AI ASSISTED CODING

LAB-12*: Algorithms with AI Assistance – Sorting, Searching, and Optimizing Algorithms*

Name: G Hasini

Roll No:2503A51L41

Batch:24BTCAICSB20

Task Description #1:

(Sorting – Merge Sort Implementation)
• Task: Use AI to generate a Python program that implements the Merge Sort algorithm.
• Instructions:
o Prompt AI to create a function merge_sort(arr) that sorts a list in ascending order.
oAsk AI to include time complexity and space complexity in the function docstring.
oVerify the generated code with test cases.
• Expected Output:
o A functional Python script implementing Merge Sort with proper documentation

PROMPT:*Generate a Python program that implements the Merge Sort algorithm create a function merge_sort(arr) that sorts a list in ascending order include time complexity and space complexity in the function docstring.Verify the generated code with test cases.*

task12-1.py > ...

```python
def merge_sort(arr):
    """
    Sorts a list in ascending order using the Merge Sort algorithm.
    Time Complexity: O(n log n) in all cases (best, average, worst).
    Space Complexity: O(n) due to the use of temporary arrays.
    Args:
        arr (list): The list of elements to be sorted.
    Returns:
        list: A new sorted list in ascending order.
    """
    if len(arr) <= 1:
        return arr
    # Find the middle point and divide the array
    mid = len(arr) // 2
    left_half = merge_sort(arr[:mid])
    right_half = merge_sort(arr[mid:])
    # Merge the sorted halves
    return merge(left_half, right_half)
def merge(left, right):
    """
    Merges two sorted lists into one sorted list.
    Args:
        left (list): First sorted list.
        right (list): Second sorted list.
    Returns:
        list: Merged sorted list.
    """
    merged = []
    i = j = 0
    # Compare elements from both lists and append the smaller one
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            merged.append(left[i])
            i += 1
        else:
            merged.append(right[j])
            j += 1
    # Append any remaining elements
    merged.extend(left[i:])
    merged.extend(right[j:])
    return merged
# Test cases to verify the implementation
if __name__ == "__main__":
    test_cases = [
        [],
        [1],
        [5, 2, 9, 1, 5, 6],
        [3, 3, 3],
        [10, -1, 2, 5, 0],
        [1, 2, 3, 4, 5],
        [5, 4, 3, 2, 1]
    ]
    for idx, test in enumerate(test_cases):
        print(f"Test case {idx+1}: Input: {test}")
        print(f"Sorted: {merge_sort(test)}\n")
```

OUTPUT:

```
PS C:\Users\DELL\OneDrive\SAHASRA SRU\AI assisted> & C
AHASRA SRU/AI assisted/task12-1.py"
Test case 1: Input: []
Sorted: []

Test case 2: Input: [1]
Sorted: [1]

Test case 3: Input: [5, 2, 9, 1, 5, 6]
Sorted: [1, 2, 5, 5, 6, 9]

Test case 4: Input: [3, 3, 3]
Sorted: [3, 3, 3]

Test case 5: Input: [10, -1, 2, 5, 0]
Sorted: [-1, 0, 2, 5, 10]

Test case 6: Input: [1, 2, 3, 4, 5]
Sorted: [1, 2, 3, 4, 5]

Test case 7: Input: [5, 4, 3, 2, 1]
Sorted: [1, 2, 3, 4, 5]
```

Observation:AI generated the merge_sort(arr) function using the divide-and-conquer strategy. It included a detailed docstring explaining time complexity (O(n log n)) and space complexity (O(n)), which made the algorithm clearer. After running the AI-generated code with test cases, I observed that the list was sorted correctly. This showed me how AI can not only implement the algorithm but also document and verify it systematically

Task Description #2

(Searching – Binary Search with AI
Optimization)
• Task: Use AI to create a binary search function that finds a target element in a sorted list.
• Instructions:
o Prompt AI to create a function binary_search(arr, target) returning the index of the target or -1 if not found.
oInclude docstrings explaining best, average, and worst-case complexities.
oTest with various inputs.
• Expected Output:
o Python code implementing binary search with AI-generated comments and docstrings.

