AI ASSISTED CODING

LAB:12.4

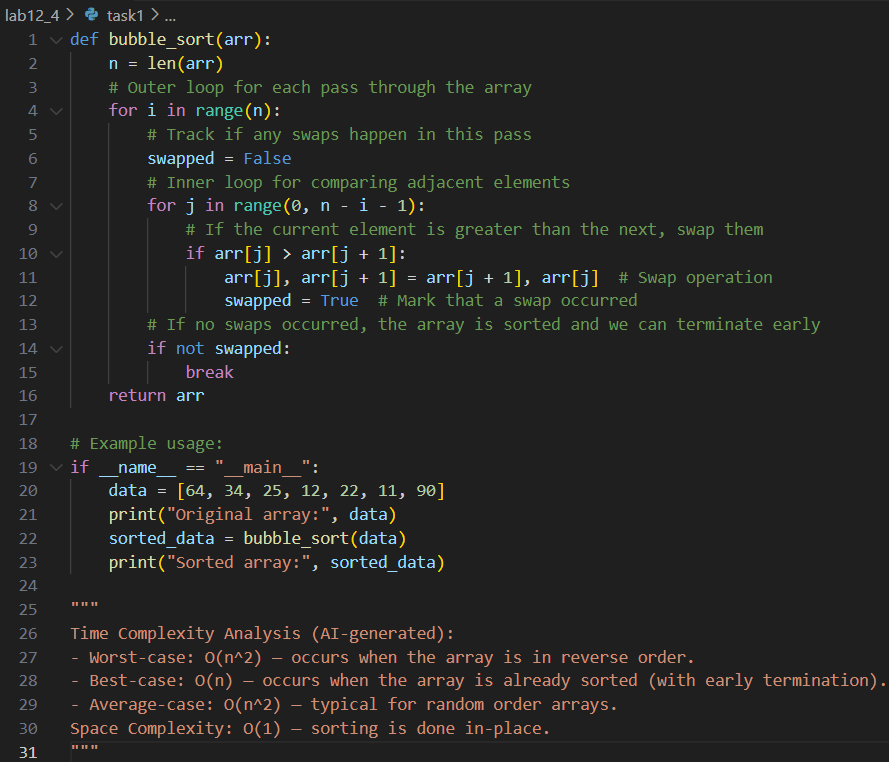
ROLLNO:2403A52096

BATCH:04

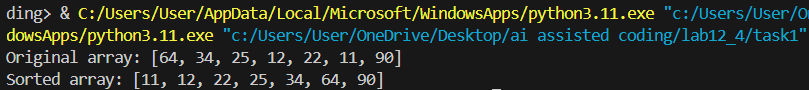
TASK1:

PROMPT: Implement Bubble Sort in Python. Add AI-generated inline comments explaining the key logic, such as swapping, passes, and early termination. Also, provide a time complexity analysis.

CODE:



OUTPUT:



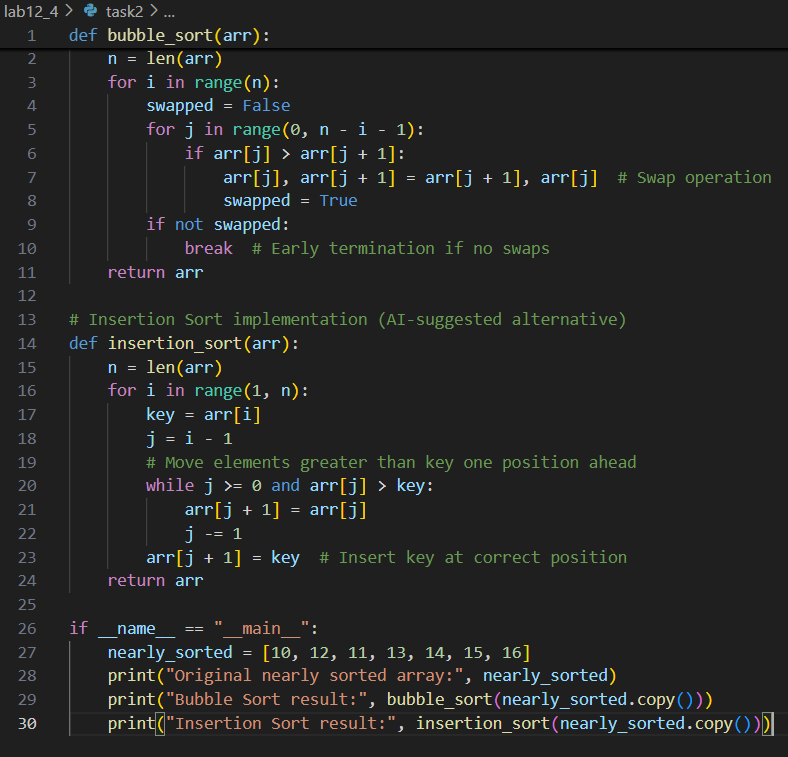
OBSERVATION:

