ASSIGNMENT-12.4

NAME:G.Harika

HT NO:2303A51612

BATCH:05

Task-01:

Prompt:

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

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

Bubble Sort implementation with:

AI-generated comments explaining the logic

Clear explanation of best, average, and worst-case complexity

Code:

```
# Sample input student_scores =
[85, 92, 78, 96, 88] # Bubble Sort
Implementation def
bubble_sort(scores):
   n = len(scores)
   # Traverse through all array elements
for i in range(n):
       swapped = False  # Early termination flag
# Last i elements are already in place         for
j in range(0, n - i - 1):
           # Traverse the array from 0 to n-i-1
           # Swap if the element found is greater than the next element
if scores[j] > scores[j + 1]:
              scores[j], scores[j + 1] = scores[j + 1], scores[j]
swapped = True
```

        # If no swapping occurred in this pass, list is sorted

if not swapped:

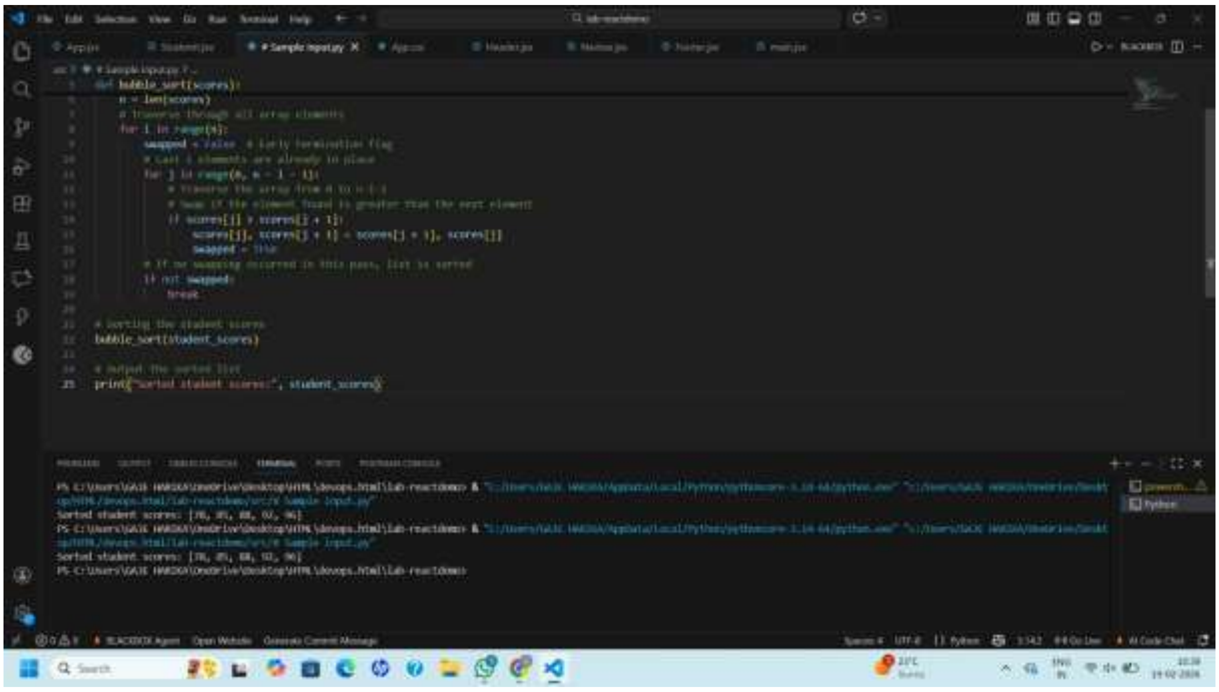
            break

Sorting the student scores

bubble_sort(student_scores) # Output the

sorted list print("Sorted student scores:",

student_scores)



```python
def bubble_sort(scores):
    n = len(scores)
    # Traverse through all array elements
    for i in range(n):
        swapped = False  # Early termination flag
        # Last i elements are already in place
        for j in range(0, n - i - 1):
            # Traverse the array from 0 to n-i-1
            # Swap if the element found is greater than the next element
            if scores[j] > scores[j + 1]:
                scores[j], scores[j + 1] = scores[j + 1], scores[j]
                swapped = True
