

ASSIGNMENT-12.4

Name:-P. Likitha

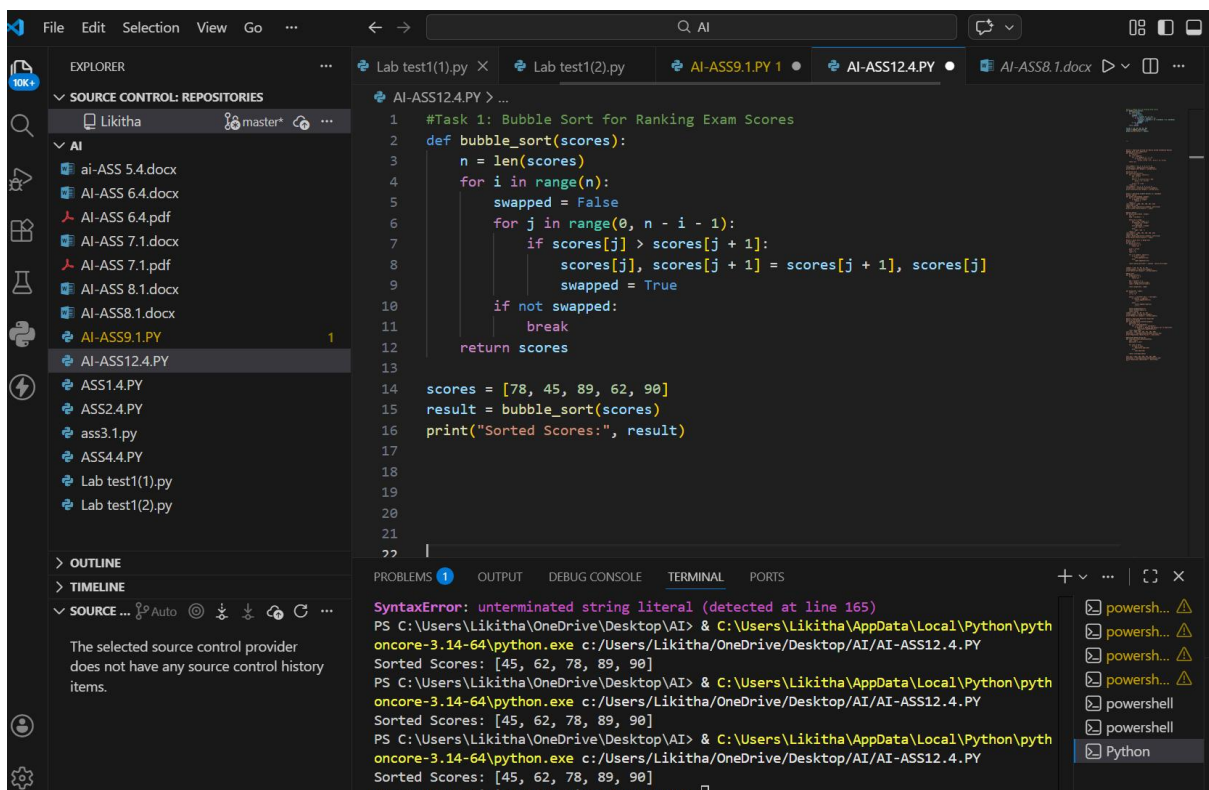
HT.NO:- 2303A52393

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.



The screenshot shows a Visual Studio Code editor with a Python file named `AI-ASS12.4.PY`. The code implements a Bubble Sort algorithm with inline comments. The initial list of scores is `[78, 45, 89, 62, 90]`. The output shows the sorted list `[45, 62, 78, 89, 90]`. The terminal window at the bottom shows the command `python AI-ASS12.4.PY` being executed, and the output is `Sorted Scores: [45, 62, 78, 89, 90]`. The left sidebar shows the Explorer view with a list of files and folders, including `AI-ASS12.4.PY`.

```
1 #Task 1: Bubble Sort for Ranking Exam Scores
2 def bubble_sort(scores):
3     n = len(scores)
4     for i in range(n):
5         swapped = False
6         for j in range(0, n - i - 1):
7             if scores[j] > scores[j + 1]:
8                 scores[j], scores[j + 1] = scores[j + 1], scores[j]
9                 swapped = True
10        if not swapped:
11            break
12    return scores
13
14 scores = [78, 45, 89, 62, 90]
15 result = bubble_sort(scores)
16 print("Sorted Scores:", result)
17
18
19
20
21
22
```

