2303a510f2 – Anitha

Batch – 14

Assignment – 12.4

Task 1: Bubble Sort for Ranking Exam Scores

PROMPT:

Write a Python program to implement Bubble Sort for sorting student exam scores.Add inline comments explaining comparisons, swaps, and iteration passes.Include an early-termination optimization if the list becomes sorted.Provide time complexity analysis (best, average, worst case).Show sample input and output.

CODE:

```
def bubble_sort(scores):
    n = len(scores)
    # Flag to check if any swapping occurs
    swapped = True

    # Outer loop to traverse through all elements
    for i in range(n):
        # If no swapping occurs, the list is already sorted
        if not swapped:
            break

        # Reset the flag for this iteration
        swapped = False

        # Inner loop to compare adjacent elements
        for j in range(0, n-i-1):
            # Compare the adjacent elements
            if scores[j] > scores[j+1]:
                # Swap if the element is greater than the next element
                scores[j], scores[j+1] = scores[j+1], scores[j]
                swapped = True  # Set the flag to True if a swap occurred
    return scores
# Sample input: List of student exam scores
scores = list(map(int, input("Enter student exam scores separated by spaces:
").split()))
```

```
# Sorting the scores using bubble sort

sorted_scores = bubble_sort(scores)
# Output the sorted scores
print("Sorted Scores:", sorted_scores)
```

OUTPUT:

```
PS C:\Users\akshi\OneDrive\Desktop\full stack> & C:/Users/akshi
3.13.exe c:/Users/akshi/OneDrive/Desktop/12.4.py
Enter student exam scores separated by spaces: 88 79 92 85 75
Sorted Scores: [75, 79, 85, 88, 92]
PS C:\Users\akshi\OneDrive\Desktop\full stack>
```

TIME COMPLEXITY:

Case	Complexity	Explanation
Best Case	O(n)	Already sorted (early termination)
Average Case	$O(n^2)$	Random order
Worst Case	$O(n^2)$	Reverse sorted

Task 2: Improving Sorting for Nearly Sorted Attendance Records

PROMPT:

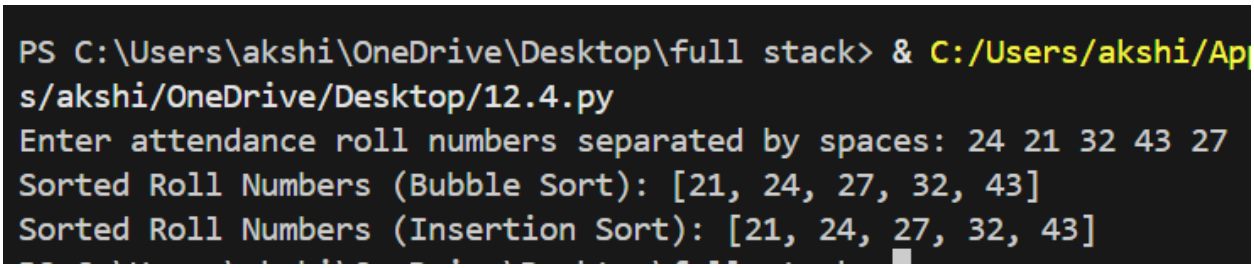
Given that attendance roll numbers are nearly sorted,suggest a better sorting algorithm than Bubble Sort.Implement Insertion Sort in Python.Explain why it performs better for nearly sorted data.Compare behavior with Bubble Sort.

