

# ASSIGNMENT-12.4

Name: E.Ramya

Ht.no: 2303A51282

Batch: 05

## **Task 1:** Bubble Sort for Ranking Exam Scores

### Scenario

You are working on a college result processing system where a small list of student scores needs to be sorted after every internal assessment.

### Task Description

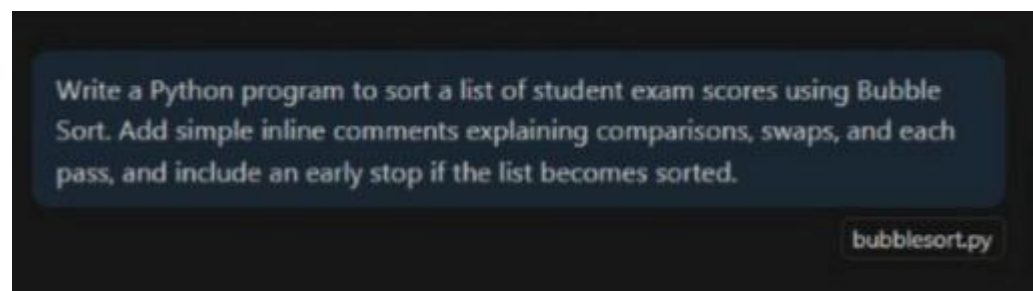
- Implement Bubble Sort in Python to sort a list of student scores.
- Use an AI tool to:

Insert inline comments explaining key operations such as comparisons, swaps, and iteration passes Identify early termination

conditions when the list becomes sorted  
Provide a brief time complexity analysis.

## **Prompt:**

Write a Python program to sort a list of student exam scores using Bubble Sort. Add simple inline comments explaining comparisons, swaps, and each pass, and include an early stop if the list becomes sorted.



## **Code:**

```

bubblesort.py > bubble_sort
def bubble_sort(scores):
    """
    Sort a list of exam scores using Bubble Sort with early stopping.

    Args:
        scores: List of integers representing exam scores

    Returns:
        Sorted list in ascending order
    """
    n = len(scores)

    # Iterate through each pass
    for i in range(n):
        # Flag to detect if any swaps occur in this pass
        swapped = False

        # Compare adjacent elements
        for j in range(0, n - i - 1):
            # Compare current score with next score
            if scores[j] > scores[j + 1]:
                # Swap if current score is greater
                scores[j], scores[j + 1] = scores[j + 1], scores[j]
                swapped = True

        # Early stop: If no swaps occurred, list is already sorted
        if not swapped:
            break

    return scores

# Example usage
if __name__ == "__main__":
    exam_scores = [85, 92, 78, 95, 82, 88, 76, 90]

    print("Original scores:", exam_scores)
    sorted_scores = bubble_sort(exam_scores)
    print("Sorted scores:", sorted_scores)

```

## Output:

```

PS C:\AI Assistant Coding> & C:/Users/edula/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/AI Assistant Coding/bubblesort.py"
Original scores: [85, 92, 78, 95, 82, 88, 76, 90]
Sorted scores: [76, 78, 82, 85, 88, 90, 92, 95]
PS C:\AI Assistant Coding>

```

## Explanation:

- Bubble Sort compares two adjacent scores in the list.
- If the left score is greater than the right score, they are swapped.
- This process repeats for all elements in one pass.
- After each pass, the largest score moves to the end of the list.
- Multiple passes are done until the list becomes sorted.

- If no swaps happen in a pass, the algorithm stops early because the list is already sorted.

Time Complexity:

- Best Case:  $O(n)$  – when the list is already sorted.
- Average Case:  $O(n^2)$
- Worst Case:  $O(n^2)$  – when the list is in reverse order.

## **Task 2: Improving Sorting for Nearly Sorted Attendance Records**

### Scenario

You are maintaining an attendance system where student roll numbers are already almost sorted, with only a few late updates.

### Task Description

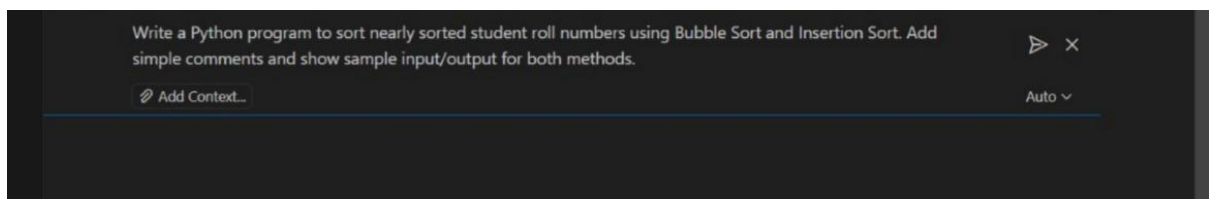
- Start with a Bubble Sort implementation.
- Ask AI to:

Review the problem and suggest a more suitable sorting algorithm  
Generate an Insertion Sort implementation  
Explain why Insertion Sort performs better on nearly sorted data.