Sorted Scores: [45, 62, 78, 89, 90]

Sorted Scores: [45, 62, 78, 89, 90]

Sorted Scores: [45, 62, 78, 89, 90]

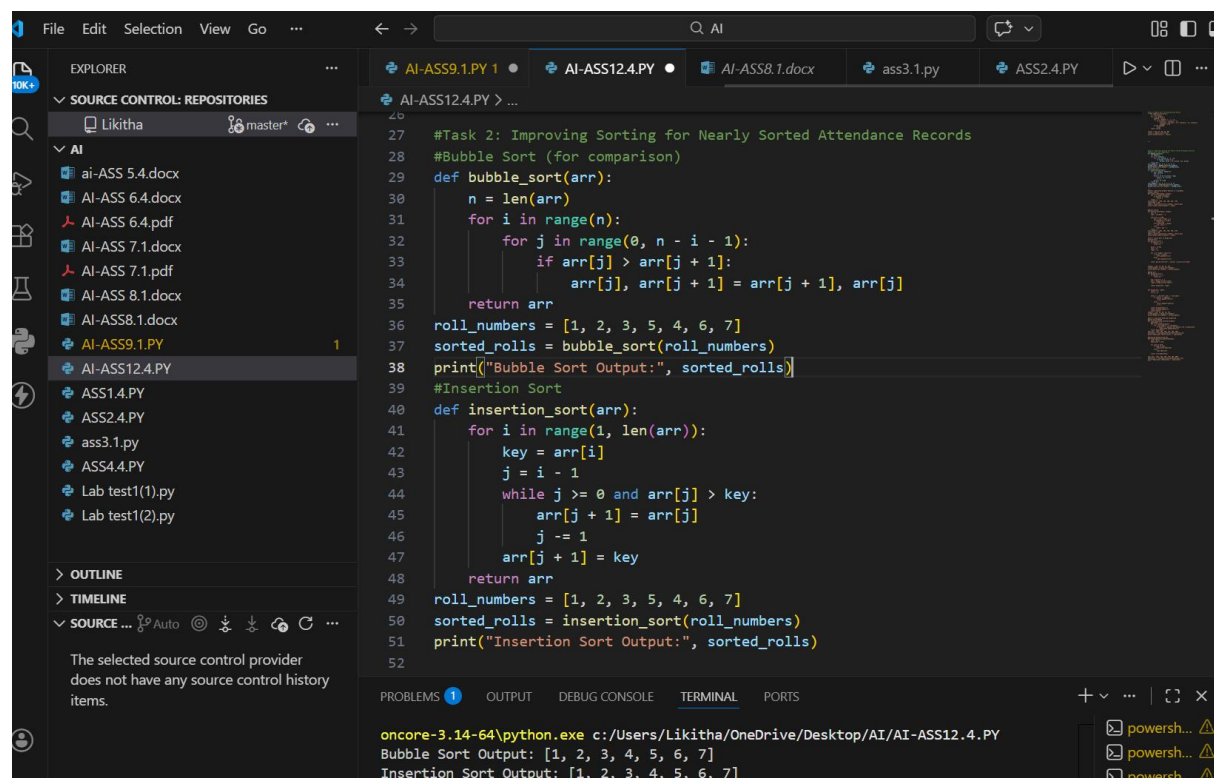
Observation

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.



