AI Assisted Coding (III Year) Assignment

Name: G.Rushindhra Goud

HT NO: 2303A52199

Batch: 35

Assignment Number: 12.1 / 24

Lab 12: Algorithms with AI Assistance – Sorting, Searching, and Optimizing Algorithms

Lab Objectives

- **Use AI to assist in designing and implementing fundamental data structures in Python**
- **Learn prompt-based structure creation, optimization, and documentation**
- **Improve understanding of Lists, Stacks, Queues, Linked Lists, Graphs, and Hash Tables**
- **Enhance readability and performance with AI-generated suggestions**

Task 1: Merge Sort Implementation

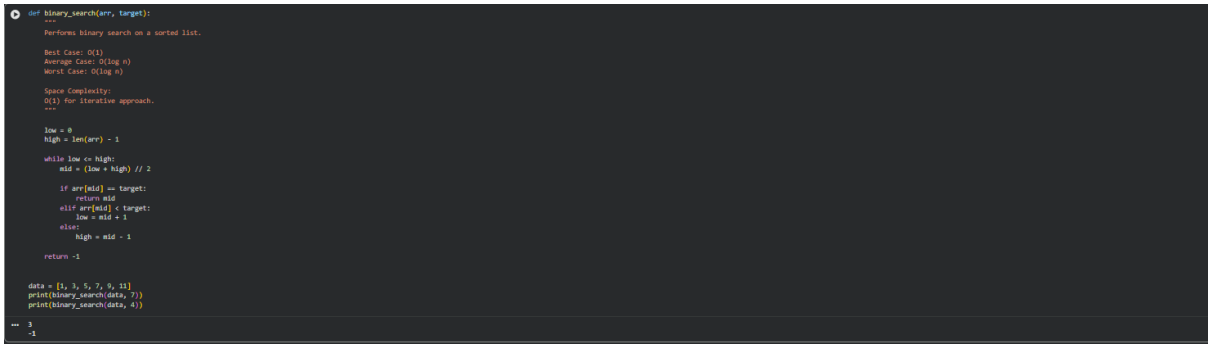
Code Explanation

Merge Sort is a divide and conquer algorithm that divides the list into smaller parts until each sublist contains only one element. Then it merges the sublists by comparing elements and arranging them in sorted order. This process continues until the complete list is sorted. The algorithm has a consistent time complexity of O(n log n) in all cases, which makes it efficient for large datasets. However, it requires extra memory during the merging process, so its space complexity is O(n). It is also a stable sorting algorithm.

AI Prompt Used

Write a Python function named merge_sort(arr) to implement the Merge Sort algorithm. Include proper docstrings explaining time and space complexity and provide test cases.

Task 2: Binary Search



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

    Best Case: O(1)
    Average Case: O(log n)
    Worst Case: O(log n)

    Space Complexity:
    O(1) for iterative approach.
    """

    low = 0
    high = len(arr) - 1

    while low <= high:
        mid = (low + high) // 2

        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1

    return -1


data = [1, 3, 5, 7, 9, 11]
print(binary_search(data, 7))
print(binary_search(data, 4))
```

```
3
-1
```

Code Explanation

Binary Search works only on sorted data and reduces the search space by half in every step. It compares the target element with the middle element of the list and continues the search in the left or right half based on comparison. This makes the algorithm much faster than linear search for large datasets. The best case occurs when the element is found in the middle, which takes O(1) time, while the average and worst cases take O(log n). Since this implementation uses an iterative approach, the space complexity is O(1).

AI Prompt Used

Generate a Python function binary_search(arr, target) that returns the index of the target element or -1 if not found. Include docstrings explaining best, average, and worst-case complexity.

Task 3: Healthcare Appointment Scheduling

Recommended Algorithms

Operation	Algorithm	Justification
Search by Appointment ID	Hashing	Fast lookup
Sort by Time	Merge Sort	Stable and efficient
Sort by Fee	Merge or Quick Sort	Efficient for large data

```
appointments = [
    {"id": 1, "patient": "A", "doctor": "Dr X", "time": 10, "fee": 500},
    {"id": 2, "patient": "B", "doctor": "Dr Y", "time": 8, "fee": 300},
    {"id": 3, "patient": "C", "doctor": "Dr Z", "time": 12, "fee": 700}
]

appointment_map = {a["id"]: a for a in appointments}

def search_appointment(aid):
    return appointment_map.get(aid, "Not found")