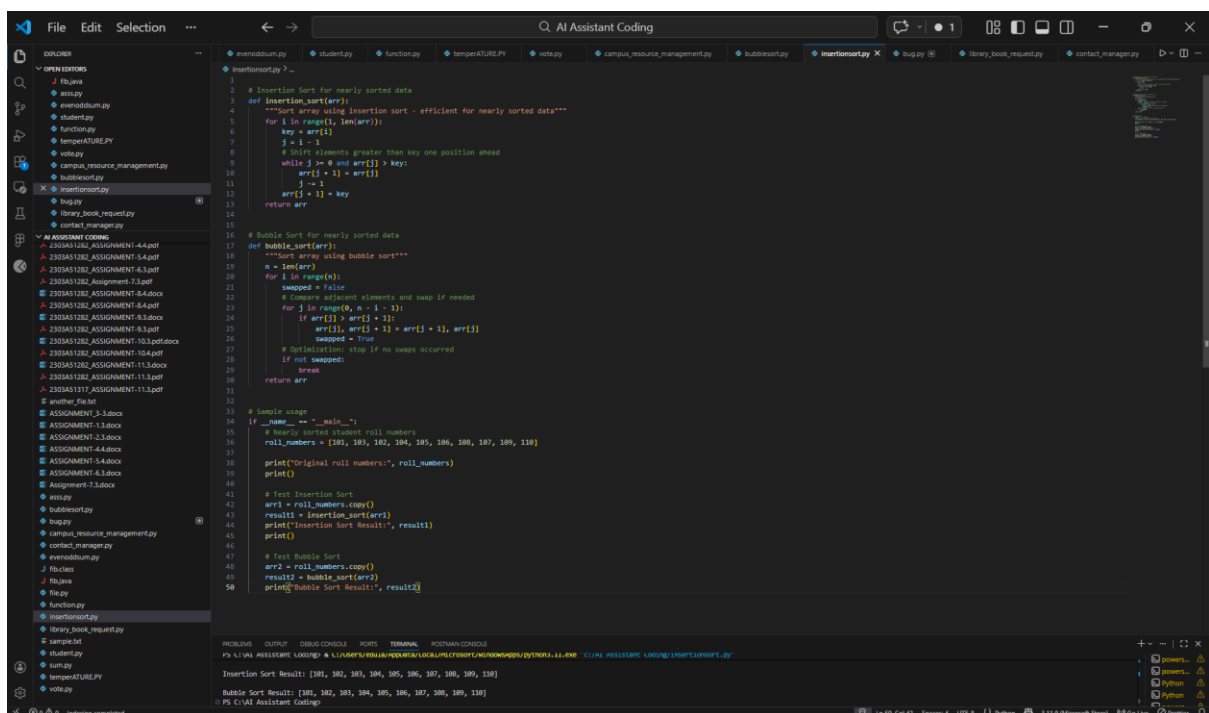
- Compare execution behavior on nearly sorted input

## Prompt:

Write a Python program to sort nearly sorted student roll numbers using Bubble Sort and Insertion Sort. Add simple comments and show sample input/output for both methods.



## Code:



## Output:

```
PS C:\AI Assistant Coding > & C:/Users/edala/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/AI Assistant Coding/bubblesort.py"
PS C:\AI Assistant Coding > & C:/Users/edala/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/AI Assistant Coding/insertionsort.py"
Original roll numbers: [101, 103, 102, 104, 105, 106, 100, 107, 109, 110]
Insertion Sort Result: [101, 102, 103, 104, 105, 106, 107, 100, 109, 110]
Bubble Sort Result: [101, 102, 103, 104, 105, 106, 107, 100, 109, 110]
PS C:\AI Assistant Coding >
```

## Explanation:

The program sorts nearly sorted student roll numbers using two methods: Bubble Sort and Insertion Sort. Bubble Sort repeatedly compares and swaps adjacent elements until the list is sorted, while Insertion Sort places each element in its correct position by shifting larger elements to the right. In nearly sorted data, Insertion Sort is faster because only a few elements need shifting, so it completes in fewer operations compared to Bubble Sort.