        # If no swapping occurred in this pass, list is sorted
        if not swapped:
            break

# Sorting the student scores
bubble_sort(student_scores)

# Output the sorted list
print("Sorted student scores:", student_scores)
```

```
PS C:\Users\SAIK HARIKA\OneDrive\Desktop\HTML\devops.html\lab-reactdemo> & "C:/Users/SAIK HARIKA/AppData/Local/Python/pythoncore-3.14-64/python.exe" "c:/Users/SAIK HARIKA/OneDrive/Desktop/HTML/devops.html/lab-reactdemo/src/# Sample input.py"
Sorted student scores: [78, 85, 88, 92, 96]
PS C:\Users\SAIK HARIKA\OneDrive\Desktop\HTML\devops.html\lab-reactdemo> & "C:/Users/SAIK HARIKA/AppData/Local/Python/pythoncore-3.14-64/python.exe" "c:/Users/SAIK HARIKA/OneDrive/Desktop/HTML/devops.html/lab-reactdemo/src/# Sample input.py"
Sorted student scores: [78, 85, 88, 92, 96]
PS C:\Users\SAIK HARIKA\OneDrive\Desktop\HTML\devops.html\lab-reactdemo>
```

Explanation:

Time Complexity Analysis:

Best Case: O(n) - This occurs when the list is already sorted, and we only need one pass to confirm it.

Average Case: O(n^2) - This occurs when the list is in random order, requiring multiple passes to sort.

Worst Case: O(n^2) - This occurs when the list is sorted in reverse order, requiring the maximum number of comparisons and swaps.

Task-02:

Prompt:

You are maintaining an attendance system where student roll numbers are

already almost sorted, with only a few late updates.

Start with a Bubble Sort implementation.

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

Two sorting implementations,Bubble Sort, Insertion Sort

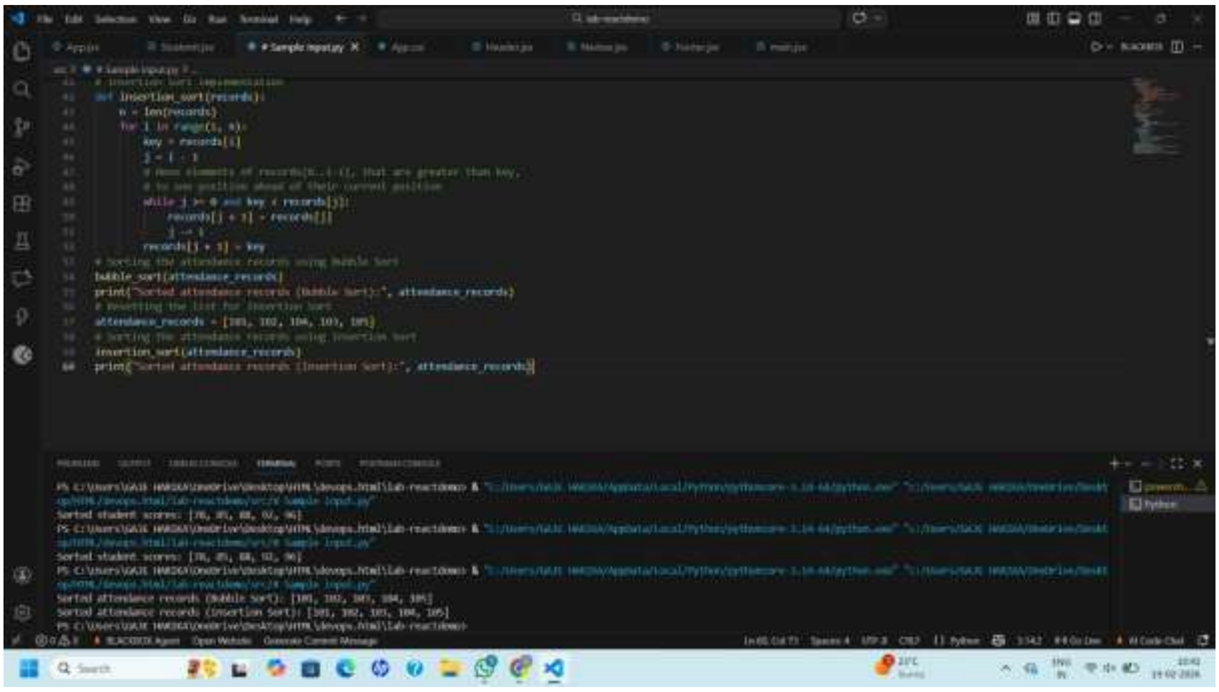
AI-assisted explanation highlighting efficiency differences for partially sorted datasets