PROMPT*:create a binary search function that finds a target element in a sorted list.create a function binary_search(arr, target) returning the index of the target or -1 if not found.Includedocstrings explaining best, average, and worst-case complexities.Test with various inputs.*

task12-2.py > binary_search

```python
def binary_search(arr, target):
    """
    Performs binary search to find the index of a target element in a sorted list.
    Time Complexity:
        Best case: O(1)      (target is at the middle)
        Average case: O(log n)
        Worst case: O(log n)
    Space Complexity: O(1) (iterative approach)
    Args:
        arr (list): A sorted list of elements.
        target: The element to search for.
    Returns:
        int: The index of the target if found, else -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 cases to verify the implementation
if __name__ == "__main__":
    test_cases = [
        ([], 5),
        ([1], 1),
        ([1, 2, 3, 4, 5], 3),
        ([1, 2, 3, 4, 5], 6),
        ([10, 20, 30, 40, 50], 10),
        ([10, 20, 30, 40, 50], 50),
        ([10, 20, 30, 40, 50], 35),
        ([2, 4, 6, 8, 10, 12], 8),
    ]
    for idx, (arr, target) in enumerate(test_cases):
        result = binary_search(arr, target)
        print(f"Test case {idx+1}: Array: {arr}, Target: {target} -> index: {result}")
```

OUTPUT:

```
AHASRA SRU/AI assisted/task12-2.py"
Test case 1: Array: [], Target: 5 -> index: -1
Test case 2: Array: [1], Target: 1 -> index: 0
Test case 3: Array: [1, 2, 3, 4, 5], Target: 3 -> index: 2
Test case 4: Array: [1, 2, 3, 4, 5], Target: 6 -> index: -1
Test case 5: Array: [10, 20, 30, 40, 50], Target: 10 -> index: 0
Test case 6: Array: [10, 20, 30, 40, 50], Target: 50 -> index: 4
Test case 7: Array: [10, 20, 30, 40, 50], Target: 35 -> index: -1
Test case 8: Array: [2, 4, 6, 8, 10, 12], Target: 8 -> index: 3
PS C:\Users\DELL\OneDrive\SAHASRA SRU\AI assisted>
```

Observation:AI implemented the binary_search(arr, target) function by repeatedly dividing the sorted list into halves. The function returned the correct index if the element was found, and -1 otherwise. The AI also explained best, average, and worst-case complexities directly in the docstring, which made it easier to connect theory with practice. Testing with multiple inputs confirmed the accuracy, and I learned how AI-generated code can be both optimized and self-explanatory.

Task Description #3

(Real-Time Application – Inventory
Management System)
• Scenario: A retail store's inventory system contains thousands of products, each with attributes like product ID, name, price, and stock quantity. Store staff need to:
1. Quickly search for a product by ID or name.
2. Sort products by price or quantity for stock analysis.
• Task:
o Use AI to suggest the most efficient search and sort algorithms for this use case.
oImplement the recommended algorithms in Python.
oJustify the choice based on dataset size, update frequency, and performance requirements

Expected Output:
o A table mapping operation → recommended algorithm → justification.
oWorking Python functions for searching and sorting the inventory

PROMPT:*Scenario: A retail store's inventory system contains thousands of products, each with attributes like product ID, name, price, and stock quantity. Store staff need to:*

1. Quickly search for a product by ID or name.

2. Sort products by price or quantity for stock analysis.

suggest the most efficient search and sort algorithms for this use case.

o Implement the recommended algorithms in Python.

o Justify the choice based on dataset size, update frequency, and performance requirements

