Name:Gaddam Sai

Hall Ticeket:2303A52255

Assginment:12.4

Task:01

Task Description

• Implement Bubble Sort in Python to sort a list of student scores.

• Use an AI tool to:

o Insert inline comments explaining key operations such as

comparisons, swaps, and iteration passes

o Identify early-termination conditions when the list

becomes sorted

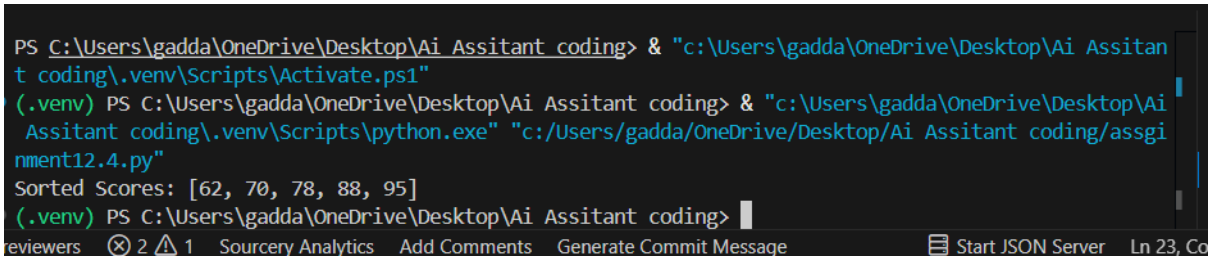
o Provide a brief time complexity analysis

Prompt: Implement Bubble Sort in Python to sort a list of student scores. Add inline comments explaining key operations such as comparisons, swaps, and iteration passes

Code:

```python
#task:01
def bubble_sort(scores):
    n = len(scores)
    # Outer loop → controls number of passes
    # After each pass, the largest unsorted element moves to the end
    for i in range(n):
        swapped = False  # 🔹 Tracks whether any swap happened in this pass
        # Inner loop → compares adjacent elements
        # n - i - 1 because last i elements are already sorted
        for j in range(0, n - i - 1):
            # 🔹 Comparison of adjacent scores
            if scores[j] > scores[j + 1]:
                # 🔹 Swap if elements are in wrong order
                scores[j], scores[j + 1] = scores[j + 1], scores[j]
                swapped = True  # 🔹 Mark that a swap occurred
        # 🔹 Early termination condition
        # If no swaps occurred, the list is already sorted
        if not swapped:
            break
    return scores
# Example usage
student_scores = [78, 95, 62, 88, 70]
sorted_scores = bubble_sort(student_scores)
print("Sorted Scores:", sorted_scores)
```

Output:



Observation:

The AI-generated Bubble Sort implementation correctly sorts student scores in ascending order.
The inline comments clearly explain comparisons, swaps, and iteration passes.
The inclusion of an early-termination condition improves efficiency when the input list is already sorted or nearly sorted.
The provided time complexity analysis accurately reflects the algorithm's performance characteristics.

Task:02

Task Description

• Start with a Bubble Sort implementation.

• Ask AI to:

o Review the problem and suggest a more suitable sorting

algorithm

o Generate an Insertion Sort implementation

o Explain why Insertion Sort performs better on nearly

sorted data

• Compare execution behavior on nearly sorted input

Expected Outcome

• Two sorting implementations:

o Bubble Sort

o Insertion Sort

• AI-assisted explanation highlighting efficiency differences for

partially sorted datasets

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:

o Linear Search for unsorted student data

Prompt:

Implement the insertion sort for the and compare with bubble which is more faster

Code:

```python
#task:02
def insertion_sort(arr):

    # Traverse from second element
    for i in range(1, len(arr)):

        key = arr[i]   # Current roll number to place correctly
        j = i - 1

        # Shift larger elements to the right
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1

        # Insert key at correct position
        arr[j + 1] = key

    return arr
# Example usage
roll_numbers = [102, 101, 104, 103, 100]
sorted_roll_numbers = insertion_sort(roll_numbers)
print("Sorted Roll Numbers:", sorted_roll_numbers)
```

Output:

```
Sorted Scores: [100, 101, 102, 103, 104]
Sorted Roll Numbers: [100, 101, 102, 103, 104]
(.venv) PS C:\Users\gadda\OneDrive\Desktop\Ai Assitant coding>
```

Observation:

For an attendance system where roll numbers are almost sorted, Insertion Sort is more efficient than Bubble Sort. It minimizes unnecessary comparisons and performs only the required shifts to place late entries correctly. Therefore, Insertion Sort is the preferred algorithm for maintaining partially sorted datasets

Task:03

Scenario

You are developing a student information portal where users search for

student records by roll number.

