AI ASS 12.4

Name: A.Harshita

H.T.NO: 2303A52211

Batch:43

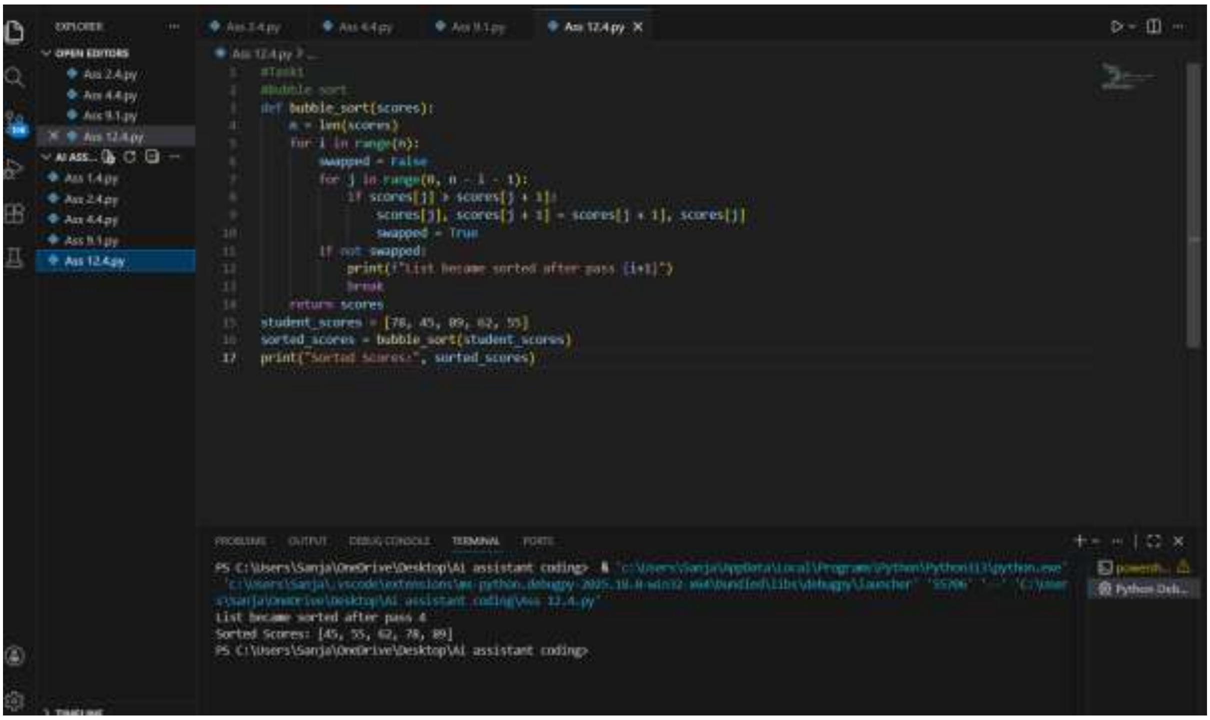
Task1:

Write a Python program to implement Bubble Sort for sorting student exam scores.

Add inline comments explaining comparisons, swaps, and passes.

Include early termination using a swapped flag.

Provide sample input/output and briefly explain best, average, and worst-case time complexity.



```python
#Task1
#Bubble sort
def bubble_sort(scores):
    n = len(scores)
    for i in range(n):
        swapped = False
        for j in range(0, n - i - 1):
            if scores[j] > scores[j + 1]:
                scores[j], scores[j + 1] = scores[j + 1], scores[j]
                swapped = True
        if not swapped:
            print(f"List became sorted after pass {i+1}")
            break
    return scores
student_scores = [78, 45, 89, 62, 55]
sorted_scores = bubble_sort(student_scores)
print("Sorted Scores:", sorted_scores)
```

```
PS C:\Users\Sanja\OneDrive\Desktop\Ai assistant coding> & 'c:\Users\Sanja\AppData\Local\Programs\Python\Python313\python.exe' 'c:\Users\Sanja\.vscode\extensions\ms-python.debugpy-2025.18.0-win32-x64\bundled\libs\debugpy\launcher' '55706' '--' 'C:\Users\Sanja\OneDrive\Desktop\Ai assistant coding\Ass 12.4.py'
List became sorted after pass 4
Sorted Scores: [45, 55, 62, 78, 89]
PS C:\Users\Sanja\OneDrive\Desktop\Ai assistant coding>
```

Observation:

Bubble Sort was implemented to sort student scores after internal assessments.

