ASSIGNMENT-12.4

Name:-P. Likitha

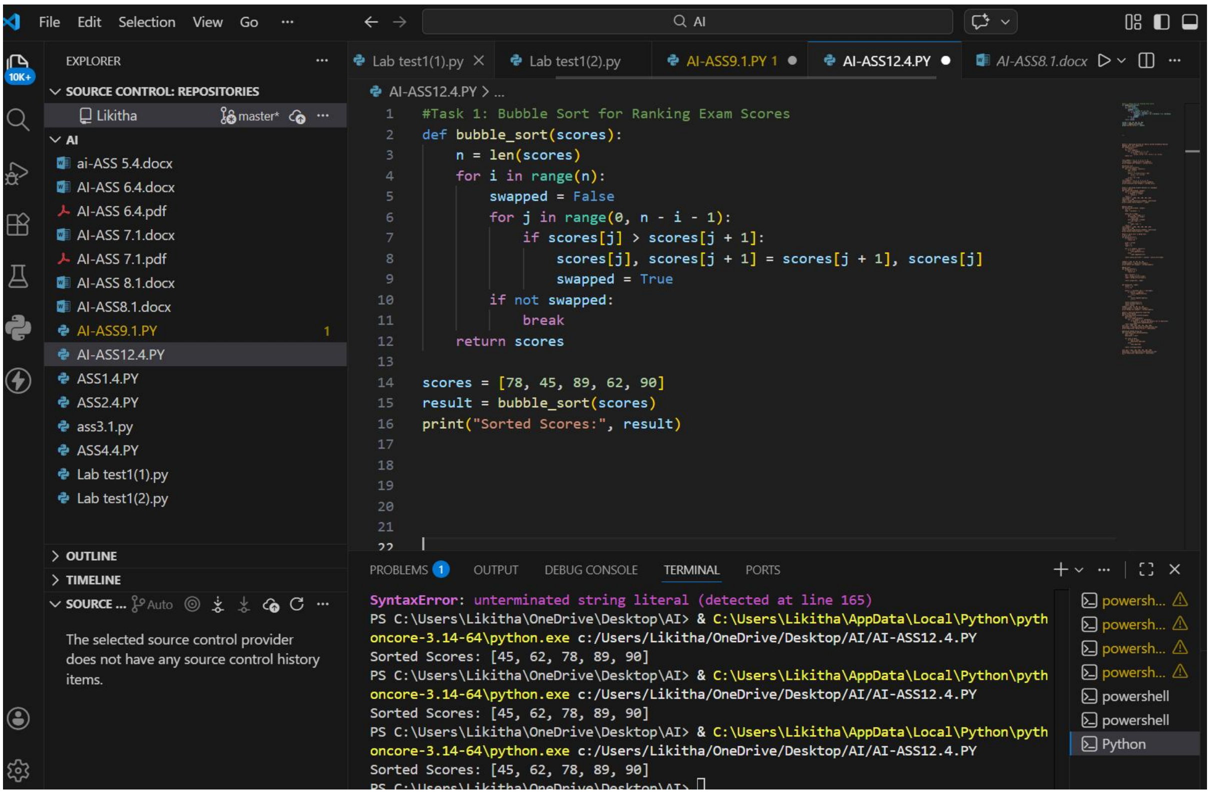
HT.NO:- 2303A52393

Batch:- 43

Task 1: Bubble Sort for Ranking Exam Scores

Prompt

Generate a Python program to implement Bubble Sort for sorting student exam scores. The prompt requested inline comments explaining comparisons, swaps, and iteration passes. It also asked to include an early termination condition to stop the algorithm when the list becomes sorted. Additionally, the prompt required a brief explanation of best, average, and worst-case time complexity.



