AI-Assisted coding Assignment-12.4

Name: vaishnavi.N

Hall no: 2303A52405

Batch-44

Category	Algorithm	Time Complexity	Space Complexity
Simple Sorting	Bubble / Insertion	$O(n^2)$	O(1)
Efficient Sorting	Merge Sort	O(n log n)	O(n)
Fast Searching	Binary Search	O(log n)	O(1)
Optimization	Duplicate (Set)	O(n)	O(n)

Task 1: Bubble Sort for Ranking Exam Scores

- **AI Prompt Used**

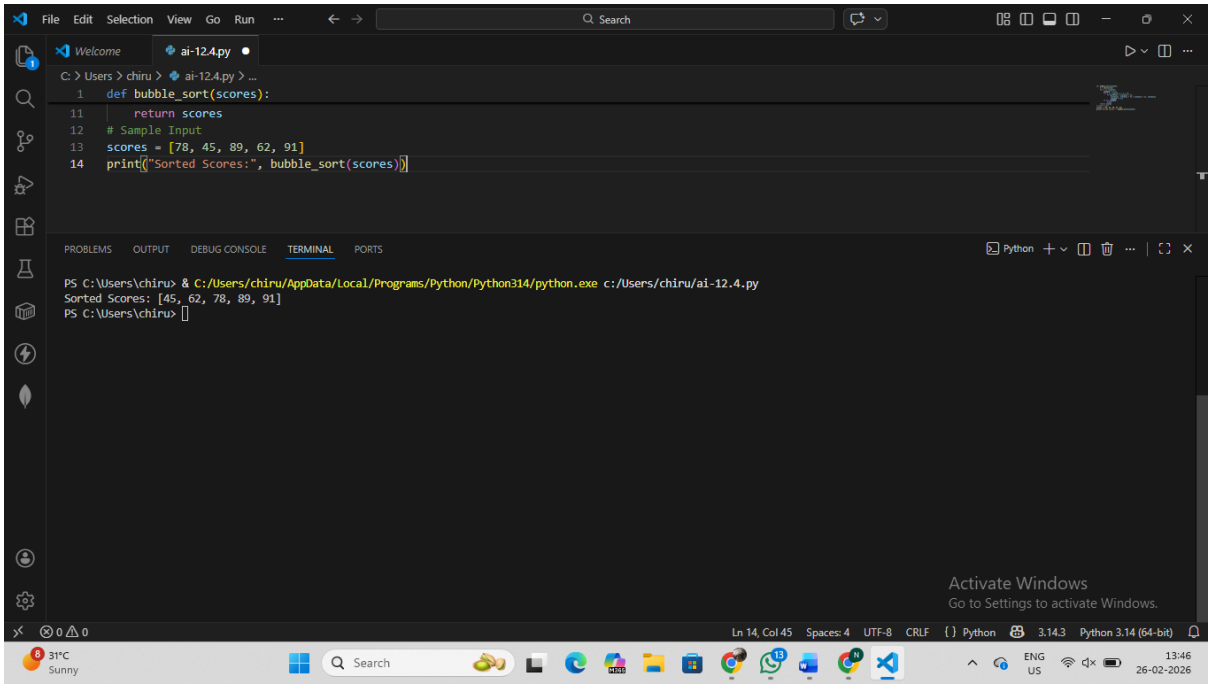
"Write a Bubble Sort program in Python to sort student scores. Add inline comments explaining comparisons, swaps, and passes. Include early termination optimization and time complexity analysis."

Input:

Code:

```python
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:
            break
    return scores
# Sample Input
scores = [78, 45, 89, 62, 91]
print("Sorted Scores:", bubble_sort(scores))
```

output:



Observations

1. **Quick Sort is generally faster in practice for random datasets.**
2. **Quick Sort performance depends heavily on pivot selection.**
3. **Merge Sort guarantees O(n log n) in all cases.**
4. **Merge Sort requires extra memory, Quick Sort is more memory-efficient.**
5. **For large stable sorting requirements, Merge Sort is preferred.**

6. **For in-place fast sorting, Quick Sort is preferred.**

Task 2: Improving Sorting for Nearly Sorted Attendance Records

- **AI Prompt Used**

"Given that attendance roll numbers are nearly sorted, suggest a better sorting algorithm than Bubble Sort and explain why. Provide Python code."