The screenshot shows a code editor with a dark theme. The Explorer panel on the left shows a project named 'AI' with several files, including 'AI-ASS12.4.PY' which is selected. The main editor area displays the code for 'Task 2: Improving Sorting for Nearly Sorted Attendance Records'. The code implements two sorting algorithms: Bubble Sort and Insertion Sort. The Bubble Sort function is defined as 'def bubble_sort(arr):' and the Insertion Sort function as 'def insertion_sort(arr):'. The code uses a list 'roll_numbers = [1, 2, 3, 5, 4, 6, 7]' and compares the output of both sorting functions. The terminal at the bottom shows the output of the program, which is the sorted list [1, 2, 3, 4, 5, 6, 7] for both algorithms.

```
27 #Task 2: Improving Sorting for Nearly Sorted Attendance Records
28 #Bubble Sort (for comparison)
29 def bubble_sort(arr):
30     n = len(arr)
31     for i in range(n):
32         for j in range(0, n - i - 1):
33             if arr[j] > arr[j + 1]:
34                 arr[j], arr[j + 1] = arr[j + 1], arr[j]
35     return arr
36 roll_numbers = [1, 2, 3, 5, 4, 6, 7]
37 sorted_rolls = bubble_sort(roll_numbers)
38 print("Bubble Sort Output:", sorted_rolls)
39 #Insertion Sort
40 def insertion_sort(arr):
41     for i in range(1, len(arr)):
42         key = arr[i]
43         j = i - 1
44         while j >= 0 and arr[j] > key:
45             arr[j + 1] = arr[j]
46             j -= 1
47         arr[j + 1] = key
48     return arr
49 roll_numbers = [1, 2, 3, 5, 4, 6, 7]
50 sorted_rolls = insertion_sort(roll_numbers)
51 print("Insertion Sort Output:", sorted_rolls)
52
```

oncore-3.14-64\python.exe c:/Users/Likitha/OneDrive/Desktop/AI/AI-ASS12.4.PY
Bubble Sort Output: [1, 2, 3, 4, 5, 6, 7]
Insertion Sort Output: [1, 2, 3, 4, 5, 6, 7]

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.

The screenshot shows the Visual Studio Code editor with a file explorer on the left and a code editor in the center. The file explorer shows a project named 'Likitha' with a 'master' branch. The code editor displays the following Python code:

```
54 #Task 3: Searching Student Records in a Database
55 #Linear Search
56 def linear_search(data, target):
57     for i in range(len(data)):
58         if data[i] == target:
59             return i
60     return -1
61 roll_numbers = [105, 102, 108, 101, 110]
62 search_key = 108
63 index = linear_search(roll_numbers, search_key)
64 print("Linear Search Result:", index)
65 #Binary Search
66 def binary_search(data, target):
67     low = 0
68     high = len(data) - 1
69
70     while low <= high:
71         mid = (low + high) // 2
72         if data[mid] == target:
73             return mid
74         elif data[mid] < target:
75             low = mid + 1
76         else:
77             high = mid - 1
78     return -1
79 roll_numbers = [101, 102, 105, 108, 110]
80 search_key = 108
81 index = binary_search(roll_numbers, search_key)
82 print("Binary Search Result:", index)
```

The screenshot shows the Visual Studio Code editor with the same Python code as the previous image. The code is being executed, and the output is displayed in the terminal at the bottom. The output shows the results of the linear and binary search algorithms:

```
oncore-3.14-64\python.exe c:/Users/Likitha/OneDrive/Desktop/AI/AI-ASS12.4.PY
Linear Search Result: 2
Binary Search Result: 3
PS C:\Users\Likitha\OneDrive\Desktop\AI>
```

