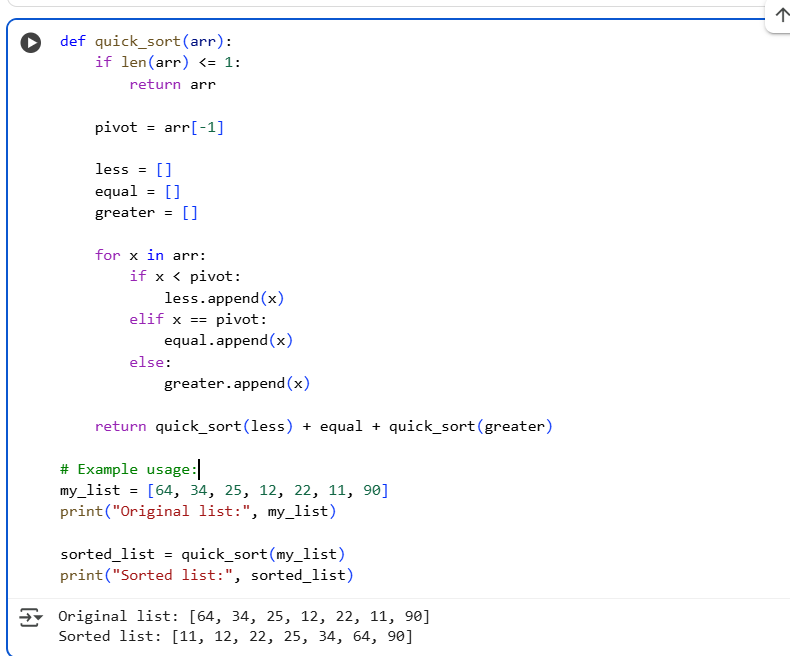
****

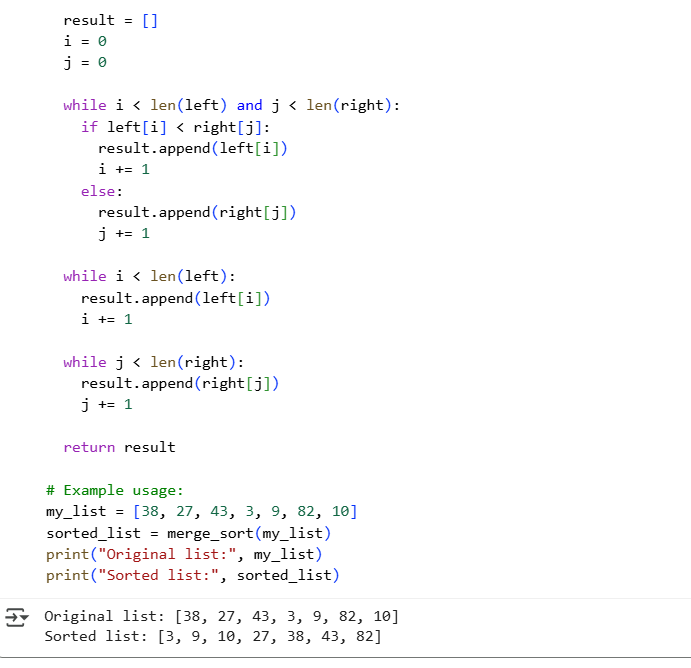
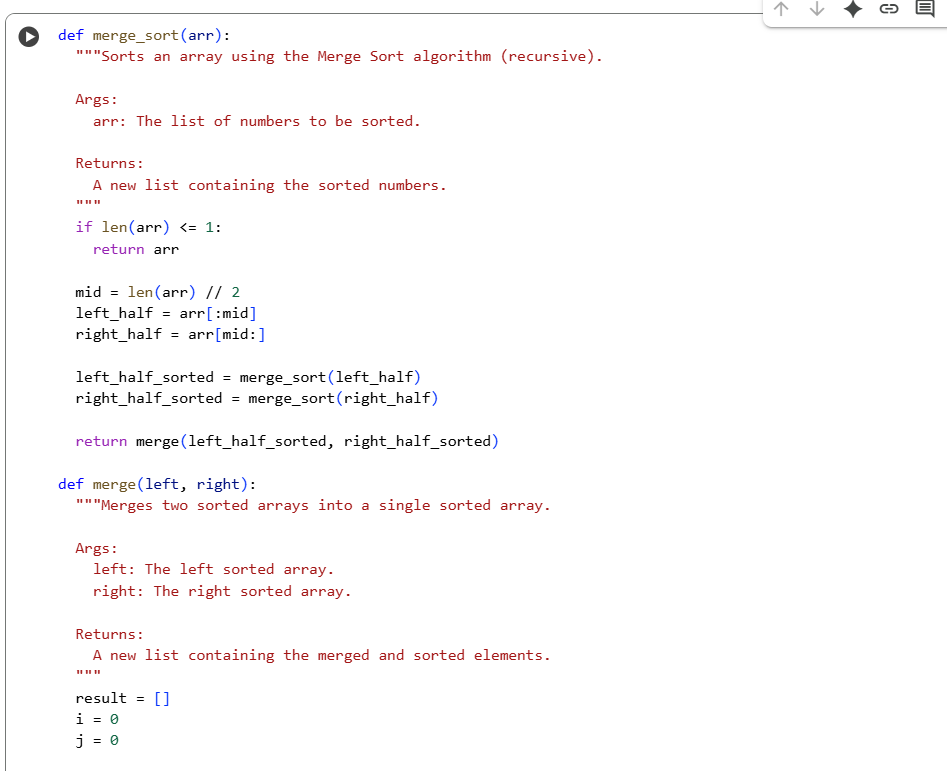
**TASK-4**