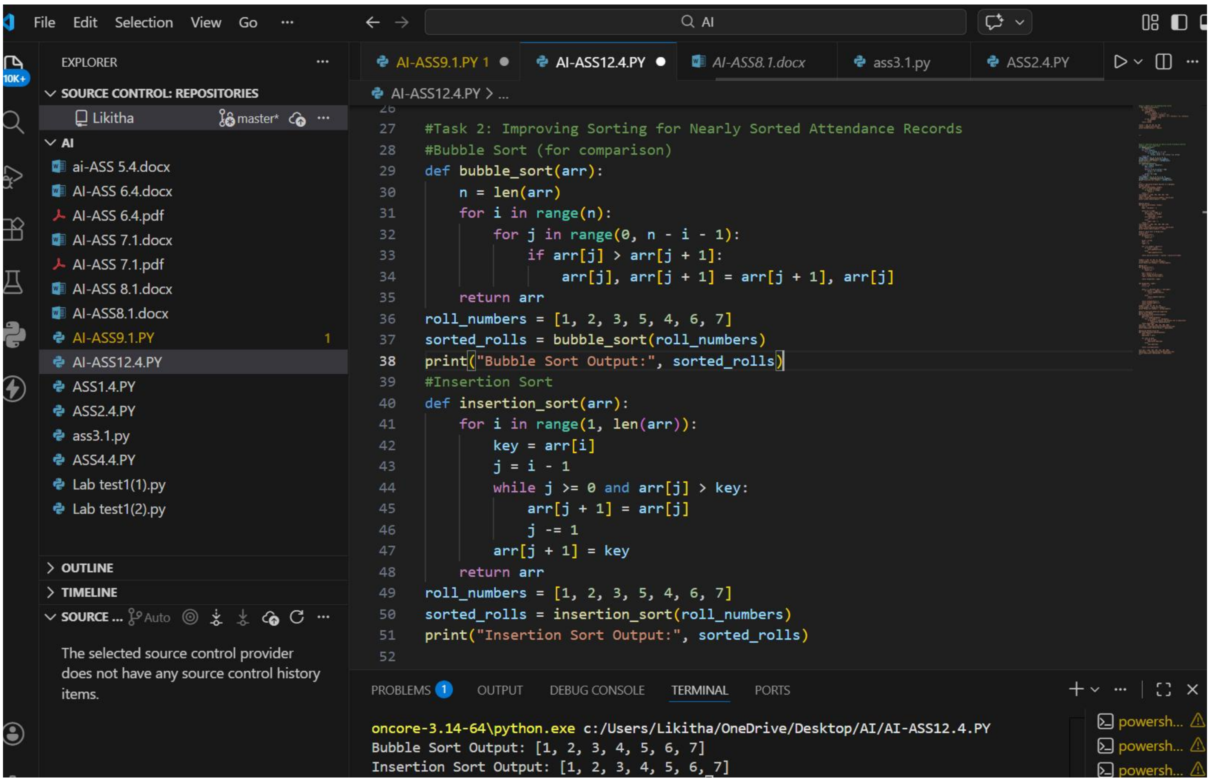
Observation

Bubble Sort successfully sorted the student scores in ascending order by repeatedly comparing adjacent elements. The early termination condition reduced unnecessary iterations when the list was already sorted. The algorithm is simple and easy to understand for small datasets. However, its performance decreases for large inputs due to quadratic time complexity. Therefore, Bubble Sort is suitable only for small-scale result processing.

Task 2: Improving Sorting for Nearly Sorted Attendance Records

Prompt

The Bubble Sort algorithm for nearly sorted attendance data and suggest a more efficient alternative. The prompt requested an Insertion Sort implementation with proper explanation. It also asked AI to explain why Insertion Sort performs better on partially sorted data. A comparison of execution behavior between both algorithms was also requested.