The algorithm shows O(n) time complexity in the best case when the list is already sorted due to early termination.
In the average case, when scores are randomly arranged, the time complexity is O(n²).
In the worst case, when scores are in reverse order, it also takes O(n²) time due to maximum comparisons and swaps.
Thus, Bubble Sort is suitable for small datasets but not efficient for large datasets.

Task2:

I have a nearly sorted list of student roll numbers (attendance records).

1. Implement Bubble Sort in Python.

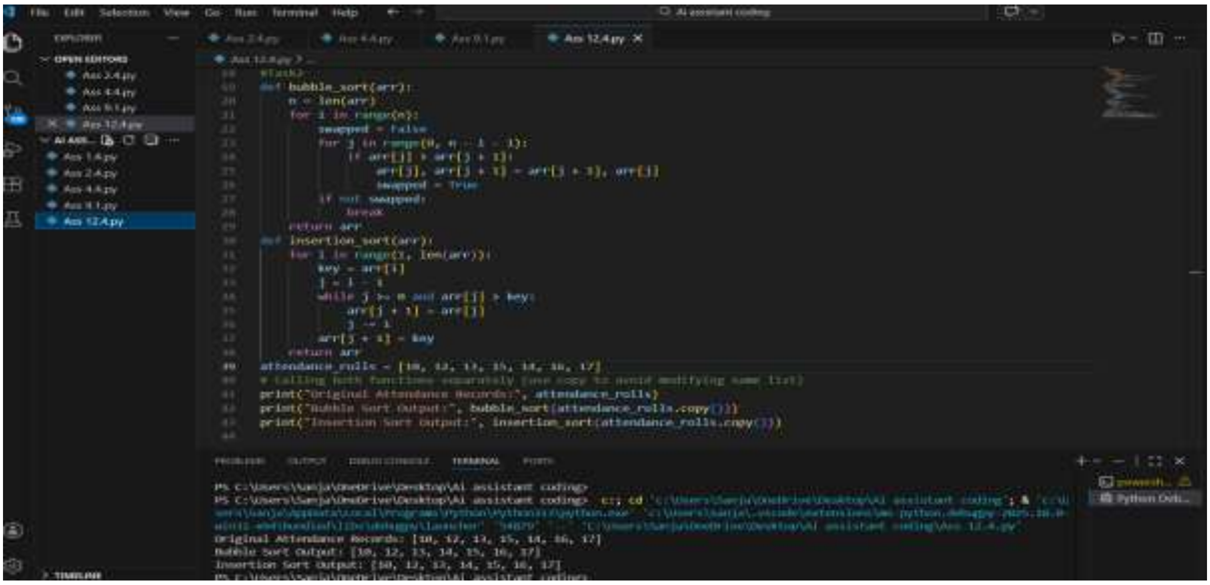
2. Suggest a more suitable sorting algorithm for nearly sorted data.

