Assignment-12.4

Name: Arjun Manoj

H.No:2303A52134

Batch:44

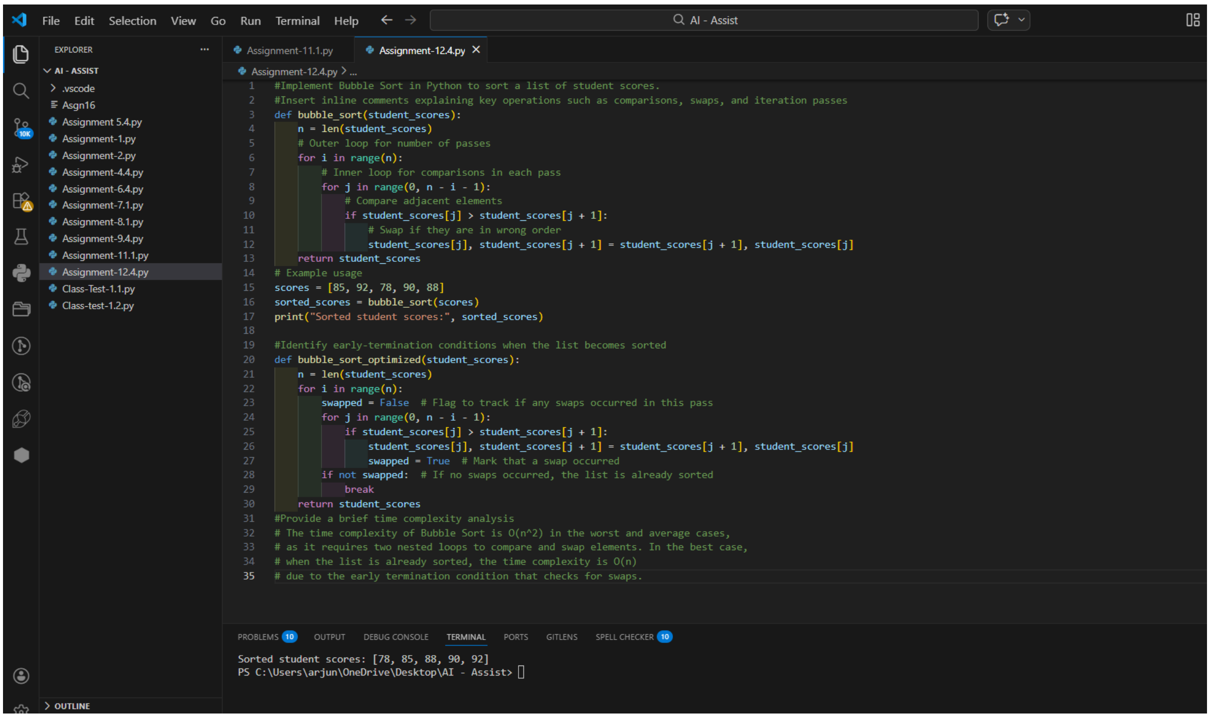
Task 1: Bubble Sort for Ranking Exam Scores

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

Prompt:

Implement Bubble Sort in Python to sort a list of student exam scores.
Include inline comments explaining comparisons, swaps, and iteration passes.
Also implement an optimized version with early termination and analyze time complexity.

Code:



Observation:

Bubble Sort repeatedly compares adjacent elements and swaps them if they are in the wrong order.

After each pass, the largest unsorted element moves to its correct position.

In the optimized version, if no swaps occur during a pass, the algorithm stops early (early termination).

- Time Complexity:
- Best Case: O(n) (already sorted list with optimization)
- Average Case: $O(n^2)$
- Worst Case: $O(n^2)$