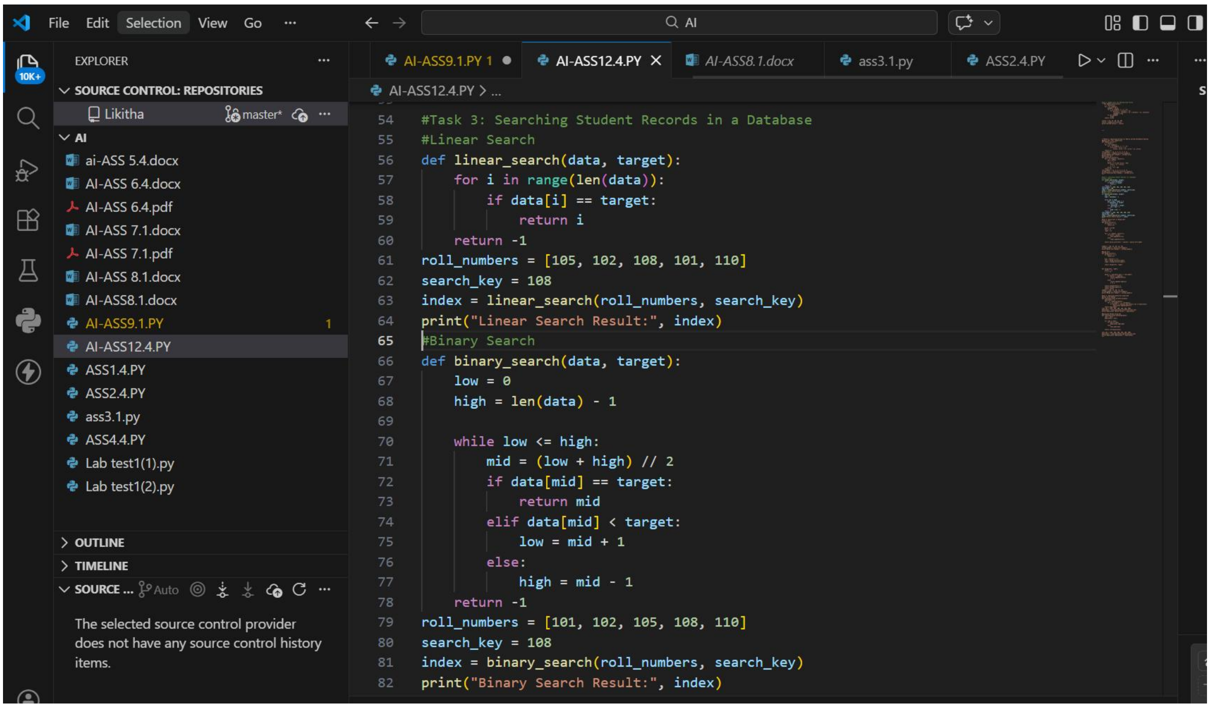
Observation

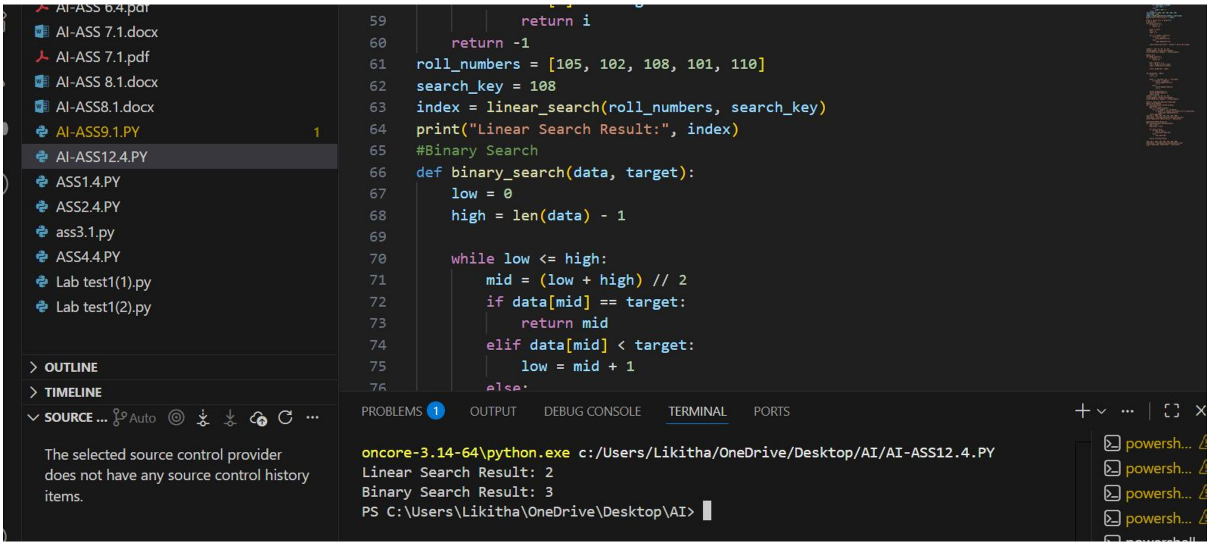
Both Bubble Sort and Insertion Sort produced the correct sorted output for attendance records. However, Bubble Sort performed unnecessary comparisons even when data was almost sorted. Insertion Sort improved performance by shifting only the misplaced elements. This resulted in faster execution and fewer operations. Hence, Insertion Sort is more suitable for nearly sorted datasets.