<u>Code:</u>

```
# Sample input attendance_records = [101,
102, 104, 103, 105]
# Bubble Sort Implementation def
bubble_sort(records):
   n = len(records)     for i in
range(n):        swapped = False
for j in range(0, n - i - 1):            if
records[j] > records[j + 1]:
            records[j], records[j + 1] = records[j + 1], records[j]
swapped = True        if not swapped:
         break
# Insertion Sort Implementation def
insertion_sort(records):
   n = len(records)
for i in range(1, n):
key = records[i]
j = i - 1
      # Move elements of records[0..i-1], that are greater than key,
      # to one position ahead of their current position
while j >= 0 and key < records[j]:
         records[j + 1] = records[j]
j -= 1
      records[j + 1] = key
# Sorting the attendance records using Bubble Sort
bubble_sort(attendance_records) print("Sorted attendance records
(Bubble Sort):", attendance_records) # Resetting the list for Insertion
Sort attendance_records = [101, 102, 104, 103, 105] # Sorting the
attendance records using Insertion Sort
```

insertion_sort(attendance_records) print("Sorted attendance records

(Insertion Sort):", attendance_records)



```
# Insertion Sort implementation
def insertion_sort(records):
    n = len(records)
    for i in range(1, n):
        key = records[i]
        j = i - 1
        # Move elements of records[0..i-1], that are greater than key,
        # to one position ahead of their current position
        while j >= 0 and key < records[j]:
            records[j + 1] = records[j]
            j -= 1
        records[j + 1] = key
# Sorting the attendance records using Bubble Sort
bubble_sort(attendance_records)
print("Sorted attendance records (Bubble Sort):", attendance_records)
# Resetting the list for Insertion Sort
attendance_records = [105, 102, 104, 103, 101]
# Sorting the attendance records using Insertion Sort
insertion_sort(attendance_records)
print("Sorted attendance records (Insertion Sort):", attendance_records)
```

```
PS C:\Users\[illegible]\OneDrive\Desktop\HTML\devops.html\lab-reactdemo> & "[illegible]/python.exe" "[illegible]/src/# Sample input.py"
Sorted student scores: [78, 85, 88, 92, 96]
PS C:\Users\[illegible]\OneDrive\Desktop\HTML\devops.html\lab-reactdemo> & "[illegible]/python.exe" "[illegible]/src/# Sample input.py"
Sorted student scores: [78, 85, 88, 92, 96]
PS C:\Users\[illegible]\OneDrive\Desktop\HTML\devops.html\lab-reactdemo> & "[illegible]/python.exe" "[illegible]/src/# Sample input.py"
Sorted attendance records (Bubble Sort): [101, 102, 103, 104, 105]
Sorted attendance records (Insertion Sort): [101, 102, 103, 104, 105]
PS C:\Users\[illegible]\OneDrive\Desktop\HTML\devops.html\lab-reactdemo>
```

Explanation:

Explanation of Efficiency Differences:

Insertion Sort is more efficient than Bubble Sort for nearly sorted data because it minimizes the number of comparisons and movements. Insertion Sort only moves elements that are out of order, while Bubble Sort continues to compare and swap adjacent elements regardless of their order, leading to unnecessary operations in nearly sorted lists.

Task-03:

Prompt:

You are developing a student information portal where users search for student

records by roll number.