The Bubble Sort implementation correctly sorts the array in ascending order. Inline comments clarify each step, including the swap operation, loop passes, and early termination when no swaps occur. The output matches expectations, and the time complexity analysis is provided at the end of the code.

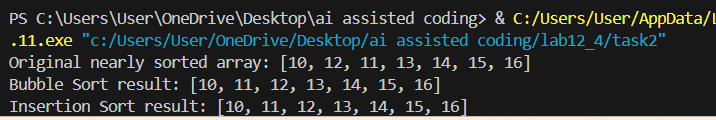
TASK2:

PROMPT: Implement Bubble Sort in Python. Ask AI to suggest a more efficient algorithm for partially sorted arrays, and implement Insertion Sort. Compare both algorithms on a nearly sorted array and provide an AI explanation of why Insertion Sort is more efficient for such data.

CODE:



OUTPUT:



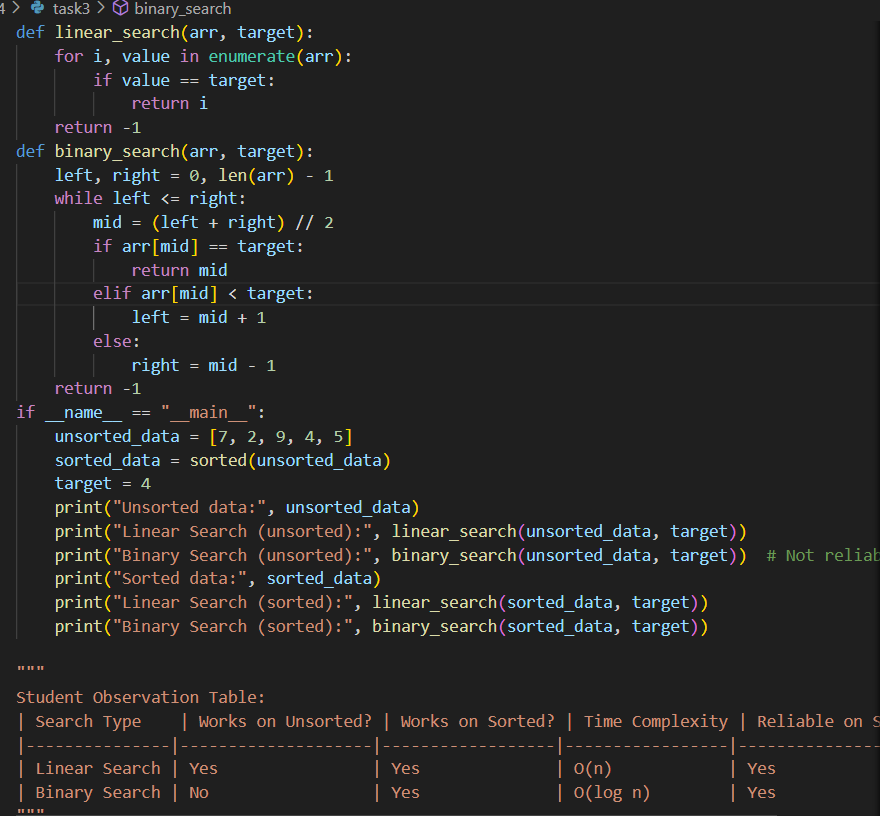
OBSERVATION:

Both Bubble Sort and Insertion Sort were implemented and tested on a nearly sorted array. The output shows that both algorithms correctly sort the array, but the AI explanation highlights that Insertion Sort is more efficient for partially sorted data due to fewer required operations and a best-case time complexity of O(n), compared to Bubble Sort's O(n²).