Observation

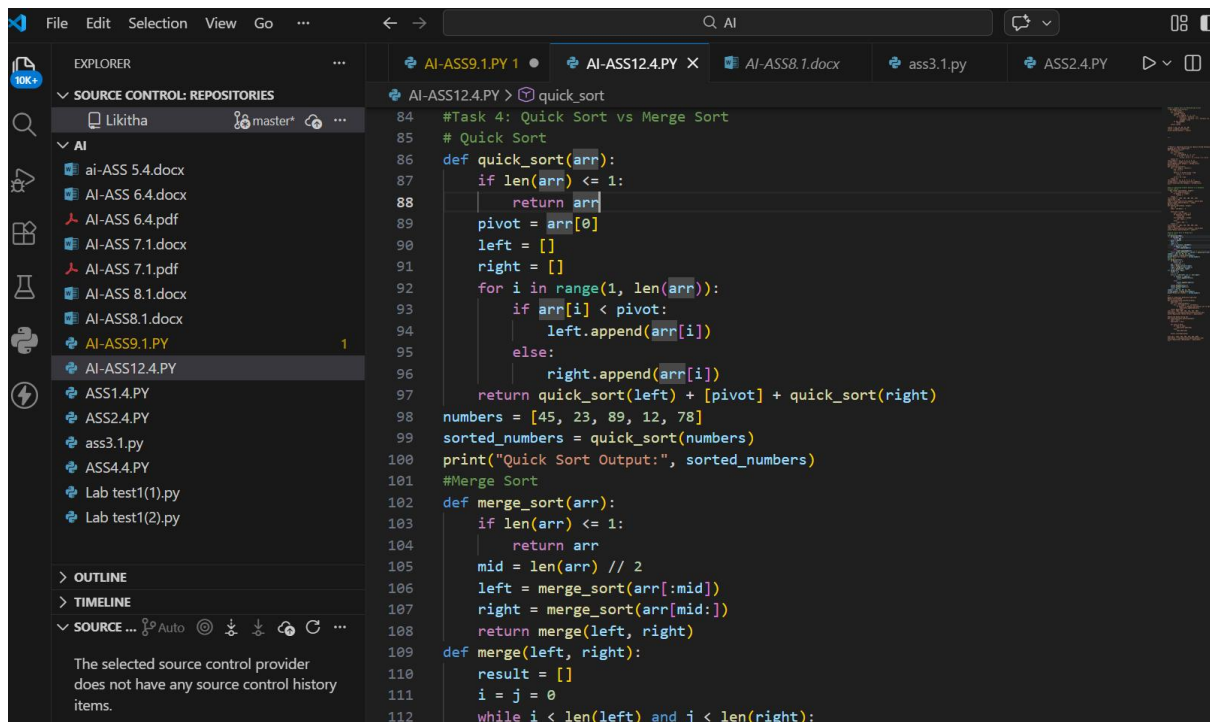
Linear Search successfully located student records by checking each element sequentially. It works for both sorted and unsorted data but is slow for large datasets. Binary Search required sorted data and reduced the search space by half at each step. This resulted in significantly faster searches. Therefore,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

Task 4: Choosing Between Quick Sort and Merge Sort

Prompt

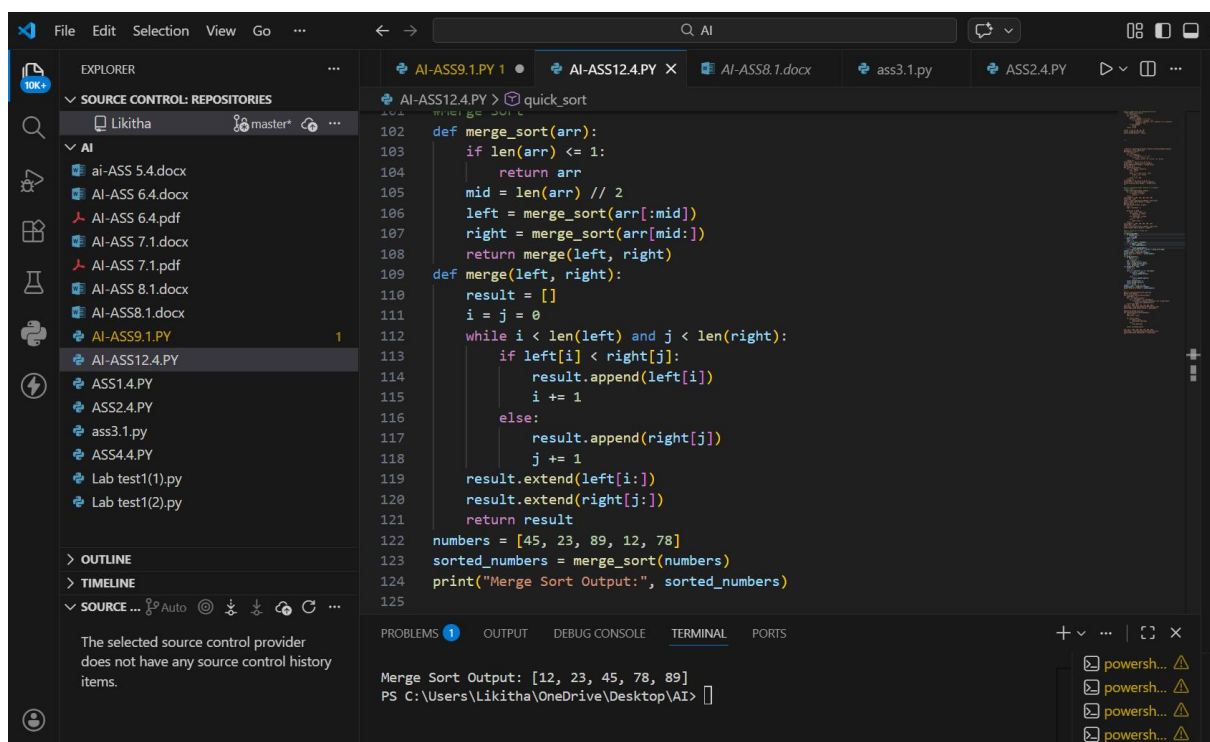
written recursive functions for Quick Sort and Merge Sort and asked to complete them. The prompt requested meaningful docstrings and explanations of recursion in each algorithm. It also asked to