Implement:

Linear Search for unsorted student data

Binary Search for sorted student data

Add docstrings explaining parameters and return values

Explain when Binary Search is applicable

Highlight performance differences between the two searches

Two working search implementations with docstrings o

Time complexity o Use cases for Linear vs Binary Search

• A short student observation comparing results on sorted vs unsorted lists **Code:**

```
# Sample input student_records = [101,
102, 104, 103, 105] # Linear Search
Implementation def linear_search(records,
target):
    """
    Perform a linear search for the target roll number in the records list.
    Parameters:
    records (list): A list of student roll numbers.
target (int): The roll number to search for.
    Returns:
    int: The index of the target if found, otherwise -1.
    """     for index in
range(len(records)):         if
records[index] == target:
            return index
return -1
# Binary Search Implementation def
binary_search(records, target):
    """
    Perform a binary search for the target roll number in the sorted records list.
    Parameters:
    records (list): A sorted list of student roll numbers.
target (int): The roll number to search for.

    Returns:
    int: The index of the target if found, otherwise -1.
    """     left, right = 0,
len(records) - 1     while left <=
right:
        mid = left + (right - left) // 2
if records[mid] == target:
```

return mid elif

records[mid] < target:

left = mid + 1

else: right =

mid - 1 return -1

Sorting the student records for Binary Search student_records.sort()

Searching for a roll number using Linear Search

target_roll_number = 103 linear_result =

linear_search(student_records, target_roll_number)

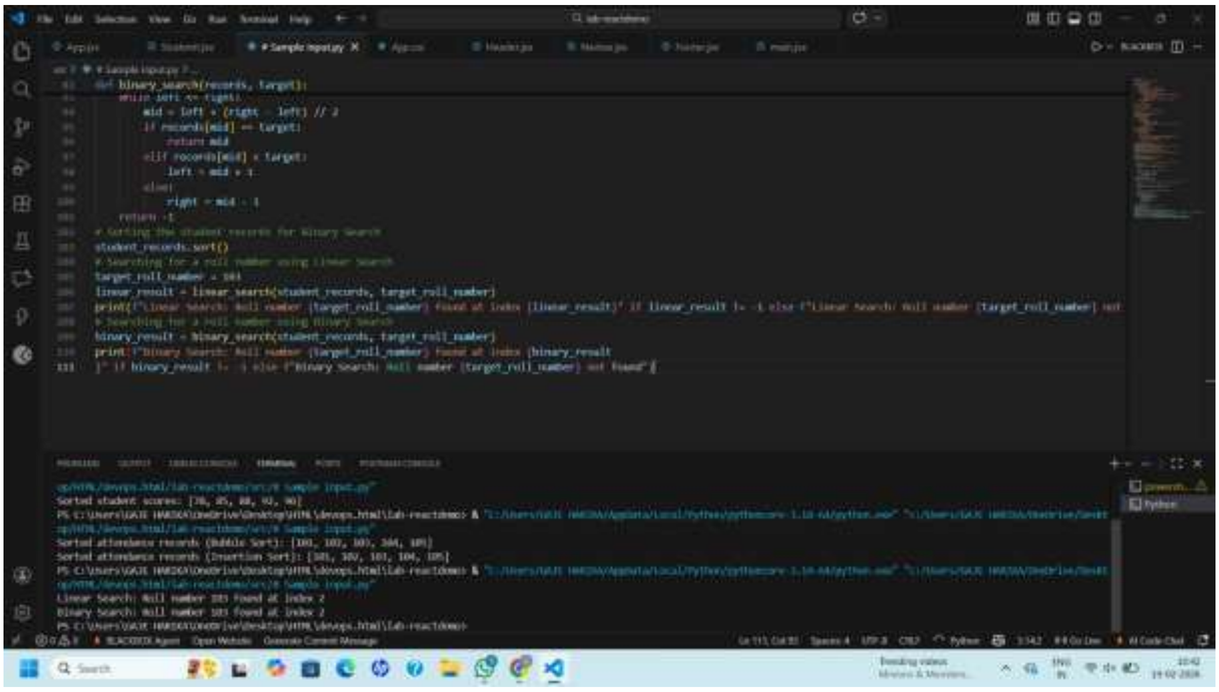
print(f"Linear Search: Roll number {target_roll_number} found at index {linear_result}" if linear_result != -1 else f"Linear Search: Roll number {target_roll_number} not found")