3. Implement Insertion Sort in Python.

4. Explain why Insertion Sort performs better on nearly sorted datasets.

5. Compare their behavior on a nearly sorted input example.

6. Include time complexity analysis.



```python
#Task2
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        swapped = False
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True
        if not swapped:
            break
    return arr
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
    return arr
attendance_rolls = [10, 12, 13, 15, 14, 16, 17]
# Calling both functions separately (use copy to avoid modifying same list)
print("Original Attendance Records:", attendance_rolls)
print("Bubble Sort Output:", bubble_sort(attendance_rolls.copy()))
print("Insertion Sort Output:", insertion_sort(attendance_rolls.copy()))
```

```
PS C:\Users\Sanja\OneDrive\Desktop\AI assistant coding>
PS C:\Users\Sanja\OneDrive\Desktop\AI assistant coding>  c:; cd 'c:\Users\Sanja\OneDrive\Desktop\AI assistant coding'; & 'c:\Users\sanja\AppData\Local\Programs\Python\Python313\python.exe' 'c:\Users\sanja\.vscode\extensions\ms-python.debugpy-2025.18.0-win32-x64\bundled\libs\debugpy\launcher' '54879' '--' 'C:\Users\Sanja\OneDrive\Desktop\AI assistant coding\Ass 12.4.py'
Original Attendance Records: [10, 12, 13, 15, 14, 16, 17]
Bubble Sort Output: [10, 12, 13, 14, 15, 16, 17]
Insertion Sort Output: [10, 12, 13, 14, 15, 16, 17]
PS C:\Users\Sanja\OneDrive\Desktop\AI assistant coding>
```

Observation:

1.The attendance roll numbers were nearly sorted with only a few misplaced elements.

2.Bubble Sort performed multiple comparisons even when only small corrections were needed.

3.Insertion Sort shifted only the misplaced elements to their correct positions.

4.For nearly sorted data, Insertion Sort behaved close to **O(n)** time complexity.

5. Bubble Sort still required more passes, making it comparatively slower.

6.Therefore, Insertion Sort is more efficient and suitable for partially sorted attendance records.

Task3:

Building a student information portal.

1. Implement Linear Search for unsorted student roll numbers.

2. Implement Binary Search for sorted roll numbers.

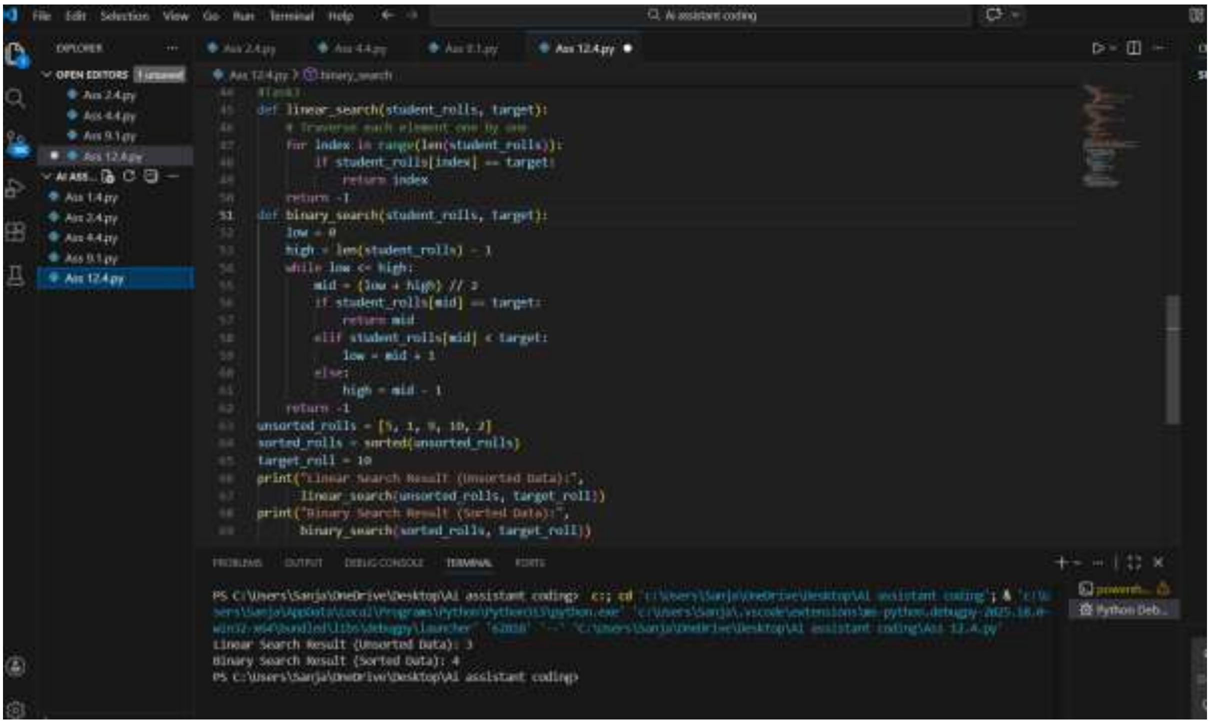
3. Add proper Python docstrings explaining parameters and return values.