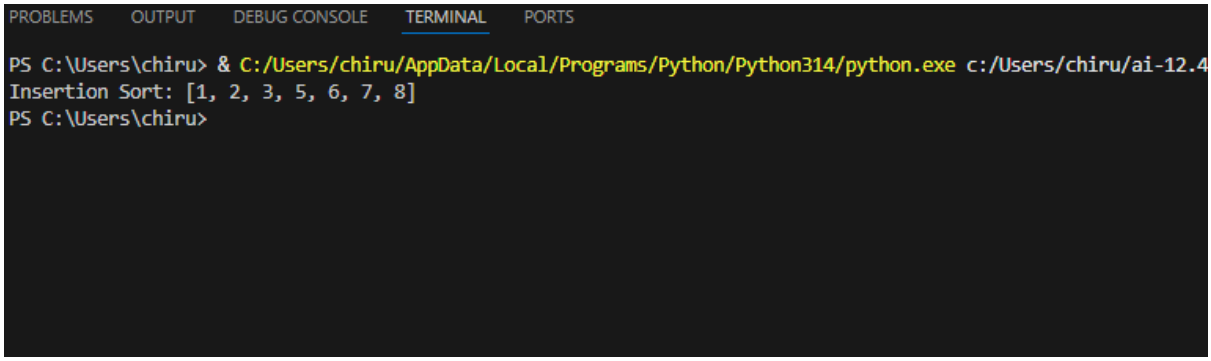
Bubble sort:

```python
#Bubble Sort (Basic Version)
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr
```

Insertion sort

```python
    return arr

#Insertion Sort Implementation
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]  # Element to be inserted
        j = i - 1

        # Move elements greater than key one position ahead
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1

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

    return arr


attendance = [1, 2, 3, 7, 5, 6, 8]
print("Insertion Sort:", insertion_sort(attendance))
```

Output:



Observations

Insertion Sort is more efficient for nearly sorted data.

1. **Bubble Sort performs unnecessary comparisons even if data is almost sorted.**
2. **Both algorithms are inefficient for large random datasets.**
3. **Insertion Sort adapts better to small changes.**
4. **For attendance records with minor updates, Insertion Sort is preferred.**
5. **Optimization depends on data characteristics.**

Task 3: Searching Student Records in a Database

AI-Generated Prompt Used

"Write Python implementations for Linear Search and Binary Search to search student roll numbers. Add proper docstrings explaining parameters and return values

Code:

Binary Search (For Sorted Data)

l

C: > Users > chiru > ai-12.4.py > ...

```
43
44
45
46  #task -3 Binary Search (For Sorted Data)
47  def binary_search(data, target):
48      left = 0
49      right = len(data) - 1
50
51      while left <= right:
52          mid = (left + right) // 2
53
54          if data[mid] == target:
55              return mid
56          elif data[mid] < target:
57              left = mid + 1
58          else:
59              right = mid - 1
60
61      return -1
62
63  # Example Usage
64  students_sorted = [101, 102, 105, 108, 110]
65  result = binary_search(students_sorted, 110)
66
67  if result != -1:
68      print("Roll number found at index:", result)
69  else:
70      print("Roll number not found")
```

Linear search: (For Unsorted Data)

Welcome ai-12.4.py

C: > Users > chiru > ai-12.4.py > linear_search

```python
#Task 3: Searching Student Records
# Linear Search (For Unsorted Data)
def linear_search(data, target):
    """
    Performs Linear Search on an unsorted list of student roll numbers.
    Parameters:
    data (list): List of student roll numbers (unsorted).
    target (int): Roll number to search.
    Returns:
    int: Index of the target roll number if found, otherwise -1.
    """
    for index in range(len(data)):
        if data[index] == target:
            return index
    return -1


# Example Usage
students_unsorted = [105, 102, 110, 101, 108]
result = linear_search(students_unsorted, 110)

if result != -1:
    print("Roll number found at index:", result)
else:
    print("Roll number not found")
```