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

print(search_appointment(2))
print(sort_by_time(appointments))
```

```
[{'id': 1002, 'restaurant': 'B', 'time': 20, 'price': 150}, {'id': 1001, 'restaurant': 'A', 'time': 30, 'price': 250}, {'id': 1003, 'restaurant': 'C', 'time': 40, 'price': 300}]
```

Code Explanation

In healthcare systems, fast appointment lookup is essential. Hashing provides near constant time access to appointment records. Sorting by time ensures efficient scheduling, while sorting by consultation fee supports financial analysis. These algorithms improve performance and reliability.

AI Prompt Used

Suggest efficient searching and sorting algorithms for an appointment system and implement Python functions with justification.

Task 4: Railway Ticket Reservation

Recommended Algorithms

Operation	Algorithm	Justification
Search by Ticket ID	Hashing	Instant retrieval
Sort by Travel Date	Merge Sort	Handles large records

Sort by Seat Number	**Quick Sort**	**Efficient**

```
tickets = [
    {"id": 101, "name": "P", "train": 123, "seat": 45, "date": 5},
    {"id": 102, "name": "Q", "train": 456, "seat": 12, "date": 2},
    {"id": 103, "name": "R", "train": 789, "seat": 30, "date": 7}
]

ticket_map = {t["id"]: t for t in tickets}

def search_ticket(tid):
    return ticket_map.get(tid, "Not found")

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

print(search_ticket(102))
print(sort_by_date(tickets))
```

```
{'id': 102, 'name': 'Q', 'train': 456, 'seat': 12, 'date': 2}
[{'id': 102, 'name': 'Q', 'train': 456, 'seat': 12, 'date': 2}, {'id': 101, 'name': 'P', 'train': 123, 'seat': 45, 'date': 5}, {'id': 103, 'name': 'R', 'train': 789, 'seat': 30, 'date': 7}]
```

Code Explanation

Railway systems require quick ticket lookup to avoid delays. Hashing provides fast search using ticket ID. Sorting by travel date helps manage schedules, while sorting by seat number supports booking optimization. Efficient algorithms ensure smooth system performance.

AI Prompt Used

Identify suitable algorithms for railway booking search and sorting and implement Python solutions.

Task 5: Smart Hostel Room Allocation System

Recommended Algorithms

Operation	Algorithm	Justification
Search by Student ID	**Hashing**	**Fast lookup**
Sort by Room Number	**Merge Sort**	**Organized**
Sort by Allocation Date	**Merge Sort**	**Efficient**

```
rooms = [
    {"student": 1, "room": 101, "floor": 1, "date": 3},
    {"student": 2, "room": 205, "floor": 2, "date": 1},
    {"student": 3, "room": 102, "floor": 1, "date": 2}
]

room_map = {r["student"]: r for r in rooms}

def search_room(sid):
    return room_map.get(sid, "Not found")

def sort_by_room(rooms):
    return sorted(rooms, key=lambda x: x["room"])

print(search_room(1))
print(sort_by_room(rooms))
```

```
{'student': 1, 'room': 101, 'floor': 1, 'date': 3}
[{'student': 1, 'room': 101, 'floor': 1, 'date': 3}, {'student': 3, 'room': 102, 'floor': 1, 'date': 2}, {'student': 2, 'room': 205, 'floor': 2, 'date': 1}]
```

Code Explanation

Hostel systems require efficient room allocation and tracking. Hashing enables fast student lookup, while sorting by room or allocation date ensures proper organization. These algorithms improve management efficiency.

AI Prompt Used

Suggest optimized algorithms for hostel allocation and implement Python functions.

Task 6: Online Movie Streaming Platform

Recommended Algorithms

Operation	Algorithm	Justification
Search by Movie ID	Hashing	Instant
Sort by Rating	Merge or Quick Sort	Ranking
Sort by Release Year	Merge Sort	Efficient

```
movies = [
    {"id": 1, "title": "M1", "rating": 8.5, "year": 2020},
    {"id": 2, "title": "M2", "rating": 7.2, "year": 2018},
    {"id": 3, "title": "M3", "rating": 9.1, "year": 2022}
]

movie_map = {m["id"]: m for m in movies}

def search_movie(mid):
    return movie_map.get(mid, "Not found")

def sort_by_rating(movies):
    return sorted(movies, key=lambda x: x["rating"], reverse=True)

print(search_movie(3))
print(sort_by_rating(movies))
```

```
{'id': 3, 'title': 'M3', 'rating': 9.1, 'year': 2022}
[{'id': 3, 'title': 'M3', 'rating': 9.1, 'year': 2022}, {'id': 1, 'title': 'M1', 'rating': 8.5, 'year': 2020}, {'id': 2, 'title': 'M2', 'rating': 7.2, 'year': 2018}]
```

Code Explanation

Streaming platforms need fast movie lookup and efficient ranking. Hashing enables instant search, while sorting by rating and year improves recommendation and user experience. Efficient algorithms ensure scalability.

AI Prompt Used

Recommend searching and sorting algorithms for a movie platform and implement them in Python.

Task 7: Smart Agriculture Crop Monitoring System

Recommended Algorithms

Operation	Algorithm	Justification
Search by Crop ID	Hashing	Fast
Sort by Moisture	Merge Sort	Accurate
Sort by Yield	Merge Sort	Efficient