## Task 3: Searching Student Records in a Database Scenario

You are developing a student information portal where users search for student records by roll number.

### Task Description

- Implement:

Linear Search for unsorted student data

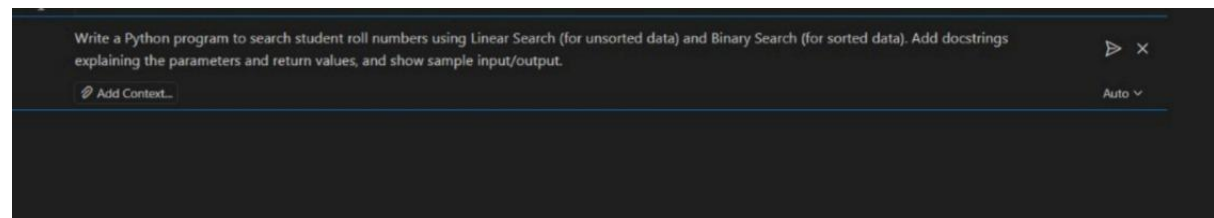
## Binary Search for sorted student data

- Use AI to:

Add docstrings explaining parameters and return values ,Explain when Binary Search is applicable Highlight performance differences between the two searches

### **Prompt:**

Write a Python program to search student roll numbers using Linear Search (for unsorted data) and Binary Search (for sorted data). Add docstrings explaining the parameters and return values, and show sample input/output.



### **Code:**

```

1 def linear_search(roll_numbers, target):
2     """
3     Search for a target roll number using Linear Search.
4
5     Parameters:
6     roll_numbers (list): List of student roll numbers (unsorted).
7     target (int): The roll number to search for.
8
9     Returns:
10    int: Index of the target if found, otherwise -1.
11    """
12    for i in range(len(roll_numbers)):
13        if roll_numbers[i] == target:
14            return i
15    return -1
16
17
18 def binary_search(roll_numbers, target):
19     """
20     Search for a target roll number using Binary Search.
21
22     Parameters:
23     roll_numbers (list): List of student roll numbers (sorted).
24     target (int): The roll number to search for.
25
26     Returns:
27     int: Index of the target if found, otherwise -1.
28     """
29    left, right = 0, len(roll_numbers) - 1
30
31    while left <= right:
32        mid = (left + right) // 2
33        if roll_numbers[mid] == target:
34            return mid
35        elif roll_numbers[mid] < target:
36            left = mid + 1
37        else:
38            right = mid - 1
39    return -1
40
41
42 # Sample Input/Output
43 if __name__ == "__main__":
44     # Unsorted data for Linear Search
45     unsorted_rolls = [105, 103, 101, 104, 102]
46     print("=== Linear Search (Unsorted Data) ===")
47     print("Roll numbers: (unsorted_rolls)")
48     result = linear_search(unsorted_rolls, 104)
49     print("Searching for 104: Found at index (result)\n")
50
51     # Sorted data for Binary Search
52     sorted_rolls = [101, 102, 103, 104, 105]
53     print("=== Binary Search (Sorted Data) ===")
54     print("Roll numbers: (sorted_rolls)")
55     result = binary_search(sorted_rolls, 104)
56     print("Searching for 104: Found at index (result)")
57     result = binary_search(sorted_rolls, 110)
58     print("Searching for 110: Found at index (result)\n")

```

## Output:

```

PS C:\AI Assistant Coding> & C:/Users/edula/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/AI Assistant Coding/searching.py"
=== Linear Search (Unsorted Data) ===
Roll numbers: [105, 103, 101, 104, 102]
Searching for 104: Found at index 3

=== Binary Search (Sorted Data) ===
Roll numbers: [101, 102, 103, 104, 105]
Searching for 104: Found at index 3
Searching for 110: Found at index -1
PS C:\AI Assistant Coding>

```

## Explanation:

- The program uses Linear Search to find a roll number in an unsorted list by checking each element one by one.
- It uses Binary Search for a sorted list by repeatedly dividing the list into halves until the roll number is found or not found.

- Binary Search is faster because it reduces the search space each step, while Linear Search checks all elements in the worst case.

Time Complexity:

- Linear Search: Best  $O(1)$ , Worst  $O(n)$
- Binary Search: Best  $O(1)$ , Worst  $O(\log n)$

#### **Task 4:** Choosing Between Quick Sort and Merge Sort for Data Processing Scenario

You are part of a data analytics team that needs to sort large datasets received from different sources (random order, already sorted, and reverse sorted).

