2303a51415

Batch:03

Lab 12: Algorithms with AI Assistance

Task 1: Merge Sort Implementation

Prompt:

Generate a Python function merge_sort(arr) that sorts a list in ascending order. Include time and space complexity in the docstring and provide test cases.

Code:

```
def merge_sort(arr):
    """
    Sorts a list in ascending order using Merge Sort.

    Time Complexity:
    Best Case: O(n log n)
    Average Case: O(n log n)
    Worst Case: O(n log n)

    Space Complexity:
    O(n)
    """
    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


# Test Case
print(merge_sort([38, 27, 43, 3, 9, 82, 10]))
```

Output:

[3, 9, 10, 27, 38, 43, 82]

Justification:

Merge Sort is a divide-and-conquer algorithm with guaranteed O(n log n) performance in all cases. It is efficient for large datasets and stable in nature.

Task 2: Binary Search Implementation

Prompt:

Generate a Python function binary_search(arr, target) that returns the index of the target element in a sorted list or -1 if not found. Include complexity analysis.

Code:

```
def binary_search(arr, target):
    """
    Performs Binary Search on a sorted list.

    Returns:
    Index of target if found, else -1.
```

```
Time Complexity:
Best Case: O(1)
Average Case: O(log n)
Worst Case: O(log n)

Space Complexity:
O(1)
"""
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
```

```
# Test Case
print(binary_search([1, 3, 5, 7, 9, 11], 7))
```

Output:

3

Justification:

Binary Search is efficient for sorted data with O(log n) complexity. It significantly reduces search time compared to linear search.

Task 3: Smart Healthcare Appointment Scheduling System

Prompt:

Recommend suitable searching and sorting algorithms for appointment records. Implement searching by appointment ID and sorting by time or consultation fee.

Code:

```
appointments = [
    {"id": 101, "patient": "Amit", "doctor": "Dr. Rao", "time": "10:00", "fee": 500},
    {"id": 102, "patient": "Sneha", "doctor": "Dr. Kumar", "time": "09:30", "fee": 700},
    {"id": 103, "patient": "Rahul", "doctor": "Dr. Mehta", "time": "11:00", "fee": 600}
]

def search_appointment(appointments, appointment_id):
    for appt in appointments:
    if appt["id"] == appointment_id:
    return appt
    return None

def sort_by_fee(appointments):
    return sorted(appointments, key=lambda x: x["fee"])

def sort_by_time(appointments):
    return sorted(appointments, key=lambda x: x["time"])

print(search_appointment(appointments, 102))
print(sort_by_fee(appointments))
```

Output:

{'id': 102, 'patient': 'Sneha', 'doctor': 'Dr. Kumar', 'time': '09:30', 'fee': 700}

Justification:

Linear search is suitable for small datasets. Python's built-in sorted() uses Timsort (O(n log n)) which is stable and efficient for real-world data.

Task 4: Railway Ticket Reservation System

Prompt:

Identify efficient algorithms for searching ticket ID and sorting by travel date or seat number. Implement them in Python.

Code:

```
tickets = [
    {"ticket_id": 201, "name": "Arjun", "train_no": 12627, "seat_no": 45, "date":
```

```
"2026-03-10"},
        {"ticket_id": 202, "name": "Priya", "train_no": 12628, "seat_no": 12, "date":
"2026-03-08"},
        {"ticket_id": 203, "name": "Kiran", "train_no": 12629, "seat_no": 30, "date":
"2026-03-12"}
 ]

 def search_ticket(tickets, ticket_id):
        for ticket in tickets:
        if ticket["ticket_id"] == ticket_id:
        return ticket
        return None

 def sort_by_date(tickets):
        return sorted(tickets, key=lambda x: x["date"])

 def sort_by_seat(tickets):
        return sorted(tickets, key=lambda x: x["seat_no"])

 print(search_ticket(tickets, 202))
 print(sort_by_date(tickets))
```

Output:

{'ticket_id': 202, 'name': 'Priya', 'train_no': 12628, 'seat_no': 12, 'date': '2026-03-08'}

Justification:

For moderate datasets, linear search is sufficient. Sorting by date and seat number using Timsort ensures O(n log n) efficiency and stability.