```
crops = [
    {"id": 1, "name": "Rice", "moisture": 60, "yield": 90},
    {"id": 2, "name": "Wheat", "moisture": 40, "yield": 70},
    {"id": 3, "name": "Corn", "moisture": 55, "yield": 85}
]

crop_map = {c["id"]: c for c in crops}

def search_crop(cid):
    return crop_map.get(cid, "Not found")

def sort_by_moisture(crops):
    return sorted(crops, key=lambda x: x["moisture"])

print(search_crop(2))
print(sort_by_moisture(crops))
```

```
{'id': 2, 'name': 'Wheat', 'moisture': 40, 'yield': 70}
[{'id': 2, 'name': 'Wheat', 'moisture': 40, 'yield': 70}, {'id': 3, 'name': 'Corn', 'moisture': 55, 'yield': 85}, {'id': 1, 'name': 'Rice', 'moisture': 60, 'yield': 90}]
```

Code Explanation

Agriculture monitoring requires fast crop analysis. Hashing helps retrieve crop data quickly. Sorting by moisture and yield helps farmers make better irrigation and planning decisions. Efficient algorithms improve productivity.

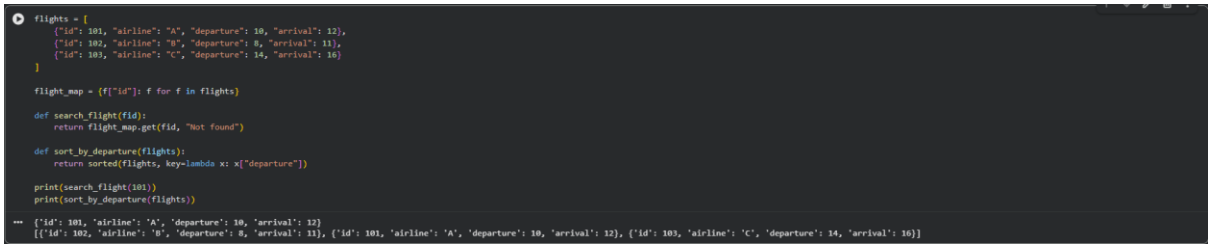
AI Prompt Used

Suggest efficient algorithms for crop monitoring and implement Python code.

Task 8: Airport Flight Management System

Recommended Algorithms

Operation	Algorithm	Justification
Search by Flight ID	Hashing	Real-time
Sort by Departure	Merge Sort	Scheduling
Sort by Arrival	Merge Sort	Efficient



```
flights = [
    {"id": 101, "airline": "A", "departure": 10, "arrival": 12},
    {"id": 102, "airline": "B", "departure": 8, "arrival": 11},
    {"id": 103, "airline": "C", "departure": 14, "arrival": 16}
]

flight_map = {f["id"]: f for f in flights}

def search_flight(fid):
    return flight_map.get(fid, "Not found")

def sort_by_departure(flights):
    return sorted(flights, key=lambda x: x["departure"])

print(search_flight(101))
print(sort_by_departure(flights))
```

```
{'id': 101, 'airline': 'A', 'departure': 10, 'arrival': 12}
[{'id': 102, 'airline': 'B', 'departure': 8, 'arrival': 11}, {'id': 101, 'airline': 'A', 'departure': 10, 'arrival': 12}, {'id': 103, 'airline': 'C', 'departure': 14, 'arrival': 16}]
```

Code Explanation

Airport systems must handle real-time scheduling. Hashing allows fast flight lookup. Sorting by departure and arrival times improves planning and reduces delays. Efficient algorithms ensure smooth operations.

AI Prompt Used

Recommend searching and sorting algorithms for an airport system and implement Python functions.

Conclusion

This lab demonstrates the role of AI in selecting efficient searching and sorting algorithms for real-world systems. Hashing provides fast data retrieval, while Merge Sort and Quick Sort ensure scalability and performance in large datasets. These techniques improve system efficiency and reliability