Searching for a roll number using Binary Search binary_result =

binary_search(student_records, target_roll_number) print(f"Binary Search: Roll

number {target_roll_number} found at index {binary_result

}" if binary_result != -1 else f"Binary Search: Roll number {target_roll_number} not found")



```
def binary_search(records, target):
    while left <= right:
        mid = left + (right - left) // 2
        if records[mid] == target:
            return mid
        elif records[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1
# Sorting the student records for Binary Search
student_records.sort()
# Searching for a roll number using Linear Search
target_roll_number = 103
linear_result = linear_search(student_records, target_roll_number)
print(f"Linear Search: Roll number {target_roll_number} found at index {linear_result}" if linear_result != -1 else f"Linear Search: Roll number {target_roll_number} not
# Searching for a roll number using Binary Search
binary_result = binary_search(student_records, target_roll_number)
print(f"Binary Search: Roll number {target_roll_number} found at index {binary_result
}" if binary_result != -1 else f"Binary Search: Roll number {target_roll_number} not found"
```

```
Sorted student scores: [78, 85, 88, 92, 90]
PS C:\Users\[illegible]\OneDrive\Desktop\HTML\devops.html\lab-reactdemo> & [illegible]
Sorted attendance records (Bubble Sort): [101, 102, 103, 104, 105]
Sorted attendance records (Insertion Sort): [101, 102, 103, 104, 105]
PS C:\Users\[illegible]\OneDrive\Desktop\HTML\devops.html\lab-reactdemo> & [illegible]
Linear Search: Roll number 103 found at index 2
Binary Search: Roll number 103 found at index 2
PS C:\Users\[illegible]\OneDrive\Desktop\HTML\devops.html\lab-reactdemo>
```

Explanation:

Explanation of Performance Differences:

Linear Search has a time complexity of O(n) because it may need to check each element in the worst case. Binary Search, on the other hand, has a time complexity of O(log n) because it halves the search space with each iteration. Binary Search is only applicable to sorted lists, while Linear Search can be used on unsorted data. In practice, Binary Search is much faster for large datasets, while Linear Search may be more suitable for small or unsorted lists.

Task-04:

Prompt:

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).

Provide AI with partially written recursive functions for:

Quick Sort,Merge Sort

Complete the recursive logic

Add meaningful docstrings

Explain how recursion works in each algorithm

• Test both algorithms on:

o Random data, Sorted data,Reverse-sorted data

Fully functional Quick Sort and Merge Sort implementations

• AI-generated comparison covering:

o Best, average, and worst-case complexities

Practical scenarios where one algorithm is preferred over the

other

Code:

```
# Sample input data_random = [38, 27, 43, 3,
9, 82, 10] data_sorted = [3, 9, 10, 27, 38, 43,
82] data_reverse_sorted = [82, 43, 38, 27, 10,
9, 3]
# Quick Sort Implementation def
quick_sort(arr):
    """
    Sorts an array using the Quick Sort algorithm.

    Parameters:
    arr (list): The list of elements to be sorted.

    Returns:
```

```
    list: A new sorted list.
    """     if len(arr)
<= 1:
        return arr
else:
        pivot = arr[len(arr) // 2]  # Choosing the middle element as pivot         left = [x for x in arr if x
< pivot]  # Elements less than pivot         middle = [x for x in arr if x == pivot]  # Elements equal to
pivot         right = [x for x in arr if x > pivot]  # Elements greater than pivot         return
quick_sort(left) + middle + quick_sort(right)  # Recursively sort left and right partitions
# Merge Sort Implementation def
merge_sort(arr):
    """
    Sorts an array using the Merge Sort algorithm.
    Parameters:
    arr (list): The list of elements to be sorted.
    Returns:
    list: A new sorted list.
    """     if len(arr) <= 1:         return arr     mid = len(arr) // 2  #
Finding the mid of the array     left_half = merge_sort(arr[:mid])  #
Recursively sort the left half     right_half = merge_sort(arr[mid:])  #
Recursively sort the right half     return merge(left_half, right_half)  #
Merge the sorted halves def merge(left, right):
    """Merges two sorted lists into a single 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:
            result.append(right[j])
j += 1
```