TASK12-3.PY > sort_by_quantity

```python
# --- Algorithm Justification ---
# For searching by Product ID or Name, we use Python dictionaries (hash tables).
#   - Dictionary lookup is O(1) on average, making searches extremely fast even for thousands of products.
#   - This is ideal for inventory systems where staff need instant access to product details.
# For sorting by price or quantity, we use Python's built-in sorted() function (Timsort).
#   - Timsort has O(n log n) time complexity and is highly efficient for real-world data.
#   - Sorting is performed on-demand for analysis, so it does not impact update speed.
# These choices are optimal for:
#   - Dataset size: Thousands of products fit easily in memory.
#   - Update frequency: Dictionaries allow fast updates and lookups.
#   - Performance: Staff get instant search and analysis results.
class Product:
    def __init__(self, product_id, name, price, quantity):
        self.product_id = product_id
        self.name = name
        self.price = price
        self.quantity = quantity
    def __repr__(self):
        return f"{self.name} (ID: {self.product_id}, Price: {self.price}, Qty: {self.quantity})"
# Sample inventory
inventory = [
    Product(101, "Apple", 30, 50),
    Product(102, "Banana", 10, 100),
    Product(103, "Orange", 20, 80),
    Product(104, "Milk", 50, 30),
    Product(105, "Bread", 25, 60),
]
# Build a dictionary for fast ID lookup (O(1) average-case)
product_by_id = {product.product_id: product for product in inventory}
# Build a dictionary for fast name lookup (O(1) average-case, case-insensitive)
product_by_name = {product.name.lower(): product for product in inventory}

def search_by_id(product_id):
    """O(1) average-case lookup using dictionary."""
    return product_by_id.get(product_id, None)

def search_by_name(name):
    """O(1) average-case lookup using dictionary."""
    return product_by_name.get(name.lower(), None)

def sort_by_price(products, reverse=False):
    """O(n log n) using Timsort."""
    return sorted(products, key=lambda p: p.price, reverse=reverse)

def sort_by_quantity(products, reverse=False):
    """O(n log n) using Timsort."""
    return sorted(products, key=lambda p: p.quantity, reverse=reverse)
# --- User input for interaction ---
if __name__ == "__main__":
    while True:
        print("\nOptions:")
        print("1. Search by Product ID")
        print("2. Search by Product Name")
        print("3. Show Products Sorted by Price (Low to High)")
        print("4. Show Products Sorted by Quantity (High to Low)")
        print("5. Exit")
        choice = input("Enter your choice (1-5): ").strip()
        if choice == "1":
            pid = input("Enter Product ID: ").strip()
            if pid.isdigit():
                result = search_by_id(int(pid))
                print("Result:", result if result else "Product not found.")
            else:
                print("Invalid Product ID.")
        elif choice == "2":
            name = input("Enter Product Name: ").strip()
            result = search_by_name(name)
            print("Result:", result if result else "Product not found.")
        elif choice == "3":
            print("\nProducts sorted by Price (Low to High):")
            for p in sort_by_price(inventory):
                print(p)
        elif choice == "4":
            print("\nProducts sorted by Quantity (High to Low):")
            for p in sort_by_quantity(inventory, reverse=True):
                print(p)
        elif choice == "5":
            print("Exiting.")
            break
        else:
            print("Invalid choice. Please try again.")
```

OUTPUT:

```
Options:
1. Search by Product ID
2. Search by Product Name
3. Show Products Sorted by Price (Low to High)
4. Show Products Sorted by Quantity (High to Low)
5. Exit
Enter your choice (1-5): 5
Enter your choice (1-5): 1
Enter Product ID: 103
Result: Orange (ID: 103, Price: 20, Qty: 80)

Options:
1. Search by Product ID
2. Search by Product Name
3. Show Products Sorted by Price (Low to High)
4. Show Products Sorted by Quantity (High to Low)
5. Exit
Enter your choice (1-5): 5
Result: Orange (ID: 103, Price: 20, Qty: 80)

Options:
1. Search by Product ID
2. Search by Product Name
3. Show Products Sorted by Price (Low to High)
4. Show Products Sorted by Quantity (High to Low)
5. Exit
Enter your choice (1-5): 5
Options:
1. Search by Product ID
2. Search by Product Name
3. Show Products Sorted by Price (Low to High)
4. Show Products Sorted by Quantity (High to Low)
5. Exit
Enter your choice (1-5): 5
2. Search by Product Name
3. Show Products Sorted by Price (Low to High)
4. Show Products Sorted by Quantity (High to Low)
5. Exit
Enter your choice (1-5): 5
3. Show Products Sorted by Price (Low to High)
4. Show Products Sorted by Quantity (High to Low)
5. Exit
Enter your choice (1-5): 5
4. Show Products Sorted by Quantity (High to Low)
5. Exit
Enter your choice (1-5): 5
Exiting.
```

Observation:

For the inventory system scenario, AI suggested using Binary Search for quick product lookups and efficient sorting algorithms (Merge Sort/Quick Sort) for arranging products by price or quantity. It justified these choices based on dataset size and performance requirements. The AI also provided a Python implementation that allowed searching by ID/name and sorting using options. Through this, I observed how AI applies theoretical algorithms to solve real-world problems, while also justifying the decisions with clear reasoning.