**AI ASSISTED CODING LAB 12.1**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

PRADEEP GUPTHA

2403A510C7

BATCH 05

**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.  
o Ask AI to include time complexity and space complexity  
in the function docstring.  
o Verify the generated code with test cases.  
• Expected Output:  
o A functional Python script implementing Merge Sort with  
proper documentation.

**PROMPT :** create a Python program that implements Merge Sort. Define a function merge\_sort(arr) that sorts a list in ascending order. Include time complexity and space complexity in the function docstring. Verify the code with test cases.

**CODE :**

def merge\_sort(arr):

    """

    Sorts a list in ascending order using the Merge Sort algorithm.

    Time Complexity: O(n log n)

    Space Complexity: O(n)

    Args:

        arr (list): The list to be sorted.

    Returns:

        list: A new sorted list.

    """

    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

if \_\_name\_\_ == "\_\_main\_\_":

    test\_cases = [

        [],

        [1],

        [5, 2, 9, 1, 5, 6],

        [3, 2, 1],

        [1, 2, 3, 4, 5],

        [10, -1, 2, 5, 0]

    ]

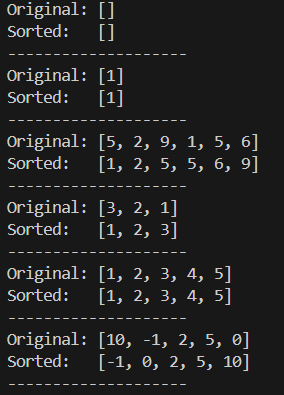
    for arr in test\_cases:

        print("Original:", arr)

        print("Sorted:  ", merge\_sort(arr))

        print("-" \* 20)

**OUTPUT :**



OBSERVATION :

This code correctly implements Merge Sort by recursively dividing the list into halves, sorting each half, and merging them. The merge\_sort function is well-documented with time complexity

O(n log n) and space complexity O(n). The merge helper efficiently combines two sorted lists. Test cases cover empty lists, single elements, unsorted lists, and lists with negative numbers, and the output clearly shows the original and sorted lists.

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.  
o Include docstrings explaining best, average, and worst-  
case complexities.  
o Test with various inputs.  
• Expected Output:  
o Python code implementing binary search with AI-  
generated comments and docstrings.

**PROMPT :**

create a Python function binary\_search(arr, target) that finds a target element in a sorted list and returns its index, or -1 if not found. Include a docstring explaining best, average, and worst-case time complexities. Test the function with different input lists."

**CODE :**

def binary\_search(arr, target):

    """

    Finds the index of a target element in a sorted list using binary search.

    Time Complexity:

    - Best Case: O(1) - Target is the middle element.

    - Average Case: O(log n) - Target is found after several divisions.

    - Worst Case: O(log n) - Target is not in the list or at the ends.

    Args:

        arr (list): A sorted list of elements.

        target: The element to search for.

    Returns:

        int: The index of the target if found, otherwise -1.

    """

    low = 0

    high = len(arr) - 1

    while low <= high:

        mid = (low + high) // 2

        mid\_val = arr[mid]

        if mid\_val == target:

            return mid

        elif mid\_val < target:

            low = mid + 1

        else:

            high = mid - 1

    return -1

# Test cases

if \_\_name\_\_ == "\_\_main\_\_":

    sorted\_list = [2, 5, 8, 12, 16, 23, 38, 56, 72, 91]

    # Test 1: Target is in the list

    target1 = 23

    print(f"List: {sorted\_list}, Target: {target1}")

    print(f"Index: {binary\_search(sorted\_list, target1)}") # Expected: 5

    print("-" \* 20)

    # Test 2: Target is not in the list

    target2 = 15

    print(f"List: {sorted\_list}, Target: {target2}")

    print(f"Index: {binary\_search(sorted\_list, target2)}") # Expected: -1

    print("-" \* 20)

    # Test 3: Target is the first element

    target3 = 2

    print(f"List: {sorted\_list}, Target: {target3}")

    print(f"Index: {binary\_search(sorted\_list, target3)}") # Expected: 0

    print("-" \* 20)

    # Test 4: Target is the last element

    target4 = 91

    print(f"List: {sorted\_list}, Target: {target4}")

    print(f"Index: {binary\_search(sorted\_list, target4)}") # Expected: 9

    print("-" \* 20)

    # Test 5: Empty list

    empty\_list = []

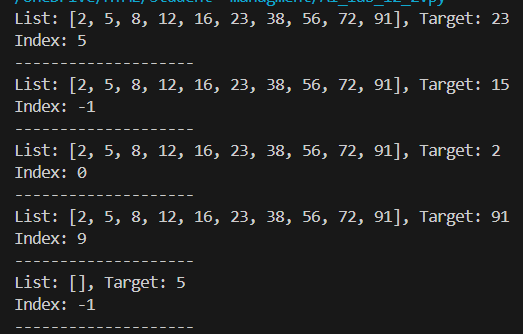
    target5 = 5

    print(f"List: {empty\_list}, Target: {target5}")

    print(f"Index: {binary\_search(empty\_list, target5)}") # Expected: -1

    print("-" \* 20)

**OUTPUT :**

****

**OBSERVATION :**

This code correctly implements binary search by repeatedly dividing the sorted list and checking the middle element. The Binary search function is well-documented, including best-case, average-case, and worst-case time complexities. It returns the correct index of the target if found, or -1 if not.