Task Description

- Provide AI with partially written recursive functions for:

Quick Sort

Merge Sort

- Ask AI to:

Complete the recursive logic

Add meaningful docstrings

Explain how recursion works in each algorithm

- Test both algorithms on:

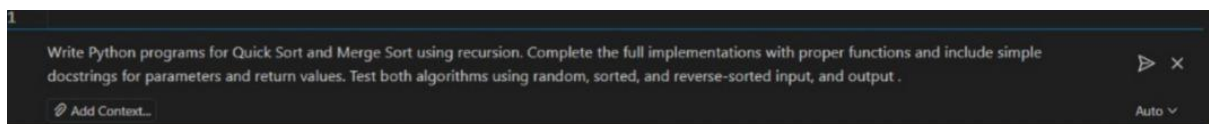
Random data

Sorted data

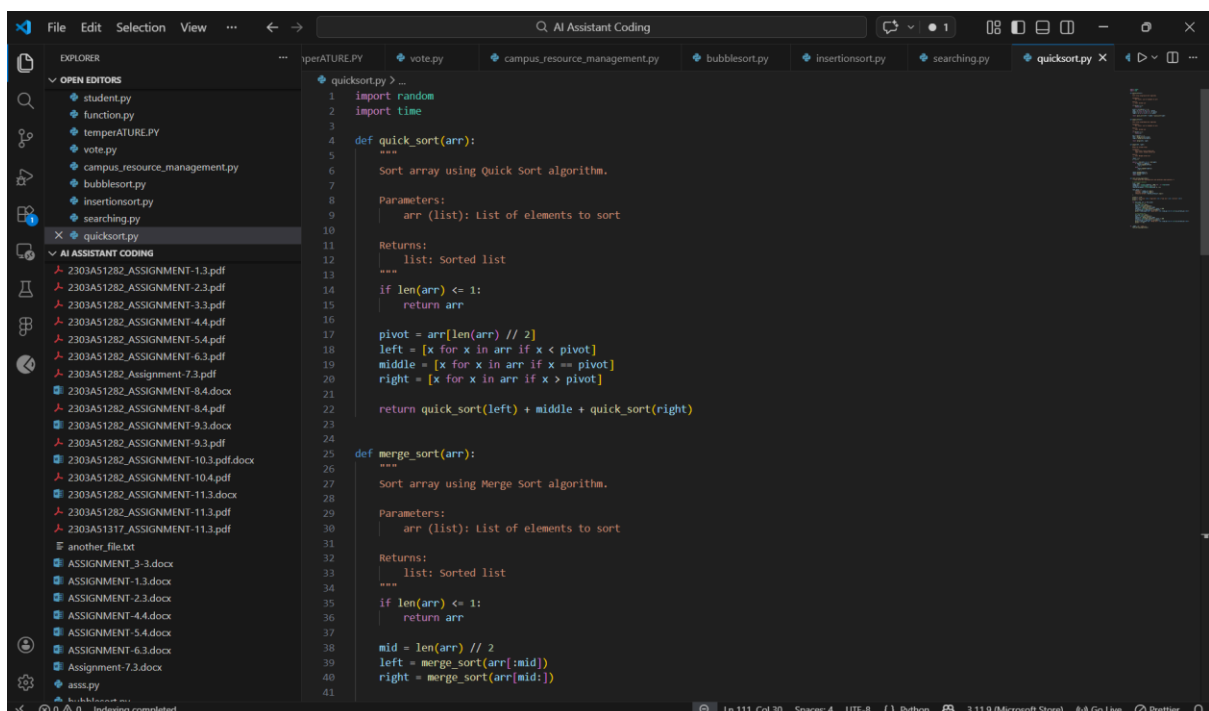
Reverse-sorted data

## Prompt:

Write Python programs for Quick Sort and Merge Sort using recursion. Complete the full implementations with proper functions and include simple docstrings for parameters and return values. Test both algorithms using random, sorted, and reverse-sorted input, and output .