analyze their performance on random, sorted, and reverse-sorted data. A comparison of time complexities and practical use cases was included.



The screenshot shows the Visual Studio Code editor with the file explorer on the left and the code editor in the center. The file explorer shows a project named 'Likitha' with a subdirectory 'AI' containing several files. The file 'AI-ASS12.4.PY' is selected. The code editor displays the implementation of Quick Sort and Merge Sort. The Quick Sort implementation is shown from line 84 to 112. The Merge Sort implementation is shown from line 102 to 112. The code is as follows:

```
84 #Task 4: Quick Sort vs Merge Sort
85 # Quick Sort
86 def quick_sort(arr):
87     if len(arr) <= 1:
88         return arr
89     pivot = arr[0]
90     left = []
91     right = []
92     for i in range(1, len(arr)):
93         if arr[i] < pivot:
94             left.append(arr[i])
95         else:
96             right.append(arr[i])
97     return quick_sort(left) + [pivot] + quick_sort(right)
98 numbers = [45, 23, 89, 12, 78]
99 sorted_numbers = quick_sort(numbers)
100 print("Quick Sort Output:", sorted_numbers)
101 #Merge Sort
102 def merge_sort(arr):
103     if len(arr) <= 1:
104         return arr
105     mid = len(arr) // 2
106     left = merge_sort(arr[:mid])
107     right = merge_sort(arr[mid:])
108     return merge(left, right)
109 def merge(left, right):
110     result = []
111     i = j = 0
112     while i < len(left) and j < len(right):
```



The screenshot shows the Visual Studio Code editor with the file explorer on the left and the code editor in the center. The file explorer shows a project named 'Likitha' with a subdirectory 'AI' containing several files. The file 'AI-ASS12.4.PY' is selected. The code editor displays the implementation of Merge Sort. The Merge Sort implementation is shown from line 102 to 125. The code is as follows:

```
102 def merge_sort(arr):
103     if len(arr) <= 1:
104         return arr
105     mid = len(arr) // 2
106     left = merge_sort(arr[:mid])
107     right = merge_sort(arr[mid:])
108     return merge(left, right)
109 def merge(left, right):
110     result = []
111     i = j = 0
112     while i < len(left) and j < len(right):
113         if left[i] < right[j]:
114             result.append(left[i])
115             i += 1
116         else:
117             result.append(right[j])
118             j += 1
119     result.extend(left[i:])
120     result.extend(right[j:])
121     return result
122 numbers = [45, 23, 89, 12, 78]
123 sorted_numbers = merge_sort(numbers)
124 print("Merge Sort Output:", sorted_numbers)
125
```

The bottom of the screenshot shows the terminal output: 'Merge Sort Output: [12, 23, 45, 78, 89]' and the command prompt 'PS C:\Users\Likitha\OneDrive\Desktop\AI>'.