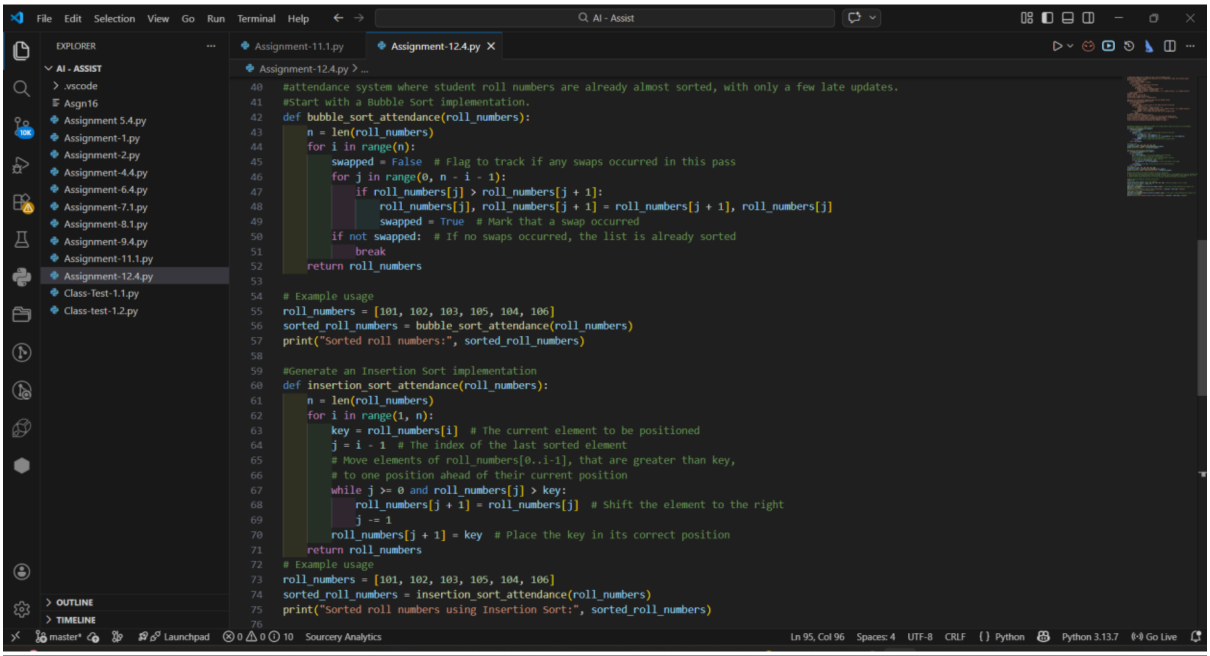
Task 2: Improving Sorting for Nearly Sorted Attendance Records

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

Prompt:

Implement Bubble Sort and Insertion Sort in Python to sort student roll numbers in an attendance system where the data is nearly sorted. Compare their performance on nearly sorted input and explain why one performs better.