Task 3: Searching Student Records in a Database

Prompt

Generate Python programs for Linear Search and Binary Search algorithms. The prompt requested proper docstrings explaining parameters and return values. It also asked AI to explain when Binary Search can be applied and to highlight performance differences. Use cases for searching sorted and unsorted student records were also requested.





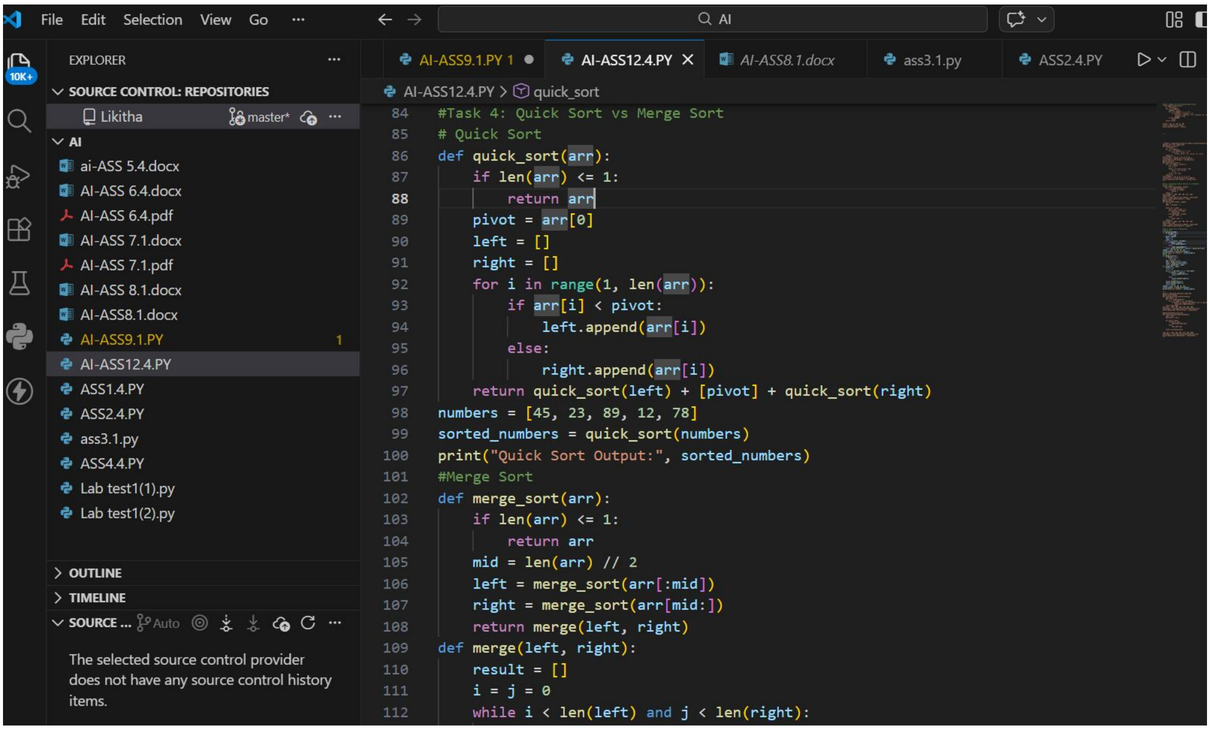
Observation

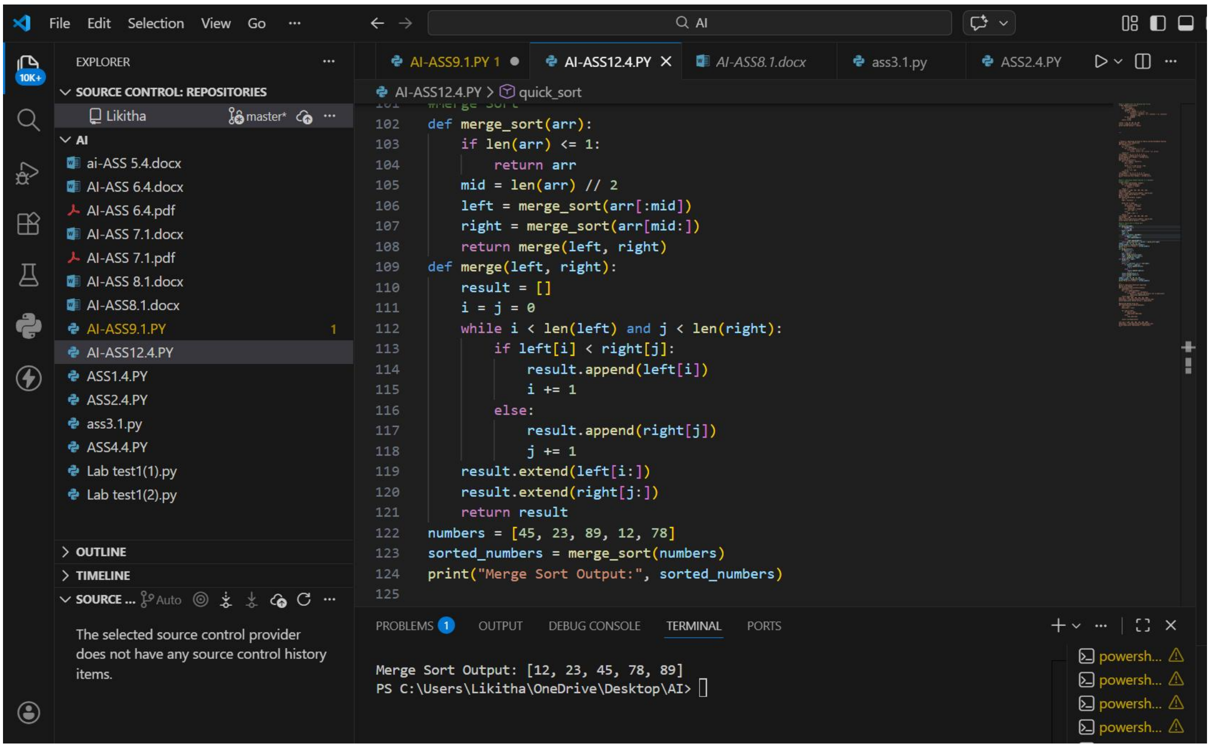
Linear Search successfully located student records by checking each element sequentially. It works for both sorted and unsorted data but is slow for large datasets. Binary Search required sorted data and reduced the search space by half at each step. This resulted in significantly faster searches. Therefore, Binary Search is preferred for large, sorted databases.

Task 4: Choosing Between Quick Sort and Merge Sort

Prompt

written recursive functions for Quick Sort and Merge Sort and asked to complete them. The prompt requested meaningful docstrings and explanations of recursion in each algorithm. It also asked to analyze their performance on random, sorted, and reverse-sorted data. A comparison of time complexities and practical use cases was included.