4. Explain when Binary Search can be used.

5. Compare time complexity and performance differences.

6. Provide sample input and output.

7. Add a short observation comparing results on sorted vs unsorted data.



```python
#Task3
def linear_search(student_rolls, target):
    # Traverse each element one by one
    for index in range(len(student_rolls)):
        if student_rolls[index] == target:
            return index
    return -1
def binary_search(student_rolls, target):
    low = 0
    high = len(student_rolls) - 1
    while low <= high:
        mid = (low + high) // 2
        if student_rolls[mid] == target:
            return mid
        elif student_rolls[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1
unsorted_rolls = [5, 1, 9, 10, 2]
sorted_rolls = sorted(unsorted_rolls)
target_roll = 10
print("Linear Search Result (Unsorted Data):",
      linear_search(unsorted_rolls, target_roll))
print("Binary Search Result (Sorted Data):",
      binary_search(sorted_rolls, target_roll))
```

```
PS C:\Users\Sanja\OneDrive\Desktop\AI assistant coding>  c:; cd 'c:\Users\Sanja\OneDrive\Desktop\AI assistant coding'; & 'c:\U
sers\Sanja\AppData\Local\Programs\Python\Python313\python.exe' 'c:\Users\Sanja\.vscode\extensions\ms-python.debugpy-2025.18.0-
win32-x64\bundled\libs\debugpy\launcher' '62016' '--' 'C:\Users\Sanja\OneDrive\Desktop\AI assistant coding\Ass 12.4.py'
Linear Search Result (Unsorted Data): 3
Binary Search Result (Sorted Data): 4
PS C:\Users\Sanja\OneDrive\Desktop\AI assistant coding>
```

Observation:

Linear Search checks each element one by one, so its time complexity is **O(n)** in average and worst cases. It works on both sorted and unsorted lists.

Binary Search divides the sorted list into halves repeatedly, so its time complexity is **O(log n)**. It works only on sorted data.

For unsorted lists, Linear Search is used.
For large sorted lists, Binary Search is faster and more efficient.

Task4:

work with large datasets that may be random, already sorted, or reverse sorted.

