**AI ASSISTED CODING**

**LAB ASSISGNMENT-12.1**

**2403A510E4-BATCH 05**

**Lab 12:** Algorithms with AI Assistance – Sorting, Searching, and  
Optimizing Algorithms  
**Lab Objectives**:  
• Apply AI-assisted programming to implement and optimize  
sorting and searching algorithms.  
• Compare different algorithms in terms of efficiency and use  
**Week6 -  
Monday**

cases.  
• Understand how AI tools can suggest optimized code and  
complexity improvements.

**Task Description #1 (Sorting – Merge Sort Implementation)**

**PROMPT:**

Generate a Python program that implements theMerge Sort algorithm.

create a function merge\_sort(arr) that sorts alist in ascending order.

include time complexity and space complexityin the function docstring.

Verify the generated code with test cases.

Expected Output:

A functional Python script implementing Merge Sort withproper documentation.

**GENERATED 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):

    """

    Merges two sorted lists into one sorted list.

    Args:

        left (list): First sorted list.

        right (list): Second sorted list.

    Returns:

        list: Merged sorted list.

    """

    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 cases

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

    test\_cases = [

        [],

        [1],

        [5, 2, 9, 1, 5, 6],

        [3, 3, 3],

        [10, -1, 2, 8, 0, 5],

        [1, 2, 3, 4, 5],

        [5, 4, 3, 2, 1]

    ]

for i, case in enumerate(test\_cases, 1):

        print(f"Test case {i}: {case} -> {merge\_sort(case)}")

**OUTPUT:**

Test case 1: [] -> []

Test case 2: [1] -> [1]

Test case 3: [5, 2, 9, 1, 5, 6] -> [1, 2, 5, 5, 6, 9]

Test case 4: [3, 3, 3] -> [3, 3, 3]

Test case 5: [10, -1, 2, 8, 0, 5] -> [-1, 0, 2, 5, 8, 10]

Test case 6: [1, 2, 3, 4, 5] -> [1, 2, 3, 4, 5]

Test case 7: [5, 4, 3, 2, 1] -> [1, 2, 3, 4, 5]

**OBSERVATION:**

The implementation correctly follows the divide-and-conquer strategy.

It handles edge cases like empty list, single element, duplicates, and reverseorder.

The time complexity is O(n log n), which is optimal for comparison sorting.

Space usage increases due to slicing, making it closer to O(n log n) instead of strict O(n).

The algorithm is stable, as equal elements keep their relative order.

Code is clean, modular, and easy to understand with proper test cases.

**Task Description #2 (Searching – Binary Search with AI  
Optimization)**

**PROMPT:**

Write a Python function binary\_search(arr, target) that returns the index of target in a sorted list or -1 if not found. Add docstring with best, average, and worst-case complexities. Test with different inputs**.**

**GENERATED CODE:**

def binary\_search(arr, target):

    """

    Performs binary search on a sorted list to find the index of the target value.

    Best-case time complexity: O(1)   (target is at the middle)

    Average-case time complexity: O(log n)

    Worst-case time complexity: O(log n)

    Args:

        arr (list): A sorted list of elements.

        target: The value to search for.

    Returns:

        int: The index of target in arr 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

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

    test\_cases = [

        ([1, 2, 3, 4, 5], 3),      # target in middle

        ([1, 2, 3, 4, 5], 1),      # target at start

        ([1, 2, 3, 4, 5], 5),      # target at end

        ([1, 2, 3, 4, 5], 6),      # target not in list

        ([], 1),                   # empty list

        ([10, 20, 30, 40, 50], 40) # target present

    ]

    for arr, target in test\_cases:

        result = binary\_search(arr, target)

        print(f"Searching for {target} in {arr}: Index {result}")

**OUTPUT:**

Searching for 3 in [1, 2, 3, 4, 5]: Index 2

Searching for 1 in [1, 2, 3, 4, 5]: Index 0

Searching for 5 in [1, 2, 3, 4, 5]: Index 4

Searching for 6 in [1, 2, 3, 4, 5]: Index -1

Searching for 1 in []: Index -1

Searching for 40 in [10, 20, 30, 40, 50]: Index 3

**OBSERVATION:**

The given code correctly implements the binary search algorithm on a sorted list. It works by repeatedly dividing the search interval in half, making the process efficient. The time complexity is **O(log n)** in average and worst cases, and **O(1)** in the best case when the element is found at the middle. The space complexity is **O(1)** as it only uses a few variables. It handles important edge cases such as empty lists, target at thebeginning, middle, end, and target not found. The code is clean, easy to read, and well-tested with multiple cases.