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

result.extend(right[j:]) # Append remaining elements of right   return

result

Testing Quick Sort and Merge Sort on different datasets print("Quick

Sort on Random Data:", quick_sort(data_random)) print("Merge Sort

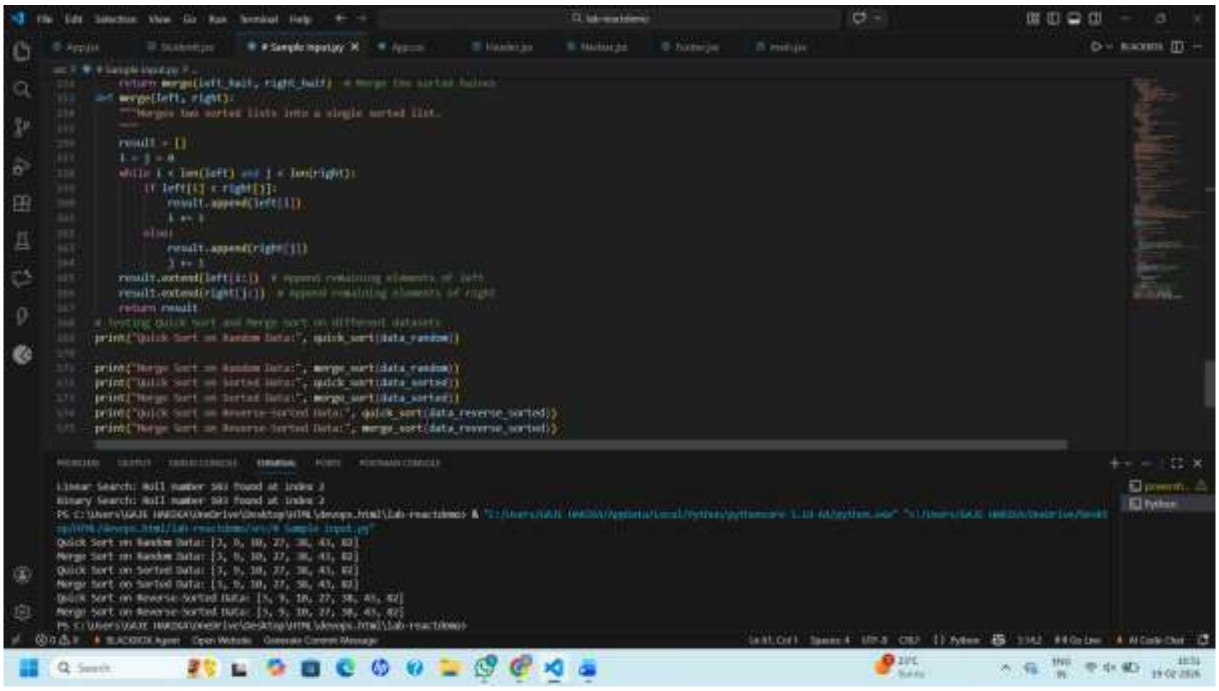
on Random Data:", merge_sort(data_random)) print("Quick Sort on