1. Complete recursive implementations of Quick Sort and Merge Sort in Python.

2. Add proper docstrings explaining parameters and return values.

3. Explain how recursion works in both algorithms.

4. Test both algorithms on:

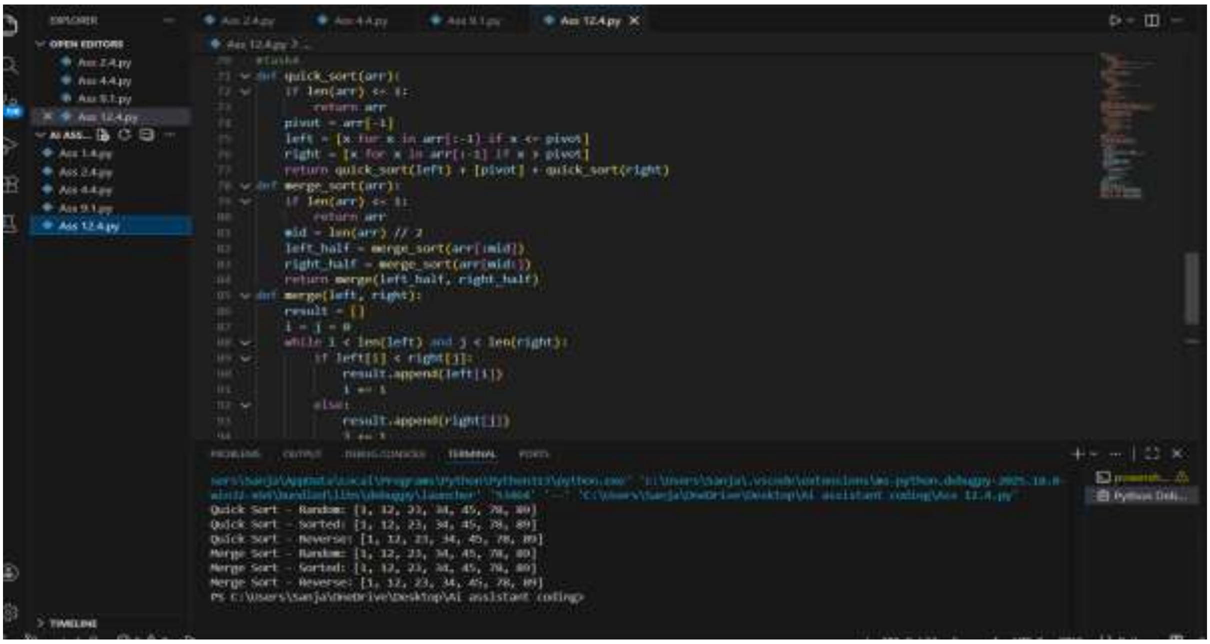
   - Random data

   - Sorted data

   - Reverse sorted data

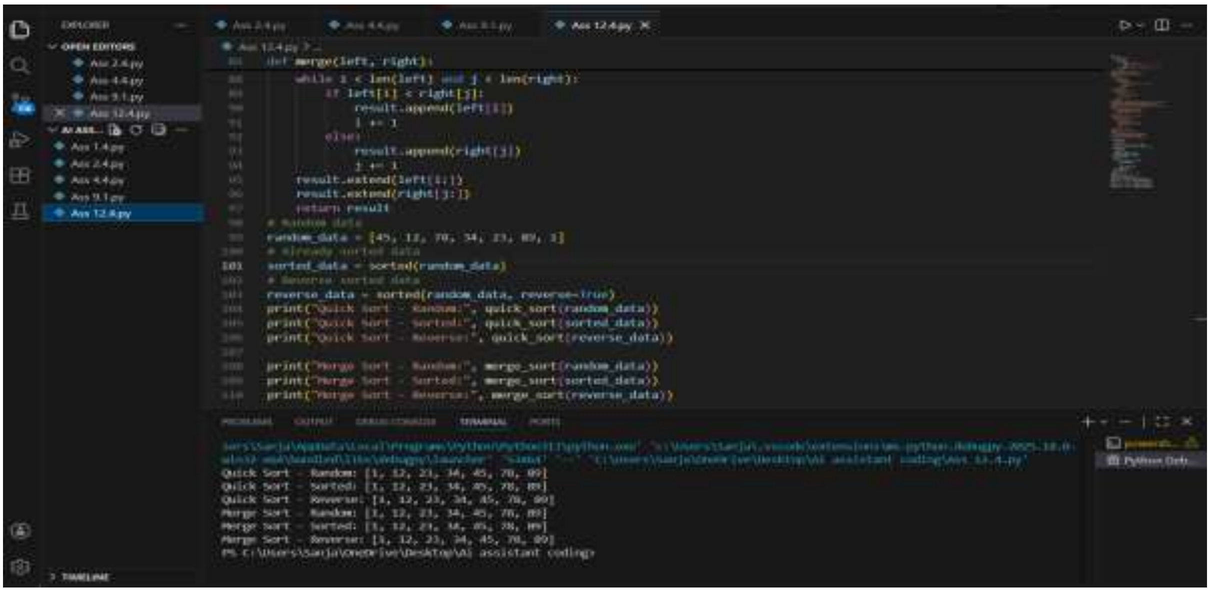
5. Compare best, average, and worst-case time complexity.