## Code:



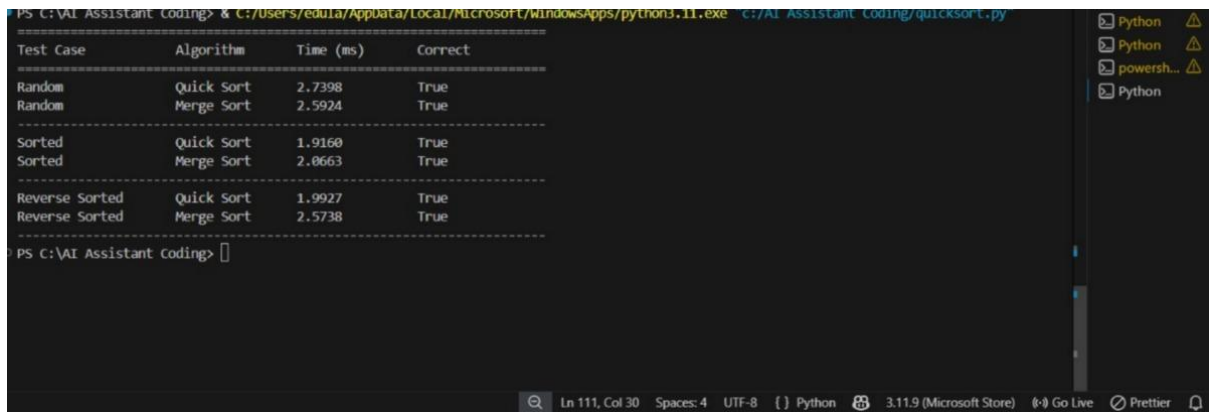
The screenshot shows the VS Code editor with the file explorer on the left. The 'EXPLORER' panel lists several files, including 'quicksort.py' which is selected. The 'AI ASSISTANT CODING' panel is also visible. The main editor window displays the code for 'quicksort.py'. The code defines a 'merge\_sort' function that uses a 'merge' function to sort an array. The 'merge' function takes two sorted arrays and merges them into a single sorted array. The 'merge\_sort' function uses a recursive approach to sort the array. The code is as follows:

```
25 def merge_sort(arr):
26     return merge(left, right)
27
28 def merge(left, right):
29     """
30     Merge two sorted arrays.
31
32     Parameters:
33     left (list): First sorted list
34     right (list): Second sorted list
35
36     Returns:
37     list: Merged sorted list
38     """
39     result = []
40     i = j = 0
41
42     while i < len(left) and j < len(right):
43         if left[i] <= right[j]:
44             result.append(left[i])
45             i += 1
46         else:
47             result.append(right[j])
48             j += 1
49
50     result.extend(left[i:])
51     result.extend(right[j:])
52     return result
53
54 def test_sorting_algorithms():
55     """Test Quick Sort and Merge Sort with different input patterns."""
56
57     # Create test datasets
58     size = 1000
59     random_arr = [random.randint(1, 1000) for _ in range(size)]
60     sorted_arr = list(range(1, size + 1))
61     reverse_sorted_arr = list(range(size, 0, -1))
62
63     test_cases = [
```

The screenshot shows the VS Code editor with the file explorer on the left. The 'EXPLORER' panel lists several files, including 'quicksort.py' which is selected. The 'AI ASSISTANT CODING' panel is also visible. The main editor window displays the code for 'quicksort.py'. The code defines a 'test\_sorting\_algorithms' function that tests the 'merge\_sort' function with different input patterns. The code is as follows:

```
72 def test_sorting_algorithms():
73     """Test Quick Sort and Merge Sort with different input patterns."""
74
75     # Create test datasets
76     size = 1000
77     random_arr = [random.randint(1, 1000) for _ in range(size)]
78     sorted_arr = list(range(1, size + 1))
79     reverse_sorted_arr = list(range(size, 0, -1))
80
81     test_cases = [
82         ("Random", random_arr.copy()),
83         ("Sorted", sorted_arr.copy()),
84         ("Reverse Sorted", reverse_sorted_arr.copy())
85     ]
86
87     print("\n" * 70)
88     print(f"{'Test Case':<20} {'Algorithm':<15} {'Time (ms)':<15} {'Correct':<10}")
89     print("\n" * 70)
90
91     for test_name, arr in test_cases:
92         # Test Quick Sort
93         arr_copy = arr.copy()
94         start = time.perf_counter()
95         result_qs = quick_sort(arr_copy)
96         time_qs = (time.perf_counter() - start) * 1000
97         is_correct_qs = result_qs == sorted(arr)
98         print(f"{'Test Case':<20} {'Quick Sort':<15} {'Time (ms)':<15.4f} {'Correct':<10}")
99
100         # Test Merge Sort
101         arr_copy = arr.copy()
102         start = time.perf_counter()
103         result_ms = merge_sort(arr_copy)
104         time_ms = (time.perf_counter() - start) * 1000
105         is_correct_ms = result_ms == sorted(arr)
106         print(f"{'Test Case':<20} {'Merge Sort':<15} {'Time (ms)':<15.4f} {'Correct':<10}")
107         print("\n" * 70)
108
109
110 if __name__ == "__main__":
111     test_sorting_algorithms()
```