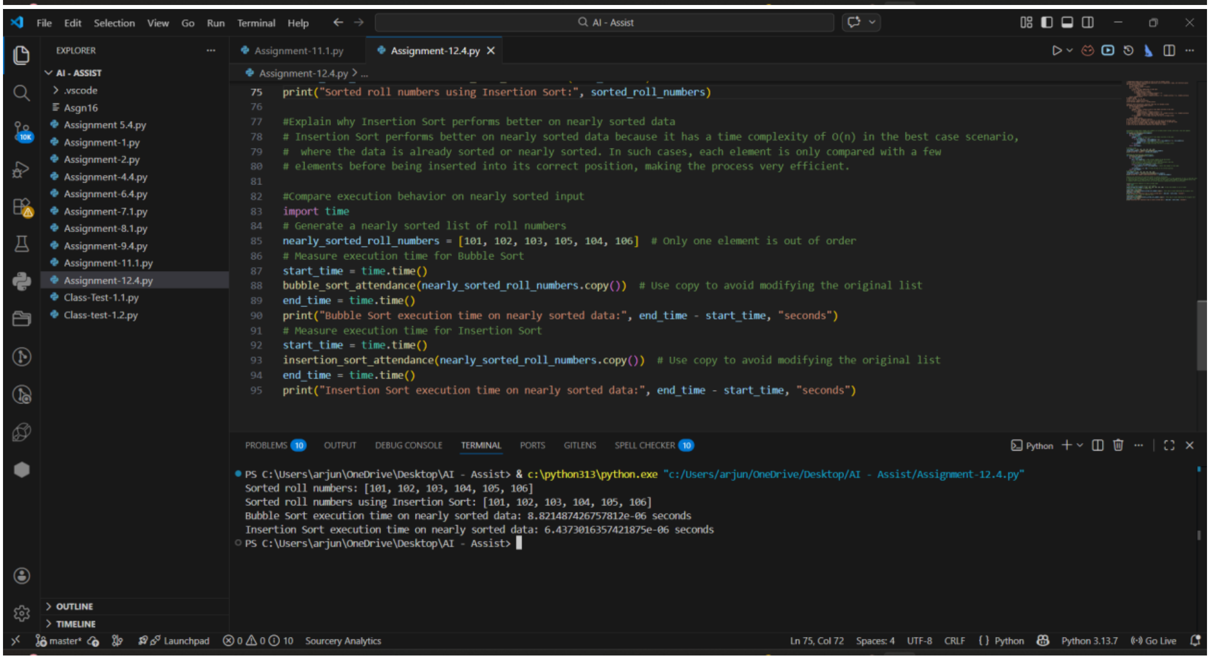
Code:



```python
#attendance system where student roll numbers are already almost sorted, with only a few late updates.
#Start with a Bubble Sort implementation.
def bubble_sort_attendance(roll_numbers):
    n = len(roll_numbers)
    for i in range(n):
        swapped = False  # Flag to track if any swaps occurred in this pass
        for j in range(0, n - i - 1):
            if roll_numbers[j] > roll_numbers[j + 1]:
                roll_numbers[j], roll_numbers[j + 1] = roll_numbers[j + 1], roll_numbers[j]
                swapped = True  # Mark that a swap occurred
        if not swapped:  # If no swaps occurred, the list is already sorted
            break
    return roll_numbers

# Example usage
roll_numbers = [101, 102, 103, 105, 104, 106]
sorted_roll_numbers = bubble_sort_attendance(roll_numbers)
print("Sorted roll numbers:", sorted_roll_numbers)

#Generate an Insertion Sort implementation
def insertion_sort_attendance(roll_numbers):
    n = len(roll_numbers)
    for i in range(1, n):
        key = roll_numbers[i]  # The current element to be positioned
        j = i - 1  # The index of the last sorted element
        # Move elements of roll_numbers[0..i-1], that are greater than key,
        # to one position ahead of their current position
        while j >= 0 and roll_numbers[j] > key:
            roll_numbers[j + 1] = roll_numbers[j]  # Shift the element to the right
            j -= 1
        roll_numbers[j + 1] = key  # Place the key in its correct position
    return roll_numbers
# Example usage
roll_numbers = [101, 102, 103, 105, 104, 106]
sorted_roll_numbers = insertion_sort_attendance(roll_numbers)
print("Sorted roll numbers using Insertion Sort:", sorted_roll_numbers)
```



```python
print("Sorted roll numbers using Insertion Sort:", sorted_roll_numbers)

#Explain why Insertion Sort performs better on nearly sorted data
# Insertion Sort performs better on nearly sorted data because it has a time complexity of O(n) in the best case scenario,
#  where the data is already sorted or nearly sorted. In such cases, each element is only compared with a few
# elements before being inserted into its correct position, making the process very efficient.

#Compare execution behavior on nearly sorted input
import time
# Generate a nearly sorted list of roll numbers
nearly_sorted_roll_numbers = [101, 102, 103, 105, 104, 106]  # Only one element is out of order
# Measure execution time for Bubble Sort
start_time = time.time()
bubble_sort_attendance(nearly_sorted_roll_numbers.copy())  # Use copy to avoid modifying the original list
end_time = time.time()
print("Bubble Sort execution time on nearly sorted data:", end_time - start_time, "seconds")
# Measure execution time for Insertion Sort
start_time = time.time()
insertion_sort_attendance(nearly_sorted_roll_numbers.copy())  # Use copy to avoid modifying the original list
end_time = time.time()
print("Insertion Sort execution time on nearly sorted data:", end_time - start_time, "seconds")
```

```
PS C:\Users\arjun\OneDrive\Desktop\AI - Assist> & c:\python313\python.exe "c:/Users/arjun/OneDrive/Desktop/AI - Assist/Assignment-12.4.py"
Sorted roll numbers: [101, 102, 103, 104, 105, 106]
Sorted roll numbers using Insertion Sort: [101, 102, 103, 104, 105, 106]
Bubble Sort execution time on nearly sorted data: 8.821487426757812e-06 seconds
Insertion Sort execution time on nearly sorted data: 6.4373016357421875e-06 seconds
PS C:\Users\arjun\OneDrive\Desktop\AI - Assist>
```

Observation:

Bubble Sort compares adjacent elements and swaps them repeatedly. Even with optimization, it may still require multiple passes.

Insertion Sort inserts each element into its correct position within the already sorted portion of the list.

For nearly sorted data, Insertion Sort performs fewer shifts and comparisons.

Time Complexity:

Bubble Sort:

Best Case: $O(n)$ (with early termination)

Average/Worst Case: $O(n^2)$

Insertion Sort:

Best Case: $O(n)$ (already/nearly sorted)

Average/Worst Case: $O(n^2)$

In practice, Insertion Sort executes faster than Bubble Sort for nearly sorted roll numbers because it minimizes unnecessary swaps.