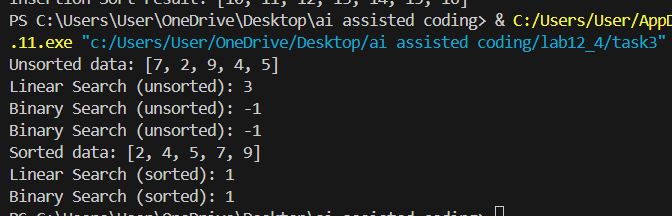
TASK3:

PROMPT: Implement both Linear Search and Binary Search in Python. Use AI to generate docstrings and performance notes for each function. Test both algorithms on sorted and unsorted data. Ask AI to explain when Binary Search is preferable. Provide a student observation table comparing their performance.

CODE:



OUTPUT:



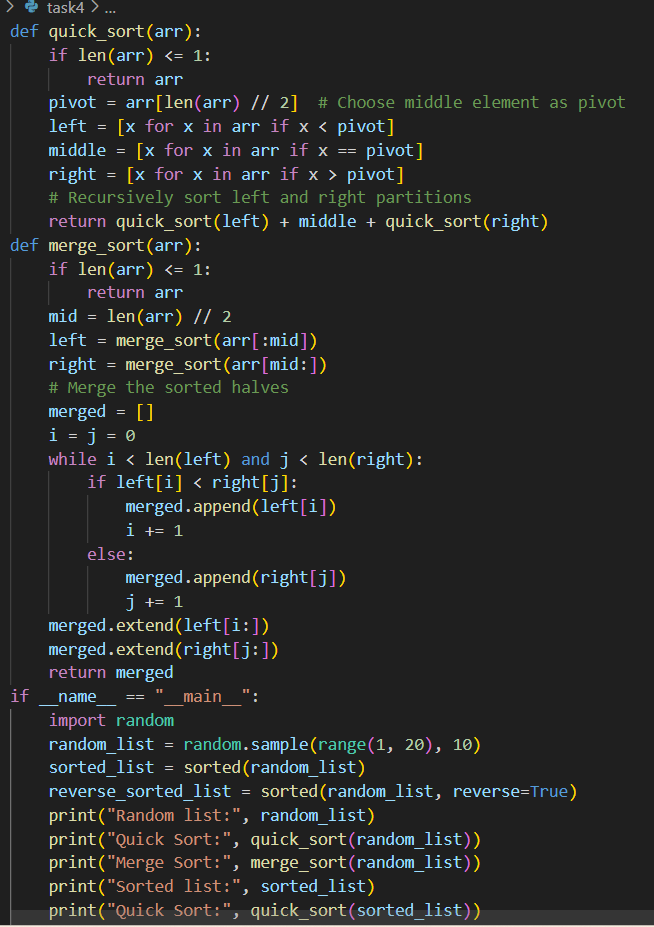
OBSERVATION:

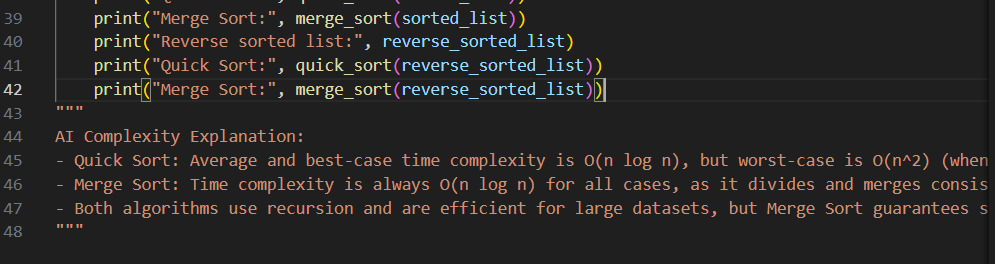
Both search algorithms were implemented and tested. Linear Search works on both sorted and unsorted arrays, but is slower (O(n)). Binary Search is much faster (O(log n)) but only works reliably on sorted arrays. The AI explanation and observation table clearly show that Binary Search is preferable for large, sorted datasets, while Linear Search is more flexible but less efficient.