Output:



Test Case	Algorithm	Time (ms)	Correct
Random	Quick Sort	2.7398	True
Random	Merge Sort	2.5924	True
Sorted	Quick Sort	1.9160	True
Sorted	Merge Sort	2.0663	True
Reverse Sorted	Quick Sort	1.9927	True
Reverse Sorted	Merge Sort	2.5738	True

## Explanation:

- Both programs use recursion, meaning the function calls itself to sort smaller parts of the list.
- Quick Sort selects a pivot element and places smaller elements on the left and larger elements on the right, then recursively sorts both sides.
- Merge Sort divides the list into two halves, recursively sorts each half, and then merges the sorted halves together.

## Time Complexity:

- Quick Sort: Best/Average  $O(n \log n)$ , Worst  $O(n^2)$
- Merge Sort: Best/Average/Worst  $O(n \log n)$

## **Task 5: Optimizing a Duplicate Detection Algorithm**

### **Scenario**

You are building a data validation module that must detect duplicate user IDs in a large dataset before importing it into a system.

### **Task Description**

- Write a naive duplicate detection algorithm using nested loops.
- Use AI to:

Analyze the time complexity

Suggest an optimized approach using sets or dictionaries

Rewrite the algorithm with improved efficiency

- Compare execution behavior conceptually for large input sizes

### **Prompt:**

Write a Python program to detect duplicate user IDs using a nested loop (brute-force) method.

Analyze the time complexity and then suggest a faster approach using a set or dictionary. Rewrite the program using the optimized method and show both versions of the code for comparison.

Write a Python program to detect duplicate user IDs using a nested loop (brute-force) method. Analyze the time complexity and then suggest a faster approach using a set or dictionary. Rewrite the program using the optimized method and show both versions of the code for comparison.

## Code:

```
1 # time complexity: O(n^2)
2 # Space Complexity: O(1)
3
4
5 def find_duplicates_brute_force(user_ids):
6     """
7     Detect duplicates using nested loops.
8     For each element, compare with all other elements.
9     """
10    duplicates = []
11    for i in range(len(user_ids)):
12        for j in range(i + 1, len(user_ids)):
13            if user_ids[i] == user_ids[j] and user_ids[i] not in duplicates:
14                duplicates.append(user_ids[i])
15    return duplicates
16
17
18 # ===== OPTIMIZED APPROACH =====
19 # Time Complexity: O(n)
20 # Space Complexity: O(n)
21
22 def find_duplicates_optimized(user_ids):
23     """
24     Detect duplicates using a set for O(1) lookups.
25     Single pass through the list.
26     """
27    seen = set()
28    duplicates = set()
29    for user_id in user_ids:
30        if user_id in seen:
31            duplicates.add(user_id)
32        else:
33            seen.add(user_id)
34    return list(duplicates)
35
36
37 # ===== COMPARISON & TESTING =====
38
39 if __name__ == "__main__":
40     # Test data
41     test_users = [101, 102, 103, 102, 104, 101, 105, 103]
42
43     print("User IDs:", test_users)
44     print("\nBrute Force Result:", find_duplicates_brute_force(test_users))
45     print("Optimized Result:", find_duplicates_optimized(test_users))
46
47     print("\n--- Time Complexity Analysis ---")
48     print("Brute Force: O(n^2) - nested loops compare each pair")
49     print("Optimized: O(n) - single pass with set lookup O(1)")
```

## Output:

```
PS C:\AI Assistant Coding> & C:/Users/edula/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/AI Assistant Coding/detect.py"
User IDs: [101, 102, 103, 102, 104, 101, 105, 103]

Brute Force Result: [101, 102, 103]
Optimized Result: [101, 102, 103]

--- Time Complexity Analysis ---
Brute Force: O(n^2) - nested loops compare each pair
Optimized: O(n) - single pass with set lookup O(1)
```

## Explanation:

- In the brute-force method, the program checks each user ID with every other ID to find duplicates, so it takes more time.

- In the optimized method, the program uses a set to remember IDs and quickly detect if an ID appears again.
- The optimized method is faster because it scans the list only once, so it works better for large data.