Lab 12 – Algorithms with AI Assistance

AI Assisted Coding
Assignment Number: 12.4

A.Vasantha Shoba Rani

2303A51395

BT.NO: 06

Task 1: Bubble Sort for Ranking Exam Scores

Question

You are working on a college result processing system where a small list of student exam scores must be sorted after each internal assessment.

Implement Bubble Sort in Python to sort a list of student scores.
Explain comparisons, swaps, early termination, and time complexity.

Python Code

```
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



# Testing

marks = [45, 78, 62, 90, 55]

print(bubble_sort(marks))
```

Output

[45, 55, 62, 78, 90]

Code Explanation

- Bubble Sort repeatedly compares adjacent elements and swaps them if they are in the wrong order.
- The swapped flag allows early termination when the list becomes sorted.
- Best Case: $O(n)$ when the list is already sorted.
- Average Case: $O(n^2)$.
- Worst Case: $O(n^2)$ when the list is reverse sorted.
- Bubble Sort is suitable only for small datasets.

Task 2: Improving Sorting for Nearly Sorted Attendance Records

Question

Student roll numbers are almost sorted with a few late updates.
Implement Bubble Sort and Insertion Sort, then explain why Insertion Sort is better for nearly sorted data.

Python Code (Bubble Sort)

```
def bubble_sort(data):
    for i in range(len(data)):
        for j in range(0, len(data) - i - 1):
            if data[j] > data[j + 1]:
                data[j], data[j + 1] = data[j + 1], data[j]
    return data
```

Output

Bubble Sort executed

Code Explanation

- Bubble Sort always performs comparisons even if the list is nearly sorted.

- This leads to unnecessary iterations.

Python Code (Insertion Sort)

```
def insertion_sort(data):
    for i in range(1, len(data)):
        key = data[i]
        j = i - 1
        while j >= 0 and data[j] > key:
            data[j + 1] = data[j]
            j -= 1
        data[j + 1] = key
    return data



# Testing
roll_numbers = [1, 2, 3, 5, 4, 6]
print(insertion_sort(roll_numbers))
```

Output

```
[1, 2, 3, 4, 5, 6]
```

Code Explanation

- Insertion Sort shifts only misplaced elements.
- It performs efficiently on nearly sorted data.
- Time Complexity:
    - Best Case: O(n)
    - Worst Case: O(n^2)
- Insertion Sort is preferred for small or partially sorted datasets.

Task 3: Searching Student Records in a Database

Question

Implement Linear Search for unsorted data and Binary Search for sorted data.
Explain use cases and performance differences.

Python Code (Linear Search)

```
def linear_search(data, target):
    """
    Searches for a target value in an unsorted list.

    Parameters:
    data (list): List of student roll numbers
    target (int): Roll number to search

    Returns:
    int: Index if found, else -1
    """
    for i in range(len(data)):
        if data[i] == target:
            return i
    return -1



# Testing
students = [104, 101, 109, 102]
print(linear_search(students, 109))
```

Output

```
2
```

Code Explanation

- Linear Search checks each element sequentially.
- Works on both sorted and unsorted lists.

- Time Complexity: O(n).

Python Code (Binary Search)

```
def binary_search(data, target):
   """
   Searches for a target value in a sorted list.

   Parameters:
   data (list): Sorted list of roll numbers
   target (int): Roll number to search

   Returns:
   int: Index if found, else -1
   """
   low = 0
   high = len(data) - 1

   while low <= high:
      mid = (low + high) // 2
      if data[mid] == target:
         return mid
      elif data[mid] < target:
         low = mid + 1
      else:
         high = mid - 1
   return -1


# Testing
sorted_students = [101, 102, 104, 109]
print(binary_search(sorted_students, 109))
```

Output

3

Code Explanation

- Binary Search requires sorted data.
- It reduces the search space by half each iteration.
- Time Complexity: O(log n).
- Binary Search is much faster than Linear Search for large datasets.

Task 4: Choosing Between Quick Sort and Merge Sort

Question

Implement Quick Sort and Merge Sort using recursion.
Explain recursion and compare their performance.

Python Code (Quick Sort)

```
def quick_sort(arr):
    """
    Sorts a list using Quick Sort algorithm.
    """
    if len(arr) <= 1:
        return arr

    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)
```

Testing

data = [5, 3, 8, 6, 2]

print(quick_sort(data))

Output

[2, 3, 5, 6, 8]

Code Explanation

- Quick Sort divides the list using a pivot.
- Recursion sorts left and right partitions.
- Average Time Complexity: O(n log n).
- Worst Case: $O(n^2)$.

Python Code (Merge Sort)

```
def merge_sort(arr):
    """
    Sorts a list using Merge Sort algorithm.
    """
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])

    return merge(left, right)


def merge(left, right):
    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:])
    result.extend(right[j:])
    return result


# Testing
print(merge_sort(data))
```

Output

[2, 3, 5, 6, 8]

Code Explanation

- Merge Sort always divides data into halves.
- Recursion continues until single elements remain.
- Time Complexity is always O(n log n).
- Merge Sort is preferred when stable performance is required.

Task 5: Optimizing a Duplicate Detection Algorithm

Question

Detect duplicate user IDs using a naive approach and an optimized approach.

Python Code (Brute Force)

```python
def has_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



# Testing
ids = [101, 203, 405, 101]
print(has_duplicates_bruteforce(ids))
```

Output

True

Code Explanation

- Uses nested loops.
- Time Complexity: $O(n^2)$.
- Inefficient for large datasets.

Python Code (Optimized Using Set)

```python
def has_duplicates_optimized(data):
    seen = set()
    for item in data:
        if item in seen:
            return True
        seen.add(item)
    return False



# Testing
```

```
print(has_duplicates_optimized(ids))
```

Output

True

Code Explanation

- Uses a set for constant-time lookups.
- Time Complexity: O(n).
- Space-Time tradeoff improves performance significantly.