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

Roll number found at index: 2
PS C:\Users\chiru>
              > & C:/Users/chiru/AppData/Local/Programs/Python/Python314/python.exe c:/Users/chiru/ai-12.4.py
Roll number found at index: 2
PS C:\Users\chiru>
```

Observations

Linear Search works on both sorted and unsorted data.

1. **Binary Search requires sorted data.**
2. **Binary Search is significantly faster for large datasets.**

3. **Linear Search is simple but inefficient for large systems.**
4. **Binary Search reduces search space exponentially.**

Task 4: Quick Sort vs Merge Sort

- **AI-Generated Prompt Used**

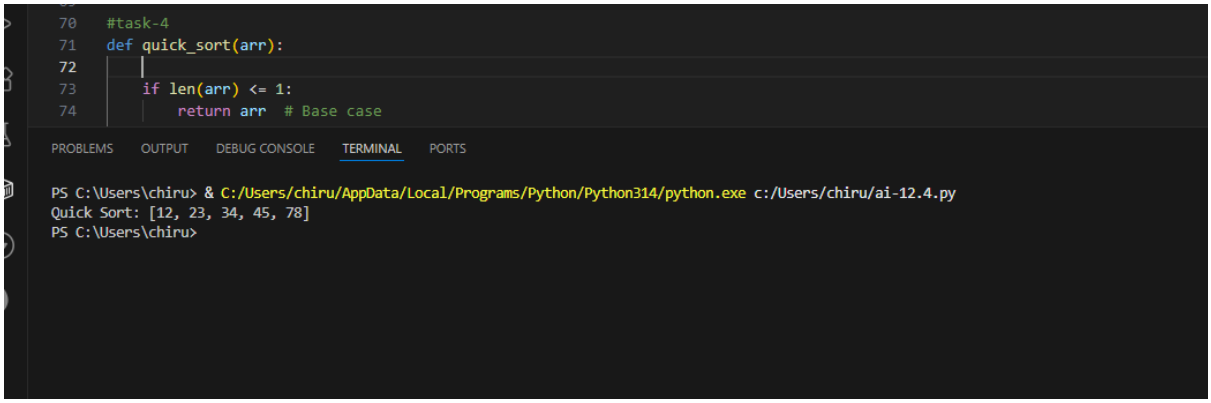
"Complete partially written recursive Quick Sort and Merge Sort functions in Python. Add meaningful docstrings explaining parameters and recursion. Provide time complexity analysis and compare performance for random, sorted, and reverse-sorted datasets."

Quick Sort Implementation:

Code:

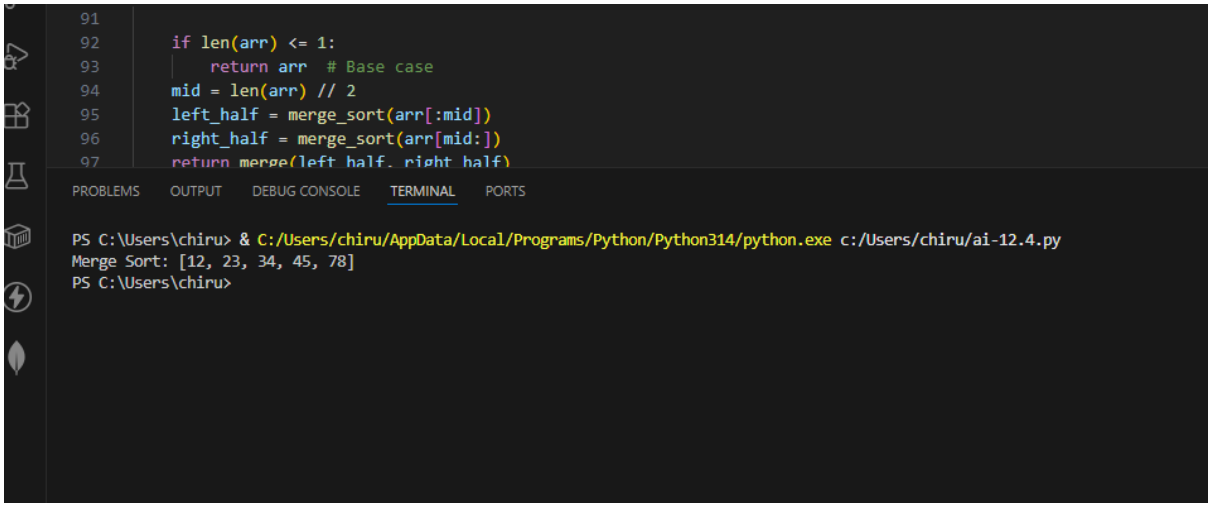
```
if result != -1:
    print("Roll number found at index:", result)
else:
    print("Roll number not found")"""

#task-4
def quick_sort(arr):
    
    if len(arr) <= 1:
        return arr  # Base case

    pivot = arr[len(arr) // 2]

    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]

    return quick_sort(left) + middle + quick_sort(right)