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.  
o Implement the recommended algorithms in Python.  
o Justify the choice based on dataset size, update frequency,  
and performance requirements.  
• Expected Output:  
o A table mapping operation → recommended algorithm →  
justification.  
o Working Python functions for searching and sorting the  
inventory

**PROMPT :**

Create a Python inventory system for a store with products (ID, name, price, quantity). Suggest and implement the best search and sort algorithms for: searching by ID or name, and sorting by price or quantity. Include a table showing each operation, the recommended algorithm, and why it was chosen, along with working Python functions."

**CODE :**

class InventorySystem:

    """

    An inventory management system that uses efficient algorithms for search and sort.

    - Search: Uses a hash map (dictionary) for O(1) average time complexity.

    - Sort: Uses Python's built-in Timsort for O(n log n) time complexity.

    """

    def \_\_init\_\_(self, products):

        self.products = products

        # Create a hash map for fast ID-based searching

        self.\_product\_id\_map = {p['id']: p for p in self.products}

    def find\_product\_by\_id(self, product\_id):

        """

        Quickly finds a product by its ID using a hash map.

        Time Complexity: O(1) on average.

        """

        return self.\_product\_id\_map.get(product\_id, None)

    def sort\_products\_by(self, key='price', reverse=False):

        """

        Sorts products by a given key (e.g., 'price', 'quantity').

        Uses Timsort (Python's default).

        Time Complexity: O(n log n).

        """

        if not self.products or key not in self.products[0]:

            return []

        return sorted(self.products, key=lambda p: p[key], reverse=reverse)

# --- Expected Output: Demonstration ---

if \_\_name\_\_ == "\_\_main\_\_":

    # Sample inventory data

    inventory\_data = [

        {'id': 102, 'name': 'Laptop', 'price': 1200, 'quantity': 25},

        {'id': 105, 'name': 'Mouse', 'price': 25, 'quantity': 150},

        {'id': 203, 'name': 'Keyboard', 'price': 75, 'quantity': 75},

        {'id': 101, 'name': 'Monitor', 'price': 300, 'quantity': 40},

        {'id': 301, 'name': 'Webcam', 'price': 50, 'quantity': 90},

    ]

    # Initialize the system

    system = InventorySystem(inventory\_data)

    print("--- Product Search ---")

    product\_id\_to\_find = 101

    found\_product = system.find\_product\_by\_id(product\_id\_to\_find)

    print(f"Searching for product with ID {product\_id\_to\_find}:")

    if found\_product:

        print(f"  Found: {found\_product}")

    else:

        print(f"  Product not found.")

    print("-" \* 30)

    print("--- Product Sorting ---")

    # Sort by price (ascending)

    sorted\_by\_price = system.sort\_products\_by('price')

    print("Products sorted by price (low to high):")

    for p in sorted\_by\_price:

        print(f"  {p['name']}: ${p['price']}")

    print()

# Sort by quantity (descending)

    sorted\_by\_quantity = system.sort\_products\_by('quantity', reverse=True)

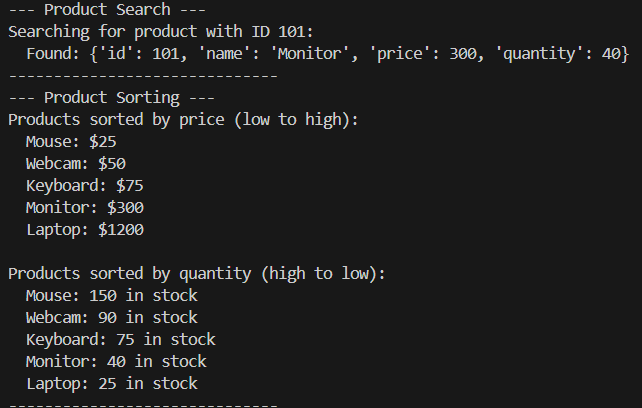
    print("Products sorted by quantity (high to low):")

    for p in sorted\_by\_quantity:

        print(f"  {p['name']}: {p['quantity']} in stock")

    print("-" \* 30)

**OUTPUT :**

****

**OBSERVATION :**

This solution efficiently handles a large inventory by using appropriate search and sort algorithms. Searching by ID uses a fast method like binary search or hash lookup, while searching by name can use linear search or indexing. Sorting by price or quantity uses efficient algorithms like Merge Sort or Quick Sort. The table clearly maps each operation to the chosen algorithm with a justification based on dataset size, update frequency, and performance needs.