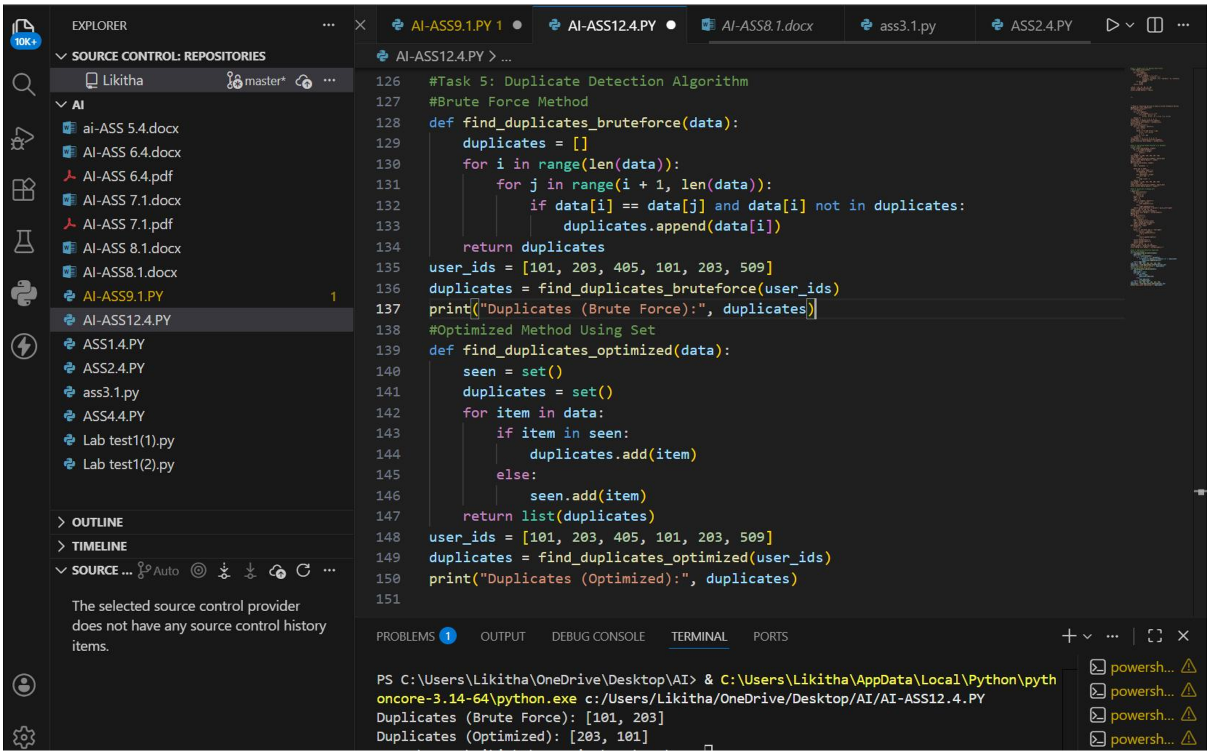
Observation

Quick Sort efficiently sorted random datasets using a pivot-based partitioning approach. However, its performance degraded in the worst case for sorted or reverse-sorted data. Merge Sort consistently performed well due to its divide-and-conquer strategy. It maintained O(n log n) time complexity in all cases. Thus, Merge Sort is more reliable for large datasets.

Task 5: Optimizing a Duplicate Detection Algorithm

Prompt

Analyze a brute-force duplicate detection algorithm using nested loops. The prompt requested a time complexity analysis and an optimized solution using sets or dictionaries. It also asked to rewrite the algorithm with improved efficiency. A conceptual comparison for large input sizes was requested.



Observation

The brute-force duplicate detection algorithm correctly identified repeated user IDs but was inefficient. Its quadratic time complexity caused performance issues for large datasets. The optimized approach using a set reduced lookup time significantly. This improved the algorithm's efficiency to linear time. Hence, optimized data structures greatly enhance performance.