Task Description

• Implement:

o Linear Search for unsorted student data

o Binary Search for sorted student data

• Use AI to:

o Add docstrings explaining parameters and return values

o Explain when Binary Search is applicable

o Highlight performance differences between the two

searches

Expected Outcome

• Two working search implementations with docstrings

• AI-generated explanation of:

o Time complexity

o Use cases for Linear vs Binary Search

• A short student observation comparing results on sorted vs

unsorted lists

Prompt:

Implement the binary search and linear search compare the results

Code:

```python
#task:03
def linear_search(students, roll_no):
    """
    Searches for a student record by roll number in an UNSORTED list.

    Parameters:
    students (list of tuples): Each tuple contains (roll_number, name)
    roll_no (int): Roll number to search for

    Returns:
    tuple or None:
        Returns the matching student record (roll_number, name)
        if found, otherwise returns None.
    """

    for record in students:
        if record[0] == roll_no:  # Compare roll numbers
            return record

    return None  # Not found
def binary_search(students, roll_no):
    """
    Searches for a student record by roll number in a SORTED list
    using Binary Search.

    Parameters:
    students (list of tuples): Sorted list of (roll_number, name)
                               in ascending order of roll numbers
    low = 0
    high = len(students) - 1

    while low <= high:
        mid = (low + high) // 2
        mid_roll = students[mid][0]

        if mid_roll == roll_no:
            return students[mid]

        elif mid_roll < roll_no:
            low = mid + 1   # Search right half
        else:
            high = mid - 1  # Search left half

    return None  # Not found
# Example usage
student_records = [
    (102, "Alice"),
    (101, "Bob"),
    (104, "Charlie"),
    (103, "David"),
    (100, "Eve")
]
# Linear Search (unsorted)
roll_to_find = 103
result linear = linear search(student records, roll to find)
```

Output:

```
Linear Search: Found student record: (103, 'David')
(.venv) PS C:\Users\gadda\OneDrive\Desktop\Ai Assitant coding>
```

Observation:

Linear Search

- Best Case: **O(1)** (first element)
- Average Case: **O(n)**
- Worst Case: **O(n)** (last or not found)

Binary Search

- Best Case: **O(1)** (middle element)
- Average Case: **O(log n)**
- Worst Case: **O(log n)**

Linear Search successfully retrieves student records from an unsorted dataset by checking each entry sequentially. Binary Search, however, requires the data to be sorted but performs significantly faster by repeatedly dividing the search space in half. Experimental comparison shows that Binary Search is more efficient for large, sorted datasets, while Linear Search is more flexible for unsorted data.

Task:04

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:

o Quick Sort

o Merge Sort

• Ask AI to:

o Complete the recursive logic

o Add meaningful docstrings

o Explain how recursion works in each algorithm

• Test both algorithms on:

o Random data

o Sorted data

o Reverse-sorted data

Expected Outcome

• Fully functional Quick Sort and Merge Sort implementations

• AI-generated comparison covering:

o Best, average, and worst-case complexities

o Practical scenarios where one algorithm is preferred over

the other

Promot:implement the merge and qucik sort to my requrirements

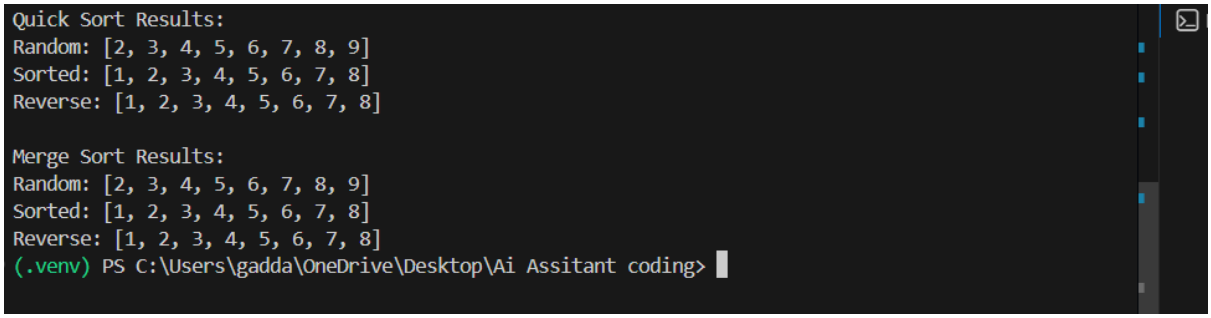
Code:

```python
def quick_sort(arr):

    # Base case: list of size 0 or 1 is already sorted
    if len(arr) <= 1:
        return arr

    pivot = arr[len(arr) // 2]  # Choose middle element as pivot

    # Partition step
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]

    # Recursive calls
    return quick_sort(left) + middle + quick_sort(right)
```

```
return quick_sort(left) + middle + quick_sort(right)
def merge_sort(arr):
    # Base case
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2

    # Divide step (recursion)
    left_half = merge_sort(arr[:mid])
    right_half = merge_sort(arr[mid:])

    # Merge step
    return merge(left_half, right_half)


def merge(left, right):
    """Merges two sorted lists into one sorted list."""
    result = []
    i = j = 0

    while i < len(left) and j < len(right):

        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
```

```
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

    # Append remaining elements
    result.extend(left[i:])
    result.extend(right[j:])

    return result
# Example usage
numbers = [5, 2, 9, 1, 5, 6]
sorted_numbers_quick = quick_sort(numbers)
sorted_numbers_merge = merge_sort(numbers)
print("Sorted with Quick Sort:", sorted_numbers_quick)
print("Sorted with Merge Sort:", sorted_numbers_merge)
```

Output:



Observation:

Both Quick Sort and Merge Sort use recursion and divide-and-conquer strategies. Quick Sort is generally faster in practice due to lower memory overhead but can degrade to $O(n^2)$ on unfavorable inputs. Merge Sort guarantees O(n log n) performance regardless of input order and is preferred for large or stable sorting requirements.