**Task Description #3 (Real-Time Application – Inventory  
Management System)**

**PROMPT:**

For a retail store’s inventory system with thousands of products, recommend efficient search and sort algorithms. Implement searching (by product ID/name) and sorting (by price/quantity). Justify choices in a table and implement functions in Python

**GENERATED CODE:**

from typing import List, Dict, Any

# Product structure: {'id': int, 'name': str, 'price': float, 'quantity': int}

def binary\_search\_by\_id(products: List[Dict[str, Any]], target\_id: int) -> Dict[str, Any]:

    """

    Performs binary search on a sorted list of products by 'id'.

    Returns the product dict if found, else None.

    """

    left, right = 0, len(products) - 1

    while left <= right:

        mid = (left + right) // 2

        if products[mid]['id'] == target\_id:

            return products[mid]

        elif products[mid]['id'] < target\_id:

            left = mid + 1

        else:

            right = mid - 1

    return None

def search\_by\_name(products: List[Dict[str, Any]], name: str) -> Dict[str, Any]:

    """

    Searches for a product by name using a hash table (dict).

    Returns the product dict if found, else None.

    """

    name\_map = {p['name']: p for p in products}

    return name\_map.get(name)

def sort\_by\_price(products: List[Dict[str, Any]]) -> List[Dict[str, Any]]:

    """

    Returns a new list of products sorted by price (ascending).

    """

    return sorted(products, key=lambda x: x['price'])

def sort\_by\_quantity(products: List[Dict[str, Any]]) -> List[Dict[str, Any]]:

    """

    Returns a new list of products sorted by quantity (ascending).

    """

    return sorted(products, key=lambda x: x['quantity'])

# Sample data and tests

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

    inventory = [

        {'id': 101, 'name': 'Apple', 'price': 0.99, 'quantity': 50},

        {'id': 102, 'name': 'Banana', 'price': 0.59, 'quantity': 120},

        {'id': 103, 'name': 'Carrot', 'price': 0.29, 'quantity': 200},

        {'id': 104, 'name': 'Dates', 'price': 2.99, 'quantity': 30},

    ]

    # Ensure inventory is sorted by id for binary search

    inventory.sort(key=lambda x: x['id'])

    print("Search by ID (102):", binary\_search\_by\_id(inventory, 102))

    print("Search by Name ('Carrot'):", search\_by\_name(inventory, 'Carrot'))

    print("Sort by Price:", sort\_by\_price(inventory))

    print("Sort by Quantity:", sort\_by\_quantity(inventory))

**OUTPUT:**

Search by ID (102): {'id': 102, 'name': 'Banana', 'price': 0.59, 'quantity': 120}

Search by Name ('Carrot'): {'id': 103, 'name': 'Carrot', 'price': 0.29, 'quantity': 200}

Sort by Price: [{'id': 103, 'name': 'Carrot', 'price': 0.29, 'quantity': 200}, {'id': 102, 'name': 'Banana', 'price': 0.59, 'quantity': 120}, {'id': 101, 'name': 'Apple', 'price': 0.99, 'quantity': 50}, {'id': 104, 'name': 'Dates', 'price': 2.99, 'quantity': 30}]

Sort by Quantity: [{'id': 104, 'name': 'Dates', 'price': 2.99, 'quantity': 30}, {'id': 101, 'name': 'Apple', 'price': 0.99, 'quantity': 50}, {'id': 102, 'name': 'Banana', 'price': 0.59, 'quantity': 120}, {'id': 103, 'name': 'Carrot', 'price': 0.29, 'quantity': 200}]

**OBSERVATION:**

The code provides an efficient way to manage and search products in an inventory system. It uses **binary search** on a list sorted by product id for fast lookup in **O(log n)** time, and a **hash map** for direct search by name in **O(1)** average time. Sorting functions for price and quantity are implemented using Python’s built-in sorted(), which runs in **O(n log n)**. The code is modular, easy to understand, and handles practical operations like searching and sorting efficiently. The sample data and tests demonstrate functionality clearly, making it a good implementation for a small-scale inventory management system.