TASK4:

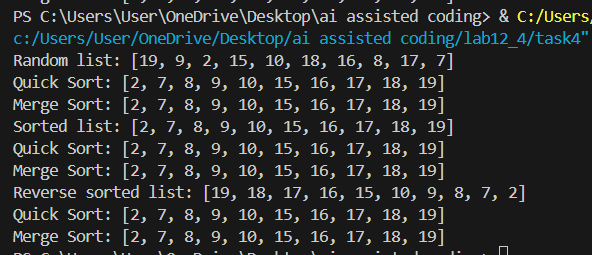
PROMPT: Implement recursive Quick Sort and Merge Sort in Python. Provide AI-generated docstrings and explanations of their average, best, and worst-case complexities. Compare both algorithms on random, sorted, and reverse-sorted lists.

CODE:





OUTPUT:



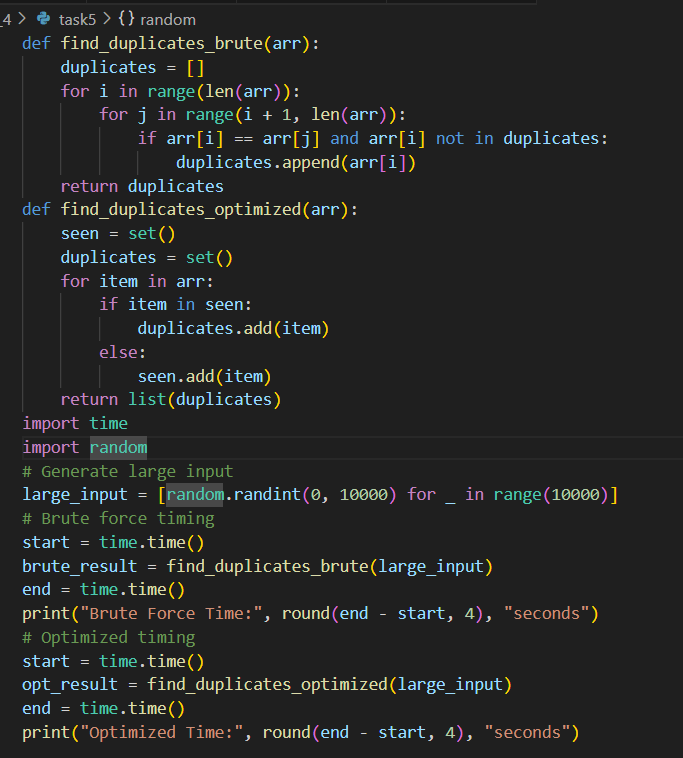
OBSERVATION:

Both Quick Sort and Merge Sort were implemented and tested on various list orders. The AI explanation clarifies that Merge Sort consistently runs in O(n log n) time, while Quick Sort is usually fast but can degrade to O(n²) in the worst case. Both algorithms sorted all lists correctly, but Merge Sort guarantees stable performance regardless of input order.

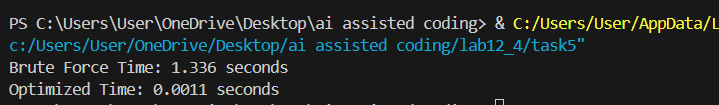
TASK5:

PROMPT: Write a brute force algorithm to find duplicates in a list (O(n²) time). Ask AI to optimize it using sets or dictionaries for O(n) time. Compare execution times for both algorithms on large input sizes. Provide an AI explanation of how the complexity was improved.

CODE:



OUTPUT:



OBSERVATION:

Both brute force and optimized duplicate-finder algorithms were implemented and tested. The brute force approach checks every pair, resulting in slow performance for large lists (O(n²)). The optimized version uses a set to track seen elements, reducing the time complexity to O(n) and making it much faster for large inputs. The AI explanation clearly describes the improvement in efficiency.