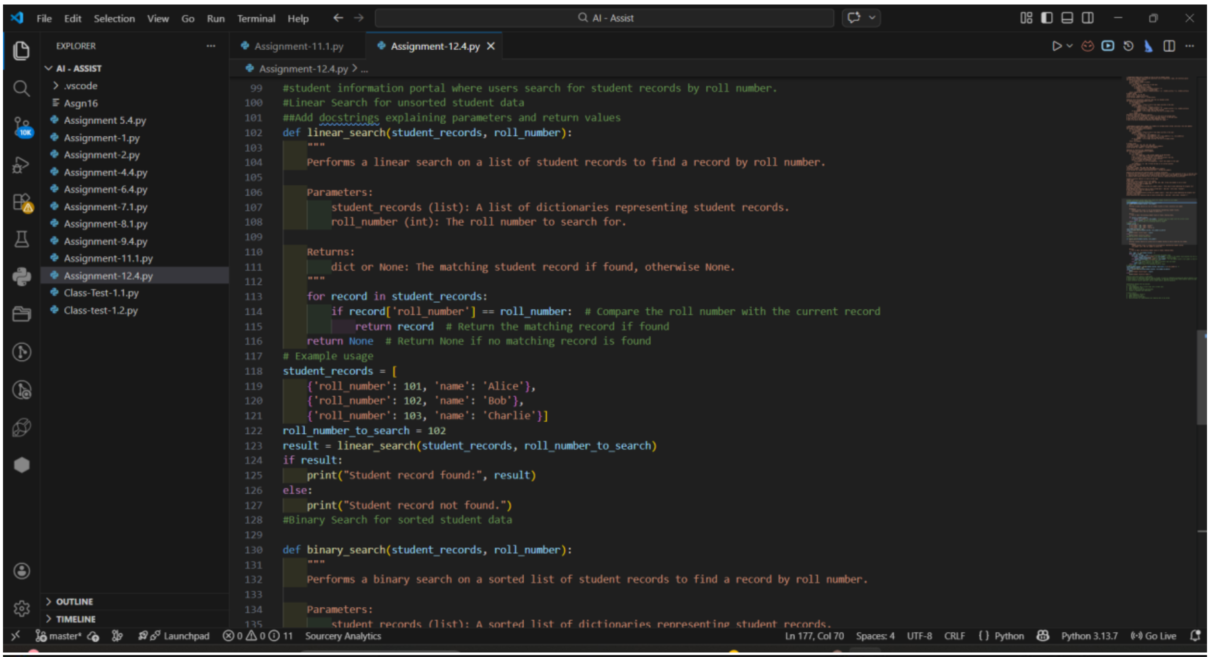
Task 3: Searching Student Records in a Database

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

Prompt:

Implement Linear Search and Binary Search in Python to search student records by roll number.
Add proper docstrings explaining parameters and return values.
Compare both searching techniques and explain when Binary Search is applicable.

Code:



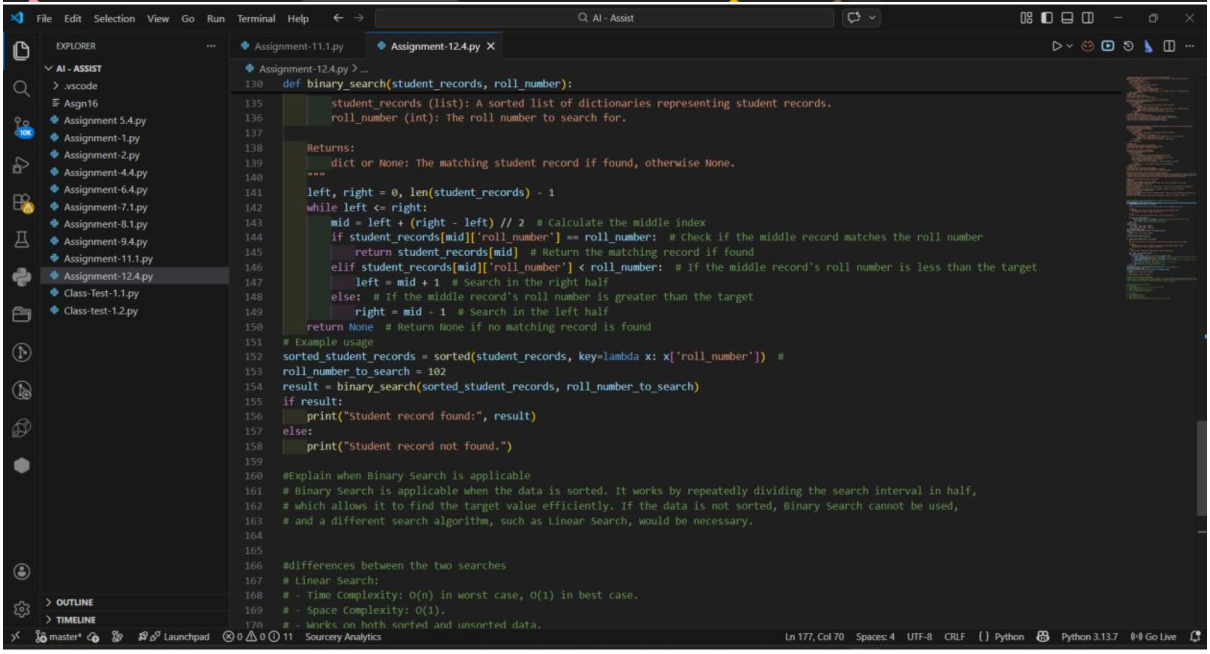
```python
#student information portal where users search for student records by roll number.
#Linear Search for unsorted student data
##Add docstrings explaining parameters and return values
def linear_search(student_records, roll_number):
    """
    Performs a linear search on a list of student records to find a record by roll number.

    Parameters:
        student_records (list): A list of dictionaries representing student records.
        roll_number (int): The roll number to search for.

    Returns:
        dict or None: The matching student record if found, otherwise None.
    """
    for record in student_records:
        if record['roll_number'] == roll_number:  # Compare the roll number with the current record
            return record  # Return the matching record if found
    return None  # Return None if no matching record is found
# Example usage
student_records = [
    {'roll_number': 101, 'name': 'Alice'},
    {'roll_number': 102, 'name': 'Bob'},
    {'roll_number': 103, 'name': 'Charlie'}]
roll_number_to_search = 102
result = linear_search(student_records, roll_number_to_search)
if result:
    print("Student record found:", result)
else:
    print("Student record not found.")
#Binary Search for sorted student data

def binary_search(student_records, roll_number):
    """
    Performs a binary search on a sorted list of student records to find a record by roll number.

    Parameters:
        student_records (list): A sorted list of dictionaries representing student records.
```

