AI ASSIST CODING

ASSIGNMENT-12.4

KIRANMAI VENDI

2303A52506

BATCH NO:15

Algorithms with AI Assistance – Sorting, Searching, and Optimizing Algorithms

Lab Objectives

• Apply AI-assisted programming to implement and optimize sorting and searching algorithms.

• Compare different algorithms in terms of efficiency and use cases.

• Understand how AI tools can suggest optimized code and complexity improvements.

Task 1: Bubble Sort for Ranking Exam Scores

PROMPT:

Implement Bubble Sort in Python.
Add inline comments explaining comparisons, swaps, and iteration passes.
Include early termination optimization.
Provide time complexity analysis.

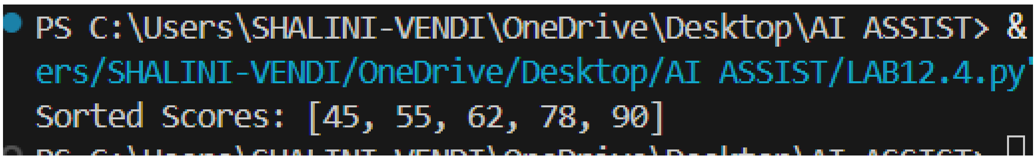
PYTHON CODE:

```
def bubble_sort(scores):
    n = len(scores)
    for i in range(n):
        swapped = False  # Optimization flag
        # Inner loop compares adjacent elements
        for j in range(0, n - i - 1):


            # Compare adjacent elements
            if scores[j] > scores[j + 1]
```

```
            # Swap if out of order
            scores[j], scores[j + 1] = scores[j + 1], scores[j]
            swapped = True
        # If no swaps happened, list is already sorted
        if not swapped:
            break
    return scores
# Sample Execution
scores = [78, 45, 90, 62, 55]
print("Sorted Scores:", bubble_sort(scores))
```

OUTPUT:



```
PS C:\Users\SHALINI-VENDI\OneDrive\Desktop\AI ASSIST> &
ers/SHALINI-VENDI/OneDrive/Desktop/AI ASSIST/LAB12.4.py
Sorted Scores: [45, 55, 62, 78, 90]
```

Task 2: Improving Sorting for Nearly Sorted Attendance Records

Expalanation:

Insertion Sort :

- It performs efficiently on nearly sorted data
- Best Case = O(n)
- Shifts fewer elements compared to Bubble Sort

Algorithm	Nearly Sorted Performance
Bubble Sort	Still performs many comparisons
Insertion Sort	Very few shifts → faster

- Insertion Sort is preferred for nearly sorted datasets.

PYTHON CODE:

#Bubble Sort

```
def bubble_sort(arr):
    for i in range(len(arr)):
        for j in range(len(arr) - i - 1):
```

```
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr
data = [1, 2, 3, 5, 4]
print("Bubble Sort Result:", bubble_sort(data))
```

OUTPUT:

[1, 2, 3, 4, 5]

#Insertion Sort

```
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        # Shift elements greater than key
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
    return arr
data = [1, 2, 3, 5, 4]
print("Insertion Sort Result:", insertion_sort(data))
```

OUTPUT:

```
Insertion Sort Result: [1, 2, 3, 4, 5]
```

Task 3: Searching Student Records in a Database

 # Linear Search (Unsorted Data):

```
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i
```

```
    return -1
arr = [101, 104, 102, 105]
target = 102
print("Linear Search Result:", linear_search(arr, target))
```

OUTPUT:

Linear Search Result: 2

 # Binary Search (Sorted Data):

```
def binary_search(arr, target):
    left, right = 0, len(arr) - 1

    while left <= right:
        mid = (left + right) // 2

        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1

    return -1
arr = [101, 102, 103, 104, 105]
target = 104
sorted_arr = sorted(arr)
print("Binary Search Result:", binary_search(sorted_arr, target))
```

OUTPUT:

Binary Search Result: 3

Explanation

Search Type	When to Use
Linear Search	Unsorted data
Binary Search	Sorted data only

Binary Search is much faster for large sorted datasets.

Task 4: Choosing Between Quick Sort and Merge Sort for Data Processing

 # Quick Sort:

```
def quick_sort(arr):
    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)
data = [34, 7, 23, 32, 5, 62]
print("Quick Sort Result:", quick_sort(arr))
```

OUTPUT:

Quick Sort Result: [5, 7, 23, 32, 34, 62]

#Merge Sort:

```
def merge_sort(arr):
    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
arr = [34, 7, 23, 32, 5, 62]
print("Merge Sort Result:", merge_sort(arr)
```

OUTPUT:

Merge Sort Result: [5, 7, 23, 32, 34, 62]

Complexity 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

#Brute Force Method ($O(n^2)$)

```
def find_duplicates_brute(arr):
    duplicates = []
    for i in range(len(arr)):
        for j in range(i + 1, len(arr)):
```

```
            if arr[i] == arr[j] and arr[i] not in duplicates:
                duplicates.append(arr[i])
    return duplicates
data = [10, 20, 30, 10, 40, 20, 50]
print("Brute Force Duplicates:", find_duplicates_brute(data))
```

OUTPUT:

Duplicates (Brute Force): [10, 20]

#Optimized Method Using Set (O(n))

```
def find_duplicates_optimized(arr):
    seen = set()
duplicates = set()
    for item in arr:
        if item in seen:
            duplicates.add(item)
        else:
            seen.add(item)
    return list(duplicates)
data = [10, 20, 30, 10, 40, 20, 50]
print("Optimized Duplicates:", find_duplicates_optimized(data))
```

OUTPUT:

Duplicates (Optimized): [10, 20]

Performance Comparison

Method	**Time Complexity**	**Efficiency**
Brute Force	$O(n^2)$	Slow for large data
Optimized	$O(n)$	Much faster