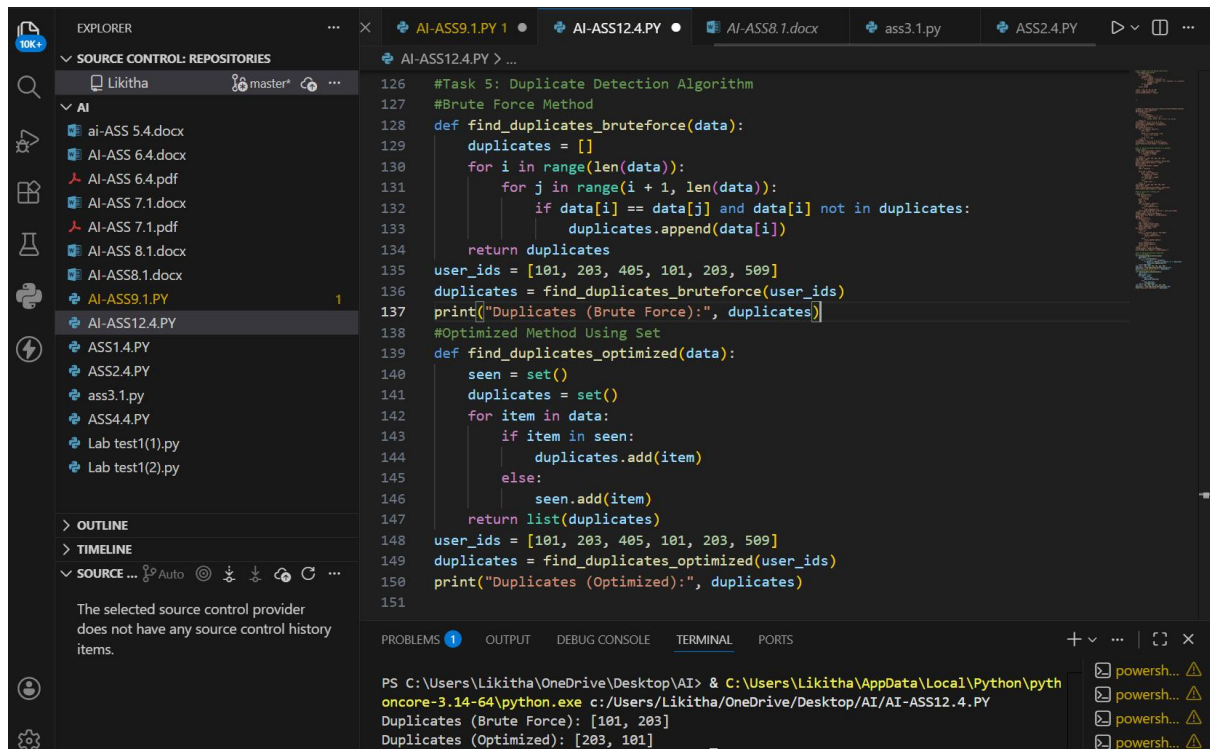
Observation

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.



```
126 #Task 5: Duplicate Detection Algorithm
127 #Brute Force Method
128 def find_duplicates_bruteforce(data):
129     duplicates = []
130     for i in range(len(data)):
131         for j in range(i + 1, len(data)):
132             if data[i] == data[j] and data[i] not in duplicates:
133                 duplicates.append(data[i])
134     return duplicates
135 user_ids = [101, 203, 405, 101, 203, 509]
136 duplicates = find_duplicates_bruteforce(user_ids)
137 print("Duplicates (Brute Force):", duplicates)
138 #Optimized Method Using Set
139 def find_duplicates_optimized(data):
140     seen = set()
141     duplicates = set()
142     for item in data:
143         if item in seen:
144             duplicates.add(item)
145         else:
146             seen.add(item)
147     return list(duplicates)
148 user_ids = [101, 203, 405, 101, 203, 509]
149 duplicates = find_duplicates_optimized(user_ids)
150 print("Duplicates (Optimized):", duplicates)
151
```

PS C:\Users\Likitha\OneDrive\Desktop\AI> & C:\Users\Likitha\AppData\Local\Python\python3.14-64\python.exe c:/Users/Likitha/OneDrive/Desktop/AI/AI-ASS12.4.PY
Duplicates (Brute Force): [101, 203]
Duplicates (Optimized): [203, 101]

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.