6. Explain practical scenarios where one is preferred over the other.



```python
#task4
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[-1]
    left = [x for x in arr[:-1] if x <= pivot]
    right = [x for x in arr[:-1] if x > pivot]
    return quick_sort(left) + [pivot] + quick_sort(right)
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left_half = merge_sort(arr[:mid])
    right_half = merge_sort(arr[mid:])
    return merge(left_half, right_half)
def merge(left, right):
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
```

```
sers\Sanja\AppData\Local\Programs\Python\Python313\python.exe' 'c:\Users\Sanja\.vscode\extensions\ms-python.debugpy-2025.10.0-
win32-x64\bundled\libs\debugpy\launcher' '51604' '--' 'C:\Users\Sanja\OneDrive\Desktop\AI assistant coding\Ass 12.4.py'
Quick Sort - Random: [1, 12, 23, 34, 45, 78, 89]
Quick Sort - Sorted: [1, 12, 23, 34, 45, 78, 89]
Quick Sort - Reverse: [1, 12, 23, 34, 45, 78, 89]
Merge Sort - Random: [1, 12, 23, 34, 45, 78, 89]
Merge Sort - Sorted: [1, 12, 23, 34, 45, 78, 89]
Merge Sort - Reverse: [1, 12, 23, 34, 45, 78, 89]
PS C:\Users\Sanja\OneDrive\Desktop\AI assistant coding>
```



```python
def merge(left, right):
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result
# Random data
random_data = [45, 12, 78, 34, 23, 89, 1]
# Already sorted data
sorted_data = sorted(random_data)
# Reverse sorted data
reverse_data = sorted(random_data, reverse=True)
print("Quick Sort - Random:", quick_sort(random_data))
print("Quick Sort - Sorted:", quick_sort(sorted_data))
print("Quick Sort - Reverse:", quick_sort(reverse_data))

print("Merge Sort - Random:", merge_sort(random_data))
print("Merge Sort - Sorted:", merge_sort(sorted_data))
print("Merge Sort - Reverse:", merge_sort(reverse_data))
```

```
sers\Sanja\AppData\Local\Programs\Python\Python313\python.exe' 'c:\Users\Sanja\.vscode\extensions\ms-python.debugpy-2025.10.0-
win32-x64\bundled\libs\debugpy\launcher' '51604' '--' 'C:\Users\Sanja\OneDrive\Desktop\AI assistant coding\Ass 12.4.py'
Quick Sort - Random: [1, 12, 23, 34, 45, 78, 89]
Quick Sort - Sorted: [1, 12, 23, 34, 45, 78, 89]
Quick Sort - Reverse: [1, 12, 23, 34, 45, 78, 89]
Merge Sort - Random: [1, 12, 23, 34, 45, 78, 89]
Merge Sort - Sorted: [1, 12, 23, 34, 45, 78, 89]
Merge Sort - Reverse: [1, 12, 23, 34, 45, 78, 89]
PS C:\Users\Sanja\OneDrive\Desktop\AI assistant coding>
```

Observation:

1.Quick Sort has best and average time complexity of O(n log n).

2.Quick Sort worst case is $O(n^2)$ when pivot selection is poor.

3.Merge Sort has O(n log n) time complexity in best, average, and worst cases.

4.Quick Sort uses less memory (in-place), while Merge Sort requires extra memory.

5.Quick Sort is usually faster for random datasets.

6.Merge Sort is preferred when guaranteed performance and stable sorting are required.

Task5:

Building a data validation module to detect duplicate user IDs.