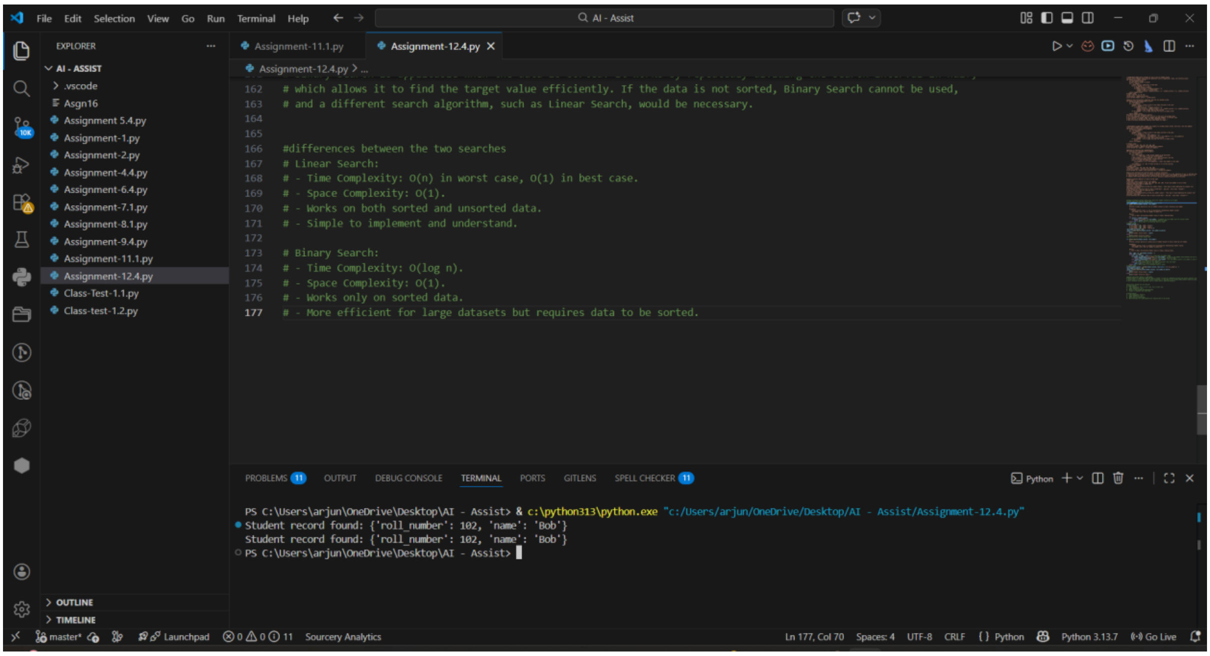


```python
def binary_search(student_records, roll_number):
        student_records (list): A sorted list of dictionaries representing student records.
        roll_number (int): The roll number to search for.

    Returns:
        dict or None: The matching student record if found, otherwise None.
    """
    left, right = 0, len(student_records) - 1
    while left <= right:
        mid = left + (right - left) // 2  # Calculate the middle index
        if student_records[mid]['roll_number'] == roll_number:  # Check if the middle record matches the roll number
            return student_records[mid]  # Return the matching record if found
        elif student_records[mid]['roll_number'] < roll_number:  # If the middle record's roll number is less than the target
            left = mid + 1  # Search in the right half
        else:  # If the middle record's roll number is greater than the target
            right = mid - 1  # Search in the left half
    return None  # Return None if no matching record is found
# Example usage
sorted_student_records = sorted(student_records, key=lambda x: x['roll_number'])  #
roll_number_to_search = 102
result = binary_search(sorted_student_records, roll_number_to_search)
if result:
    print("Student record found:", result)
else:
    print("Student record not found.")

#Explain when Binary Search is applicable
# Binary Search is applicable when the data is sorted. It works by repeatedly dividing the search interval in half,
# which allows it to find the target value efficiently. If the data is not sorted, Binary Search cannot be used,
# and a different search algorithm, such as Linear Search, would be necessary.


#differences between the two searches
# Linear Search:
# - Time Complexity: O(n) in worst case, O(1) in best case.
# - Space Complexity: O(1).
# - Works on both sorted and unsorted data.
```