Sorted Data:", quick_sort(data_sorted)) print("Merge Sort on Sorted

Data:", merge_sort(data_sorted)) print("Quick Sort on Reverse-Sorted

Data:", quick_sort(data_reverse_sorted)) print("Merge Sort on

Reverse-Sorted Data:", merge_sort(data_reverse_sorted))



```
    return merge(left_half, right_half)  # Merge the sorted halves
def merge(left, right):
    """Merges two sorted lists into a single 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:
            result.append(right[j])
            j += 1
    result.extend(left[i:])  # Append remaining elements of left
    result.extend(right[j:])  # Append remaining elements of right
    return result
# Testing Quick Sort and Merge Sort on different datasets
print("Quick Sort on Random Data:", quick_sort(data_random))

print("Merge Sort on Random Data:", merge_sort(data_random))
print("Quick Sort on Sorted Data:", quick_sort(data_sorted))
print("Merge Sort on Sorted Data:", merge_sort(data_sorted))
print("Quick Sort on Reverse-Sorted Data:", quick_sort(data_reverse_sorted))
print("Merge Sort on Reverse-Sorted Data:", merge_sort(data_reverse_sorted))
```

```
Linear Search: Roll number 103 found at index 2
Binary Search: Roll number 103 found at index 2
PS C:\Users\GAJE HARIKA\OneDrive\Desktop\HTML\devops.html\lab-reactdemo> & [illegible]
[illegible]/HTML/devops.html/lab-reactdemo/src/# Sample input.py"
Quick Sort on Random Data: [3, 9, 10, 27, 38, 43, 82]
Merge Sort on Random Data: [3, 9, 10, 27, 38, 43, 82]
Quick Sort on Sorted Data: [3, 9, 10, 27, 38, 43, 82]
Merge Sort on Sorted Data: [3, 9, 10, 27, 38, 43, 82]
Quick Sort on Reverse-Sorted Data: [3, 9, 10, 27, 38, 43, 82]
Merge Sort on Reverse-Sorted Data: [3, 9, 10, 27, 38, 43, 82]
PS C:\Users\GAJE HARIKA\OneDrive\Desktop\HTML\devops.html\lab-reactdemo>
```

Explanation:

Explanation of Performance Differences:

Quick Sort has an average and best-case time complexity of O(n log n) when the pivot divides the array into two equal halves. However, its worst-case time complexity is O(n^2) when the smallest or largest element is always chosen as the pivot (e.g., when the array is already sorted or reverse sorted). Merge Sort, on the other hand, has a consistent time complexity of O(n log n) in all cases because it always divides the array into two halves and processes them independently. Quick Sort is generally faster for smaller datasets and is an in-place sorting algorithm, while Merge Sort is more efficient for larger datasets and is a stable sorting algorithm that requires additional space for merging.

Task-05:

Prompt:

You are building a data validation module that must detect duplicate user

IDs in a large dataset before importing it into a system.

Write a naive duplicate detection algorithm using nested loops.

Suggest an optimized approach using sets or dictionaries

Rewrite the algorithm with improved efficiency

Compare execution behavior conceptually for large input sizes

Two versions of the algorithm:Brute-force (O(n²)), Optimized (O(n)) AI-assisted

explanation showing how and why performance improved

<u>Code:</u>

```
# Sample input user_ids = [101, 102, 103, 104, 105,
101, 106, 107, 102]
# Naive Duplicate Detection Algorithm    def
naive_duplicate_detection(ids):
   """
   Detects duplicates in a list of user IDs using a brute-force approach.
   Parameters:
   ids (list): A list of user IDs.
   Returns:
   set: A set of duplicate user IDs.
   """     duplicates =
set()     for i in
range(len(ids)):
      for j in range(i + 1, len(ids)):
if ids[i] == ids[j]:
            duplicates.add(ids[i])
return duplicates
# Optimized Duplicate Detection Algorithm def
optimized_duplicate_detection(ids):
   """
   Detects duplicates in a list of user IDs using an optimized approach with a set.
   Parameters:
   ids (list): A list of user IDs.
```

Returns:

set: A set of duplicate user IDs.

""" seen =

set() duplicates =

set() for id in

ids: if id in

seen:

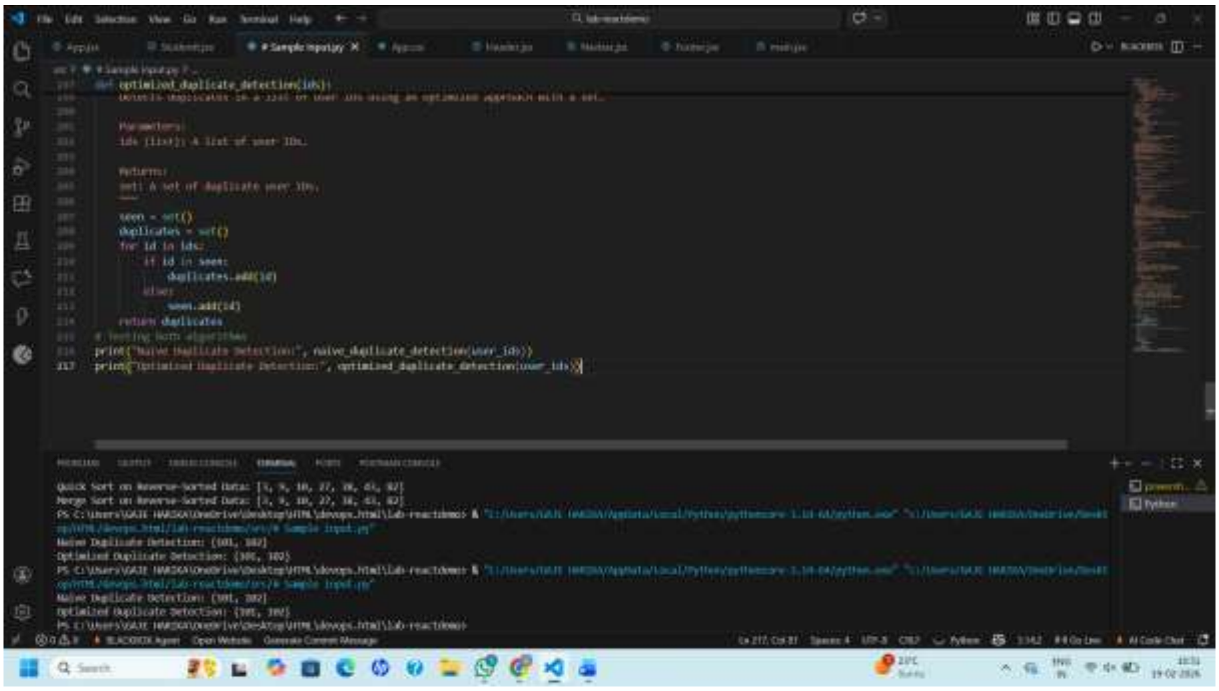
duplicates.add(id)

else:

seen.add(id) return duplicates # Testing both algorithms print("Naive

Duplicate Detection:", naive_duplicate_detection(user_ids)) print("Optimized

Duplicate Detection:", optimized_duplicate_detection(user_ids))



```
def optimized_duplicate_detection(ids):
    Detects duplicates in a list of user IDs using an optimized approach with a set.

    Parameters:
    ids (list): A list of user IDs.

    Returns:
    set: A set of duplicate user IDs.
    """
    seen = set()
    duplicates = set()
    for id in ids:
        if id in seen:
            duplicates.add(id)
        else:
            seen.add(id)
    return duplicates
# Testing both algorithms
print("Naive Duplicate Detection:", naive_duplicate_detection(user_ids))
print("Optimized Duplicate Detection:", optimized_duplicate_detection(user_ids))
```

```
Quick Sort on Reverse-Sorted Data: [3, 9, 10, 27, 38, 43, 82]
Merge Sort on Reverse-Sorted Data: [3, 9, 10, 27, 38, 43, 82]
PS C:\Users\GAJE HARIKA\OneDrive\Desktop\HTML\devops.html\lab-reactdemo> & [illegible]
Naive Duplicate Detection: {101, 102}
Optimized Duplicate Detection: {101, 102}
PS C:\Users\GAJE HARIKA\OneDrive\Desktop\HTML\devops.html\lab-reactdemo> & [illegible]
Naive Duplicate Detection: {101, 102}
Optimized Duplicate Detection: {101, 102}
PS C:\Users\GAJE HARIKA\OneDrive\Desktop\HTML\devops.html\lab-reactdemo>
```

Explanation:

Explanation of Performance Improvements:

The naive duplicate detection algorithm has a time complexity of $O(n^2)$ because it uses nested loops to compare each element with every other element in the list. This becomes inefficient as the size of the input grows. The optimized algorithm, on the other hand, has a time complexity of O(n) because it uses a set to track seen user IDs. Each lookup and insertion operation in a set is on average O(1), allowing us to detect duplicates in a single pass through the list. This results in significantly improved performance, especially for large datasets, as it avoids redundant comparisons and reduces the number of operations needed to identify duplicates.