CODE:

```
def bubble_sort(arr):
    n = len(arr)
    # Flag to check if any swapping occurs
    swapped = True

    # Outer loop to traverse through all elements
    for i in range(n):
```

```
        # If no swapping occurs, the list is already sorted
        if not swapped:
            break

        # Reset the flag for this iteration
        swapped = False

        # Inner loop to compare adjacent elements
        for j in range(0, n-i-1):
            # Compare the adjacent elements
            if arr[j] > arr[j+1]:
                # Swap if the element is greater than the next element
                arr[j], arr[j+1] = arr[j+1], arr[j]
                swapped = True  # Set the flag to True if a swap occurred
    return arr
def insertion_sort(arr):
    # Traverse through 1 to len(arr)
    for i in range(1, len(arr)):
        key = arr[i]  # The current element to be compared
        j = i - 1    # The index of the last sorted element

        # Move elements of arr[0..i-1], that are greater than key,
        # to one position ahead of their current position
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]  # Shift the element one position to the right
            j -= 1

        arr[j + 1] = key  # Place the key in its correct position
    return arr
# Sample input: List of attendance roll numbers (nearly sorted)
roll_numbers = list(map(int, input("Enter attendance roll numbers separated by
spaces: ").split()))
# Sorting the roll numbers using bubble sort
sorted_roll_numbers_bubble = bubble_sort(roll_numbers.copy())
# Sorting the roll numbers using insertion sort
sorted_roll_numbers_insertion = insertion_sort(roll_numbers.copy())
# Output the sorted roll numbers
print("Sorted Roll Numbers (Bubble Sort):", sorted_roll_numbers_bubble)
print("Sorted Roll Numbers (Insertion Sort):", sorted_roll_numbers_insertion)
```

OUTPUT:



EXPLANATION:

- Insertion Sort shifts elements only when necessary.
- If list is almost sorted → very few shifts.
- Best Case Complexity: **O(n)**
- Bubble Sort still makes unnecessary comparisons.

COMPARISON:

Algorithm	Nearly Sorted Performance
Bubble Sort	Still $O(n^2)$ comparisons
Insertion Sort	Close to O(n)

TASK 3: Searching Student Records

PROMPT:

Implement Linear Search and Binary Search in Python.Add proper docstrings explaining parameters and return values.Explain when Binary Search can be used.Compare time complexities and use cases.

CODE:

```
def linear_search(arr, target):
    """
    Perform a linear search for the target element in the given array.

    Parameters:
    arr (list): The list of elements to search through.
    target: The element to search for.

    Returns:
    int: The index of the target element if found, otherwise -1.
    """
    for index in range(len(arr)):
        if arr[index] == target:
            return index  # Target found, return its index
    return -1  # Target not found, return -1
def binary_search(arr, target):
    """Perform a binary search for the target element in the given sorted array.
    Parameters:
    arr (list): The sorted list of elements to search through.
    target: The element to search for.
    Returns:
    int: The index of the target element if found, otherwise -1.
    """
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = left + (right - left) // 2  # Calculate the middle index
        if arr[mid] == target:
            return mid  # Target found, return its index
        elif arr[mid] < target:
            left = mid + 1  # Search in the right half
        else:
            right = mid - 1  # Search in the left half
    return -1  # Target not found, return -1
arr=list(map(int, input("Enter a sorted list of elements separated by spaces:
").split()))
target=int(input("Enter the target element to search for: "))
# Using Linear Search
linear_result = linear_search(arr, target)
# Using Binary Search
binary_result = binary_search(arr, target)
print(f"Linear Search: Target found at index {linear_result}" if linear_result !=
-1 else "Linear Search: Target not found")
```

```
print(f"Binary Search: Target found at index {binary_result}" if binary_result !=
-1 else "Binary Search: Target not found")
```

OUTPUT:

```
Enter a sorted list of elements separated by spaces: 24 32 45 63 57 26
Enter the target element to search for: 32
Linear Search: Target found at index 1
Binary Search: Target found at index 1
PS C:\Users\akshi\OneDrive\Desktop\full stack>
```

COMPARISON:

Algorithm	Time Complexity	Use Case
Linear Search	O(n)	Unsorted data
Binary Search	O(log n)	Sorted data only

OBSERVATION:

Binary Search is much faster for large sorted datasets.

TASK 4: Quick Sort vs Merge Sort

PROMPT:

Complete recursive implementations of Quick Sort and Merge Sort.Add meaningful docstrings.Explain how recursion works in each.Compare performance for random, sorted, and reverse sorted data.

CODE:

```
def quick_sort(arr):
    """
    Perform Quick Sort on the given array.

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

```
    Returns:
    list: A new sorted list containing the elements of the original array.
    """
    if len(arr) <= 1:
        return arr  # Base case: a list of zero or one element is already sorted
    pivot = arr[len(arr) // 2]  # Choose the middle element as the pivot
    left = [x for x in arr if x < pivot]  # Elements less than the pivot
    middle = [x for x in arr if x == pivot]  # Elements equal to the pivot
    right = [x for x in arr if x > pivot]  # Elements greater than the pivot
    return quick_sort(left) + middle + quick_sort(right)  # Recursively sort and
combine
def merge_sort(arr):
    """Perform Merge Sort on the given array.
    Parameters:
    arr (list): The list of elements to be sorted.
    Returns:
    list: A new sorted list containing the elements of the original array.
    """
    if len(arr) <= 1:
        return arr  # Base case: a list of zero or one element is already sorted
    mid = len(arr) // 2  # Find the middle index
    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):
    """Merge two sorted lists into a single sorted list.