Observation:

- Linear Search checks each student record one by one until the roll number is found.
- It works on both sorted and unsorted data.
- Time Complexity of Linear Search:
  - Best Case: O(1)
  - Worst Case: O(n)
- Binary Search repeatedly divides the sorted data into halves to find the roll number.
- It works only when the student records are sorted.
- Time Complexity of Binary Search:
  - Best/Worst Case: O(log n)
- For small or unsorted student data, Linear Search is suitable.
- For large and sorted datasets, Binary Search is much more efficient.

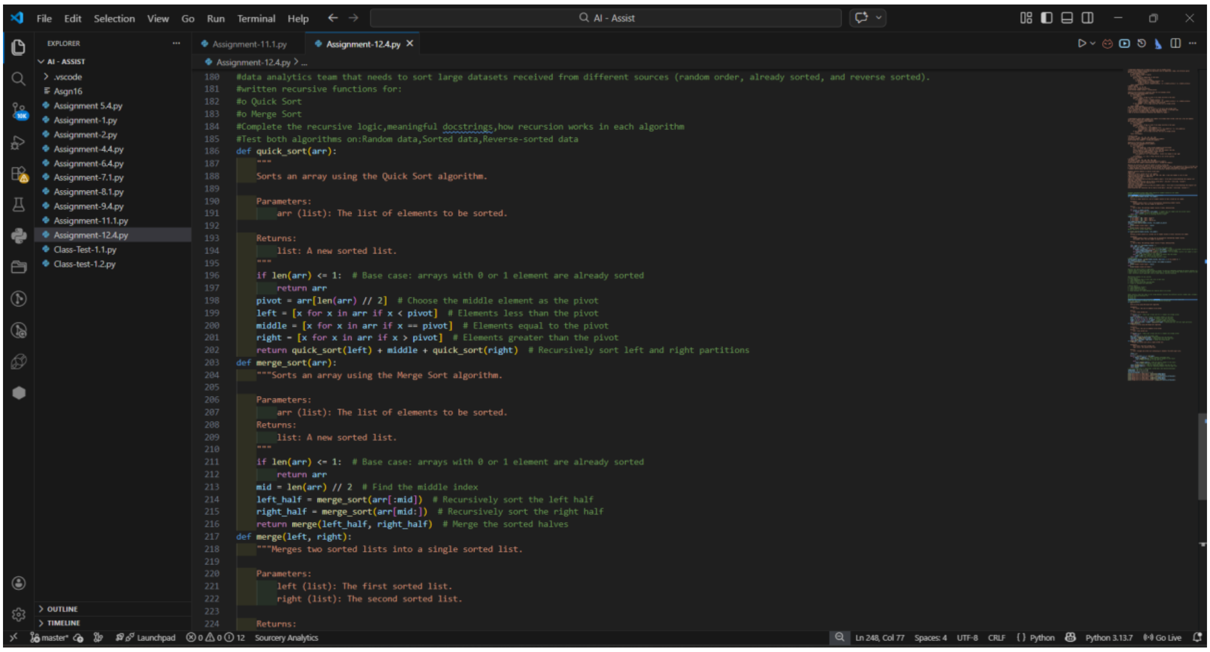
Task 4: Choosing Between Quick Sort and Merge Sort for Data Processing