Quick Sort and Merge Sort Comparison

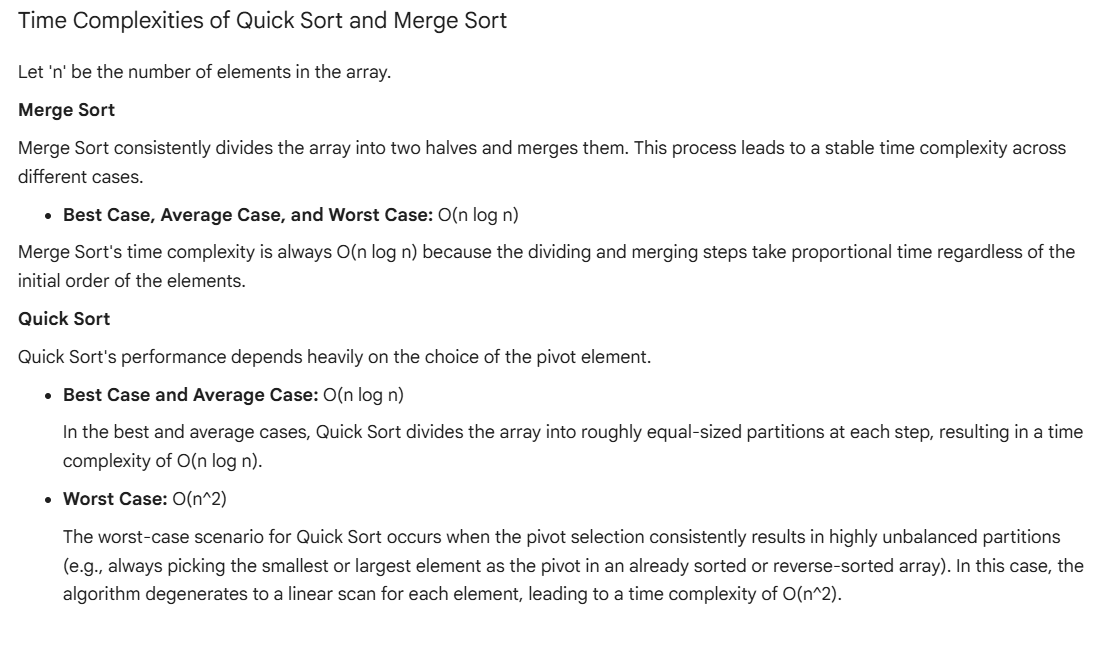
* **Task:** Implement Quick Sort and Merge Sort using recursion**.**
* **Instructions:**
  + Provide AI with partially completed functions for recursion.
  + Ask AI to complete the missing logic and add docstrings.
  + Compare both algorithms on random, sorted, and reverse-sorted lists.
* **Expected Output:**
  + Working Quick Sort and Merge Sort implementations.
  + AI-generated explanation of average, best, and worst-case complexities.

**CODES:**





**EXPLANATION**



**TASK-5**

AI-Suggested Algorithm Optimization

* Task: Give AI a naive algorithm (e.g., O(n²) duplicate search).
* Instructions:
  + Students write a brute force duplicate-finder.
  + Ask AI to optimize it (e.g., by using sets/dictionaries with O(n) time).
  + Compare execution times with large input sizes.
* Expected Output:
  + Two versions of the same algorithm (brute force + optimized).
  + AI explanation of how complexity was improved.