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

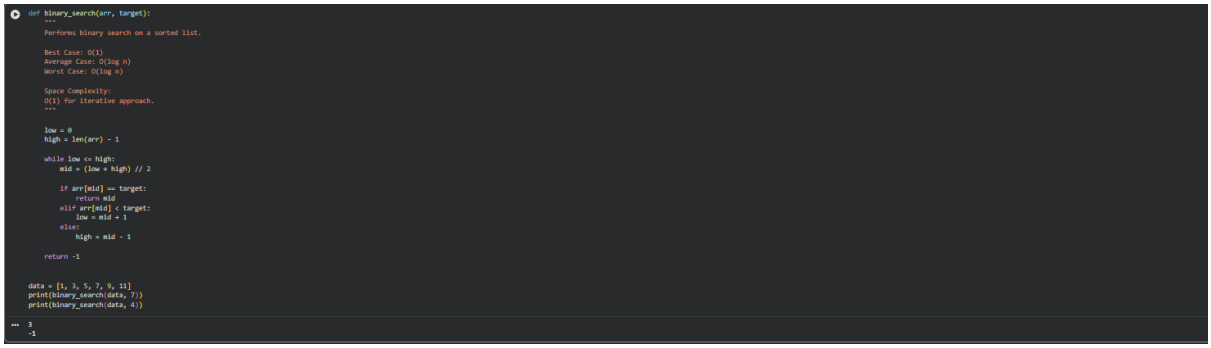
Code Explanation

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).

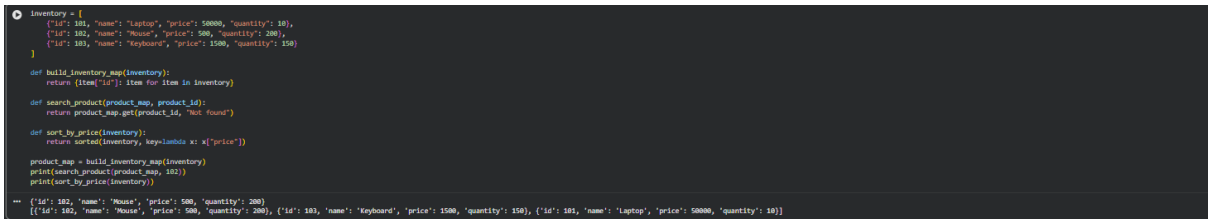
AI Prompt Used

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: Inventory Management System

Recommended Algorithms

Operation	Algorithm	Justification
Search by Product ID	Hashing	Fast lookup
Search by Name	Hash Table	Efficient retrieval
Sort by Price	Merge Sort	Stable and efficient
Sort by Quantity	Merge Sort	Suitable for large data



```
inventory = [
    {"id": 101, "name": "Laptop", "price": 50000, "quantity": 10},
    {"id": 102, "name": "Mouse", "price": 500, "quantity": 200},
    {"id": 103, "name": "Keyboard", "price": 1500, "quantity": 150}
]

def build_inventory_map(inventory):
    return {item["id"]: item for item in inventory}

def search_product(product_map, product_id):
    return product_map.get(product_id, "Not found")

def sort_by_price(inventory):
    return sorted(inventory, key=lambda x: x["price"])

product_map = build_inventory_map(inventory)
print(search_product(product_map, 102))
print(sort_by_price(inventory))
```

```
{'id': 102, 'name': 'Mouse', 'price': 500, 'quantity': 200}
[{'id': 102, 'name': 'Mouse', 'price': 500, 'quantity': 200}, {'id': 103, 'name': 'Keyboard', 'price': 1500, 'quantity': 150}, {'id': 101, 'name': 'Laptop', 'price': 50000, 'quantity': 10}]
```

Code Explanation

Large inventory systems require fast searching and sorting. Hashing is used to search products quickly using product IDs because it provides constant time complexity on average. Python dictionaries work efficiently for this purpose. Sorting products based on price or quantity helps in analysis and decision-making, and Merge Sort or Python's built-in sorting ensures stability and efficiency. This approach improves performance even when the dataset grows large.

AI Prompt Used

Suggest efficient searching and sorting algorithms for an inventory system with thousands of products. Implement Python functions and justify the selection.

Task 4: Smart Hospital Patient Management System

Recommended Algorithms

Operation	Algorithm	Justification
Search by Patient ID	Hashing	Instant access
Search by Name	Hash Table	Efficient
Sort by Severity	Priority sorting	Critical cases first

Sort by Bill **Merge Sort** **Efficient**

```python
class HashTable:
    """Hash table using chaining."""

    def __init__(self, size=10):
        self.size = size
        self.table = [[] for _ in range(size)]

    def _hash(self, key):
        return hash(key) % self.size

    def insert(self, key, value):
        """Insert key-value."""
        index = self._hash(key)
        for pair in self.table[index]:
            if pair[0] == key:
                pair[1] = value
                return
        self.table[index].append([key, value])

    def search(self, key):
        """Search value."""
        index = self._hash(key)
        for pair in self.table[index]:
            if pair[0] == key:
                return pair[1]
        return None

    def delete(self, key):
        """Delete key."""
        index = self._hash(key)
        for i, pair in enumerate(self.table[index]):
            if pair[0] == key:
                self.table[index].pop(i)
                return
```

Task 5: University Examination Result Processing

```python
students = [
    {"roll": 1, "name": "X", "marks": 80},
    {"roll": 2, "name": "Y", "marks": 95},
    {"roll": 3, "name": "Z", "marks": 60}
]

student_map = {s["roll"]: s for s in students}

def search_student(roll):
    return student_map.get(roll, "Not found")

def rank_students(students):
    return sorted(students, key=lambda x: x["marks"], reverse=True)

print(rank_students(students))
```

```
[{'roll': 2, 'name': 'Y', 'marks': 95}, {'roll': 1, 'name': 'X', 'marks': 80}, {'roll': 3, 'name': 'Z', 'marks': 60}]
```

Code Explanation

Large universities process thousands of results, so fast searching is essential. Hashing provides efficient retrieval of student records. Sorting based on marks helps generate rank lists quickly and accurately. Efficient algorithms reduce processing time and improve academic management.

AI Prompt Used

Identify efficient searching and sorting methods for university results and implement Python functions with justification.

Task 6: Online Food Delivery Platform

Recommended Algorithms

Operation	Algorithm	Justification
Search by Order ID	Hashing	Real-time system
Sort by Delivery Time	Merge or Heap Sort	Priority handling
Sort by Price	Quick or Merge Sort	Efficient for large data

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

order_map = {o["id"]: o for o in orders}

def find_order(order_id):
    return order_map.get(order_id, "Not found")

def sort_by_delivery(orders):
    return sorted(orders, key=lambda x: x["time"])

print(sort_by_delivery(orders))
```

```
[{'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

Food delivery platforms handle real-time data, so quick search and sorting are critical. Hashing enables instant order lookup. Sorting by delivery time improves efficiency and customer satisfaction, while sorting by price supports business analysis. Efficient algorithms ensure scalability and faster performance.

AI Prompt Used

Suggest optimized algorithms for searching and sorting in an online food delivery platform and implement them in Python with justification.

Conclusion

This lab demonstrates how AI can assist in selecting and implementing efficient searching and sorting algorithms. Merge Sort and Binary Search provide strong performance for large datasets, while hashing ensures fast real-time lookup. These techniques improve scalability, performance, and reliability in real-world applications.