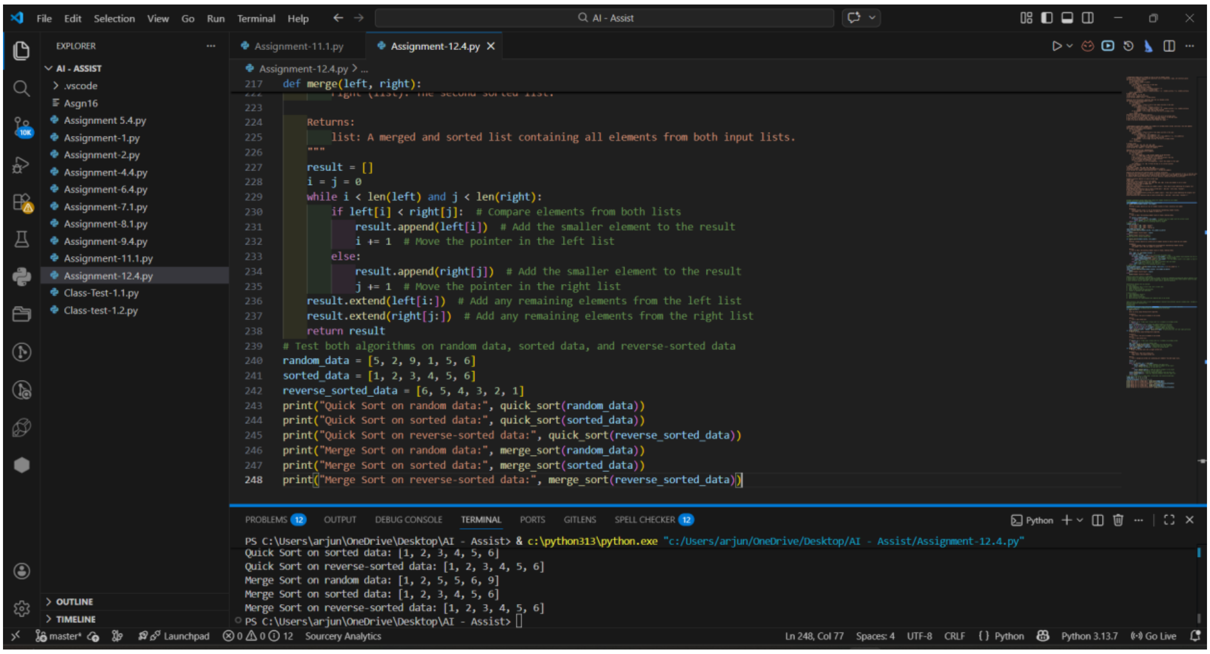
You are part of a data analytics team that needs to sort large datasets received from different sources (random order, already sorted, and reverse sorted).

Prompt:

Implement recursive Quick Sort and Merge Sort algorithms in Python to sort large datasets.
Add meaningful docstrings explaining parameters and return values.
Explain how recursion works in each algorithm.
Test both algorithms on random, already sorted, and reverse-sorted data.

Code:





Observation:

- Quick Sort and Merge Sort both use recursion and follow the Divide and Conquer approach.
- Quick Sort divides the list around a pivot and recursively sorts left and right parts.
- Merge Sort divides the list into halves and recursively merges them in sorted order.
- Merge Sort always runs in O(n log n) time.
- Quick Sort runs in O(n log n) on average but O(n²) in the worst case (bad pivot choice).

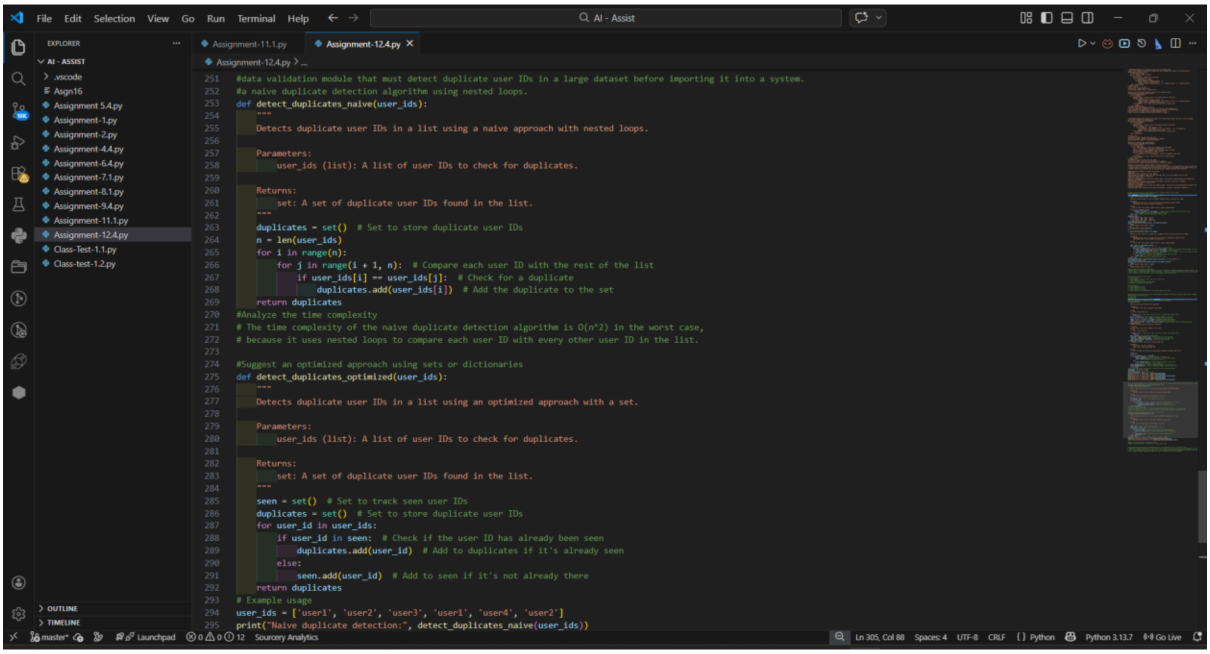
Task 5: Optimizing a Duplicate Detection Algorithm