    Parameters:
    left (list): The first sorted list.
    right (list): The second sorted list.

    Returns:
    list: A merged and sorted list containing all elements from both input lists.
    """
    merged = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            merged.append(left[i])  # Add the smaller element from left
            i += 1
        else:
            merged.append(right[j])  # Add the smaller element from right
            j += 1
    merged.extend(left[i:])  # Add any remaining elements from left
```

```
    merged.extend(right[j:])  # Add any remaining elements from right
    return merged
arr=list(map(int, input("Enter a list of elements separated by spaces:
").split()))
# Sorting using Quick Sort
sorted_quick = quick_sort(arr)
# Sorting using Merge Sort
sorted_merge = merge_sort(arr)
print("Sorted using Quick Sort:", sorted_quick)
print("Sorted using Merge Sort:", sorted_merge)
```

OUTPUT:

```
PS C:\Users\akshi\OneDrive\Desktop\full stack> & C:/Users/aks
s/akshi/OneDrive/Desktop/12.4.py
Enter a list of elements separated by spaces: 20 30 40 60 50
Sorted using Quick Sort: [20, 30, 40, 50, 60]
Sorted using Merge Sort: [20, 30, 40, 50, 60]
```

COMPARISON:

Algorithm	Best	Average	Worst
Quick Sort	O(n log n)	O(n log n)	$O(n^2)$
Merge Sort	O(n log n)	O(n log n)	O(n log n)

Task 5: Optimizing a Duplicate Detection Algorithm

CODE :

```
def naive_duplicate_detection(arr):
    """
    Detect duplicates in the given array using a naive approach with nested
loops.

    Parameters:
    arr (list): The list of elements to check for duplicates.
```

```
    Returns:
    bool: True if duplicates are found, False otherwise.
    """
    for i in range(len(arr)):
        for j in range(i + 1, len(arr)):
            if arr[i] == arr[j]:  # Compare each element with every other element
                return True  # Duplicate found
    return False  # No duplicates found
def optimized_duplicate_detection(arr):
    """Detect duplicates in the given array using an optimized approach with a
set.
    Parameters:
    arr (list): The list of elements to check for duplicates.
    Returns:
    bool: True if duplicates are found, False otherwise.
    """
    seen = set()  # Create an empty set to store seen elements
    for element in arr:
        if element in seen:  # Check if the element is already in the set
            return True  # Duplicate found
        seen.add(element)  # Add the element to the set
    return False  # No duplicates found
arr=list(map(int, input("Enter a list of elements separated by spaces:
").split()))
# Using Naive Duplicate Detection
naive_result = naive_duplicate_detection(arr)
# Using Optimized Duplicate Detection
optimized_result = optimized_duplicate_detection(arr)
print("Naive Duplicate Detection:", "Duplicates found" if naive_result else "No
duplicates found")
print("Optimized Duplicate Detection:", "Duplicates found" if optimized_result
else "No duplicates found")
```

OUTPUT:

```
PS C:\Users\akshi\OneDrive\Desktop\full stack> & C:/Users/aks
s/akshi/OneDrive/Desktop/12.4.py
Enter a list of elements separated by spaces: 25 34 26 25 32
Naive Duplicate Detection: Duplicates found
Optimized Duplicate Detection: Duplicates found
```

EXPLANATION:

- The brute-force duplicate detection algorithm uses nested loops to compare every user ID with every other ID, resulting in **$O(n^2)$** time complexity.
- The optimized algorithm uses a **set (hashing)** to track seen IDs, allowing duplicate checks in **O(1)** time per element.
- This reduces the overall time complexity from **$O(n^2)$** to **O(n)**, making the algorithm significantly faster.
- For large datasets, the optimized version scales efficiently, while the brute-force approach becomes impractically slow.