# Example
data = [45, 12, 78, 34, 23]
print("Quick Sort:", quick_sort(data))
```

Output:




```python
#Merge Sort Implementation
def merge_sort(arr):

    if len(arr) <= 1:
        return arr  # Base case
    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):
    merged = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            merged.append(left[i])
            i += 1
        else:
            merged.append(right[j])
            j += 1
    merged.extend(left[i:])
    merged.extend(right[j:])
    return merged
# Example
data = [45, 12, 78, 34, 23]
print("Merge Sort:", merge_sort(data))
```

Output:



Observations

1. **Quick Sort is generally faster in practice for random datasets.**
2. **Quick Sort performance depends heavily on pivot selection.**
3. **Merge Sort guarantees O(n log n) in all cases.**
4. **Merge Sort requires extra memory, Quick Sort is more memory-efficient.**
5. **For large stable sorting requirements, Merge Sort is preferred.**
6. **For in-place fast sorting, Quick Sort is preferred.**

Task 5: Optimizing Duplicate Detection Algorithm

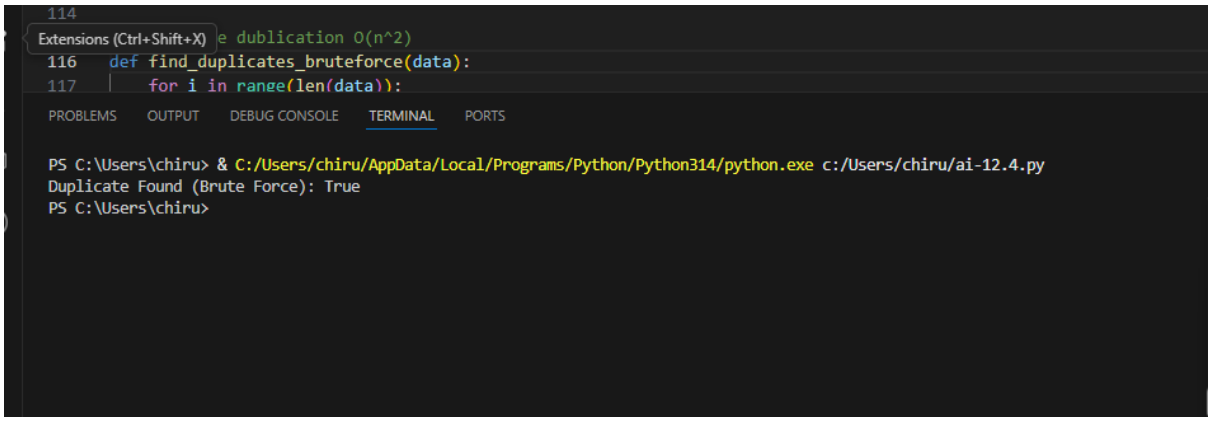
- **AI-Generated Prompt Used**

"Write a naive duplicate detection algorithm using nested loops in Python. Analyze its time complexity. Then optimize it using sets or dictionaries and explain the performance improvement."

- **Brute Force Duplicate Detection ($O(n^2)$)**

```python
data = [45, 12, 78, 34, 23]
print("Merge Sort:", merge_sort(data))"""

#Brute force dublication O(n^2)
def find_duplicates_bruteforce(data):
    for i in range(len(data)):
        for j in range(i + 1, len(data)):
            if data[i] == data[j]:
                return True
    return False


# Example
ids = [101, 203, 405, 203]
print("Duplicate Found (Brute Force):", find_duplicates_bruteforce(ids))
```

Output:



Optimized Duplicate Detection Using Set (O(n))

Code:

```python
def find_duplicates_bruteforce(data):
    for i in range(len(data)):
        for j in range(i + 1, len(data)):
            if data[i] == data[j]:
                return True
    return False


# Example
ids = [101, 203, 405, 203]
print("Duplicate Found (Brute Force):", find_duplicates_bruteforce(ids))"""
#Optimized Duplicate Detection Using Set (O(n))
def find_duplicates_optimized(data):
    seen = set()
    for item in data:
        if item in seen:
            return True
        seen.add(item)
    return False
# Example
ids = [101, 203, 405, 203]
print("Duplicate Found (Optimized):", find_duplicates_optimized(ids))
```

Output:

```
124   # Example

PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS

PS C:\Users\chiru> & C:/Users/chiru/AppData/Local/Programs/Python/Python314/python.exe c:/Users/chiru/ai-12.4.py
Duplicate Found (Optimized): True
PS C:\Users\chiru>
```

Observations

1. Brute-force approach is inefficient for large datasets.
2. Optimized approach reduces time complexity from $O(n^2)$ to $O(n)$.
3. Set-based method trades extra space for faster execution.
4. For very large datasets, optimized version is significantly better.
5. Brute-force is only acceptable for very small lists.
6. Performance improvement becomes dramatic as input size increases.