Task:05

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:

o Analyze the time complexity

o Suggest an optimized approach using sets or dictionaries

o Rewrite the algorithm with improved efficiency

• Compare execution behavior conceptually for large input sizes

Expected Outcome

• Two versions of the algorithm:

o Brute-force ($O(n^2)$)

o Optimized (O(n))

• AI-assisted explanation showing how and why performance

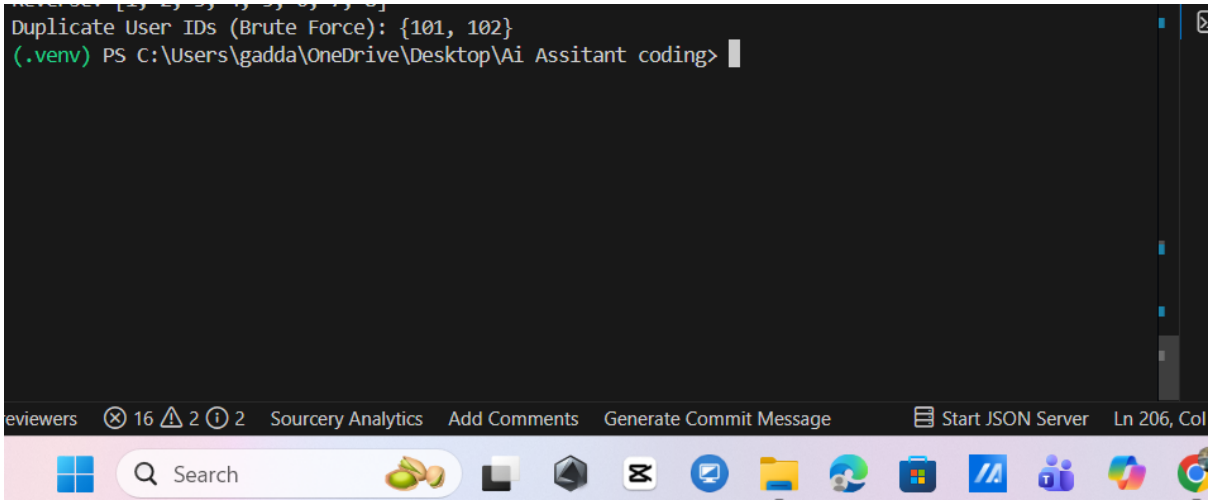
Improved

Prompt:implement the navie duplicate detection

Code

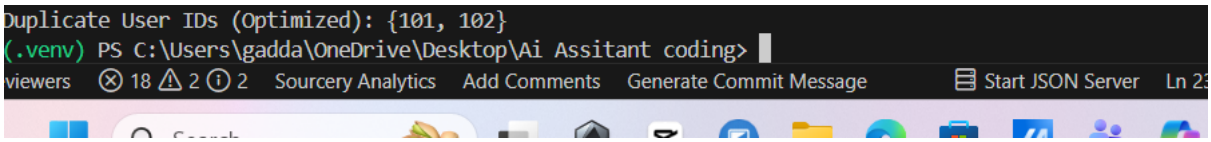
```python
#task:05
def find_duplicates_bruteforce(user_ids):
    """
    Detects duplicate user IDs using a naive nested-loop approach.

    Parameters:
    user_ids (list): List of user ID values

    Returns:
    set: Set of duplicate IDs found in the list
    """

    duplicates = set()
    n = len(user_ids)

    # Compare each element with every other element after it
    for i in range(n):
        for j in range(i + 1, n):

            if user_ids[i] == user_ids[j]:
                duplicates.add(user_ids[i])

    return duplicates
# Example usage
user_id_list = [101, 102, 103, 104, 102, 105, 101]
duplicates_found = find_duplicates_bruteforce(user_id_list)
print("Duplicate User IDs (Brute Force):", duplicates_found)
```

Output:



Optimized code:

```python
def find_duplicates_optimized(user_ids):    """

    seen = set()
    duplicates = set()

    for uid in user_ids:

        if uid in seen:
            duplicates.add(uid)    # Duplicate found
        else:
            seen.add(uid)          # First occurrence

    return duplicates
# Example usage
duplicates_optimized = find_duplicates_optimized(user_id_list)
print("Duplicate User IDs (Optimized):", duplicates_optimized)
```



Observation:

The brute-force approach detects duplicates by comparing every pair of elements, resulting in $O(n^2)$ time complexity, which becomes impractical for large datasets. The optimized solution uses a set to track previously seen IDs, allowing duplicates to be identified in constant time per element. This reduces the overall complexity to $O(n)$, making it suitable for large-scale data validation tasks.