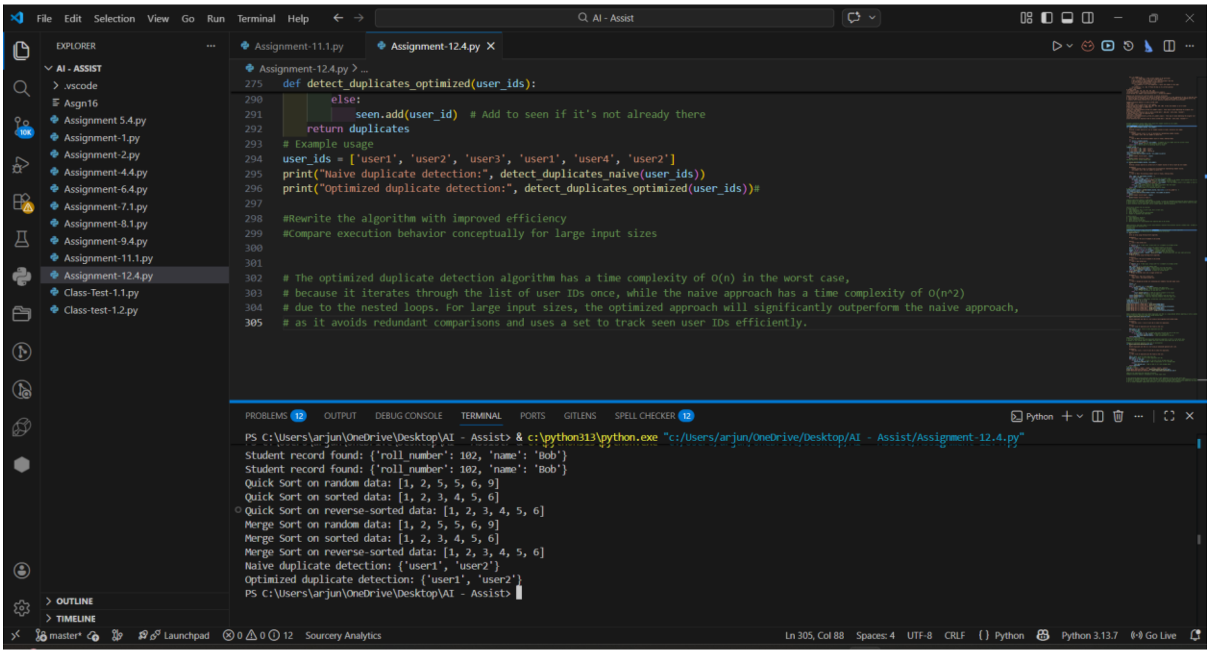
You are building a data validation module that must detect duplicate user IDs in a large dataset before importing it into a system.

Prompt:

Implement a naive duplicate detection algorithm using nested loops to detect duplicate user IDs.
Analyze its time complexity. Then implement an optimized version using sets and compare their performance conceptually for large datasets.

Code:





Observation:

- The naive approach uses nested loops and compares every pair of user IDs.
- Its time complexity is $O(n^2)$, which is inefficient for large datasets.
- The optimized approach uses a set to track seen user IDs.
- Its time complexity is $O(n)$ since it scans the list only once.
- For large input sizes, the optimized method performs significantly faster and avoids unnecessary comparisons.