1. Write a brute-force duplicate detection algorithm using nested loops.

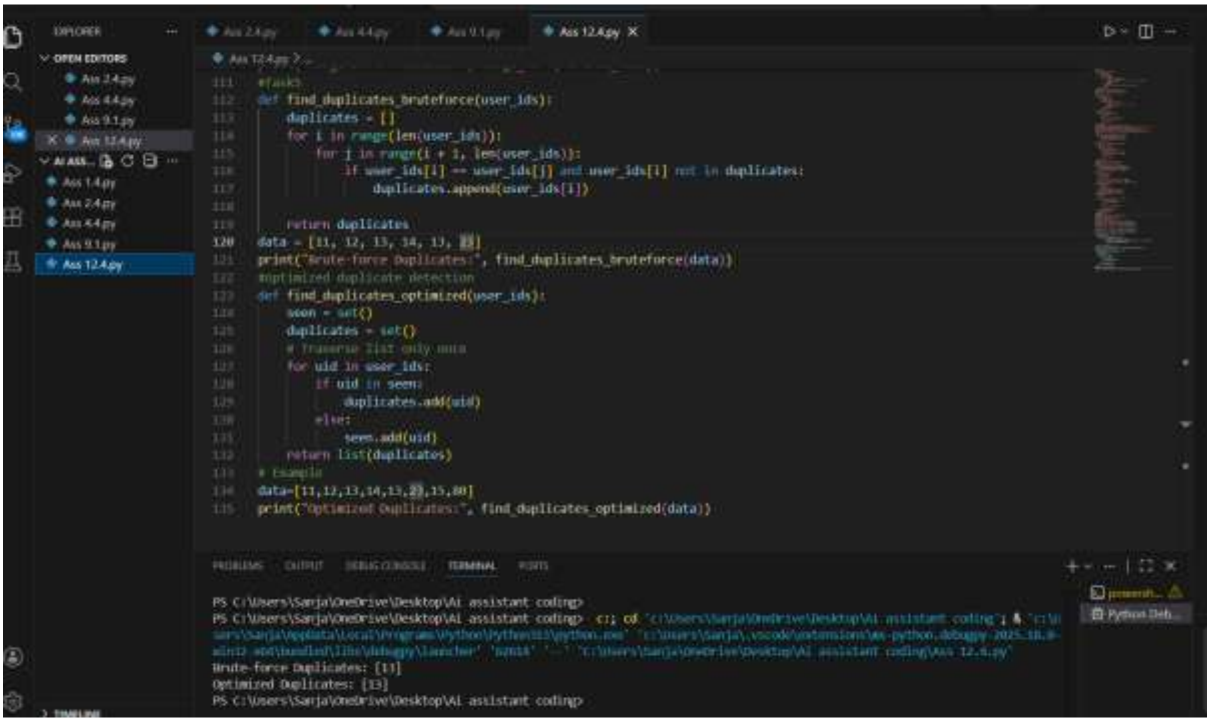
2. Analyze its time complexity.

3. Suggest an optimized approach using a set or dictionary.

4. Rewrite the algorithm with improved efficiency.

5. Compare performance conceptually for large datasets.

6. Keep explanation simple and clear.



```python
#task5
def find_duplicates_bruteforce(user_ids):
    duplicates = []
    for i in range(len(user_ids)):
        for j in range(i + 1, len(user_ids)):
            if user_ids[i] == user_ids[j] and user_ids[i] not in duplicates:
                duplicates.append(user_ids[i])

    return duplicates
data = [11, 12, 13, 14, 13, 23]
print("Brute-force Duplicates:", find_duplicates_bruteforce(data))
#optimized duplicate detection
def find_duplicates_optimized(user_ids):
    seen = set()
    duplicates = set()
    # Traverse list only once
    for uid in user_ids:
        if uid in seen:
            duplicates.add(uid)
        else:
            seen.add(uid)
    return list(duplicates)
# Example
data=[11,12,13,14,13,23,15,80]
print("Optimized Duplicates:", find_duplicates_optimized(data))
```

```
PS C:\Users\Sanja\OneDrive\Desktop\Ai assistant coding>
PS C:\Users\Sanja\OneDrive\Desktop\Ai assistant coding>  c:; cd 'c:\Users\Sanja\OneDrive\Desktop\Ai assistant coding'; & 'c:\Users\Sanja\AppData\Local\Programs\Python\Python313\python.exe' 'c:\Users\Sanja\.vscode\extensions\ms-python.debugpy-2025.18.0-win32-x64\bundled\libs\debugpy\launcher' '[illegible]' '--' 'c:\Users\Sanja\OneDrive\Desktop\Ai assistant coding\Ass 12.4.py'
Brute-force Duplicates: [13]
Optimized Duplicates: [13]
PS C:\Users\Sanja\OneDrive\Desktop\Ai assistant coding>
```

Observation:

1.The brute-force algorithm uses nested loops to compare every element with all others, resulting in O(n²) time complexity.

2.It performs a large number of unnecessary comparisons, especially for large datasets.

3.The optimized algorithm uses a set (or dictionary) to store already seen elements.

4.Since set lookup takes O(1) time, the entire list is processed in a single loop, giving O(n) time complexity.

5.The performance improved because repeated comparisons were replaced with fast lookups.

6.Therefore, the optimized approach is much faster and more suitable for large datasets.