

SCHOOL OF COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE		DEPARTMENT OF COMPUTER SCIENCE ENGINEERING	
Program Name: B. Tech		Assignment Type: Lab	
Course Coordinator Name		Venkataramana Veeramsetty	
Instructor(s) Name		Dr. V. Venkataramana (Co-ordinator) Dr. T. Sampath Kumar Dr. Pramoda Patro Dr. Brij Kishor Tiwari Dr.J.Ravichander Dr. Mohammand Ali Shaik Dr. Anirodh Kumar Mr. S.Naresh Kumar Dr. RAJESH VELPULA Mr. Kundhan Kumar Ms. Ch.Rajitha Mr. M Prakash Mr. B.Raju Intern 1 (Dharma teja) Intern 2 (Sai Prasad) Intern 3 (Sowmya) NS_2 (Mounika)	
Course Code	24CS002PC 215	Course Title	AI Assisted Coding
Year/Sem	II/I	Regulation	R24
Date and Day of Assignment	Week6 - Monday	Time(s)	
Duration	2 Hours	Applicable to Batches	
AssignmentNumber: 12.1(Present assignment number)/ 24 (Total number of assignments)			

	Question	Expected Time to complete
1	Lab 12: Algorithms with AI Assistance – Sorting, Searching, and Optimizing Algorithms Lab Objectives: <ul style="list-style-type: none"> • Apply AI-assisted programming to implement and 	Week 6 - Monday

- optimize sorting and searching algorithms.
- Compare different algorithms in terms of efficiency and use cases.
 - Understand how AI tools can suggest optimized code and complexity improvements.

Task Description #1 (Sorting – Merge Sort Implementation)

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

PROMPT:

Generate a Python program that implements the Merge Sort algorithm, and also should create a function `merge_sort(arr)` that sorts a list in ascending order, and include time complexity and space complexity in the function docstring.

CODE:

```

❶ def merge_sort(arr):
    """
    Sorts a list in ascending order using the Merge Sort algorithm.

    Args:
        arr: The list to be sorted.

    Returns:
        A new list containing the sorted elements.

    Time Complexity: O(n log n)
    Space Complexity: O(n)
    """
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left_half = arr[:mid]
    right_half = arr[mid:]

    left_sorted = merge_sort(left_half)
    right_sorted = merge_sort(right_half)

    return merge(left_sorted, right_sorted)

def merge(left, right):
    """
    Merges two sorted lists into a single sorted list.

    Args:
        left: The left sorted list.
        right: The right sorted list.

    Returns:
        A new list containing the merged sorted elements.

    """
    merged_list = []
    left_index = 0
    right_index = 0

    while left_index < len(left) and right_index < len(right):
        if left[left_index] < right[right_index]:
            merged_list.append(left[left_index])
            left_index += 1
        else:
            merged_list.append(right[right_index])
            right_index += 1

    # Append any remaining elements from either list
    merged_list.extend(left[left_index:])
    merged_list.extend(right[right_index:])

    return merged_list

# Example usage:
my_list = [38, 27, 43, 3, 9, 82, 10]
sorted_list = merge_sort(my_list)
print("Original list:", my_list)
print("Sorted list:", sorted_list)

```

Original list: [38, 27, 43, 3, 9, 82, 10]
 Sorted list: [3, 9, 10, 27, 38, 43, 82]

Verify the generated code with test cases :

```

❶ # Test case 1: Empty list
list1 = []
sorted_list1 = merge_sort(list1)
print("Original list:", list1)
print("Sorted list:", sorted_list1)
assert sorted_list1 == [], "Test case 1 failed"

# Test case 2: List with one element
list2 = [5]
sorted_list2 = merge_sort(list2)
print("Original list:", list2)
print("Sorted list:", sorted_list2)
assert sorted_list2 == [5], "Test case 2 failed"

# Test case 3: List with duplicate elements
list3 = [1, 5, 2, 5, 3, 1]
sorted_list3 = merge_sort(list3)
print("Original list:", list3)
print("Sorted list:", sorted_list3)
assert sorted_list3 == [1, 1, 2, 3, 5, 5], "Test case 3 failed"

# Test case 4: List with negative numbers
list4 = [-3, 0, -1, 2, 1]
sorted_list4 = merge_sort(list4)
print("Original list:", list4)
print("Sorted list:", sorted_list4)
assert sorted_list4 == [-3, -1, 0, 1, 2], "Test case 4 failed"

# Test case 5: Already sorted list
list5 = [1, 2, 3, 4, 5]
sorted_list5 = merge_sort(list5)
print("Original list:", list5)
print("Sorted list:", sorted_list5)
assert sorted_list5 == [1, 2, 3, 4, 5], "Test case 5 failed"

# Test case 6: Reverse sorted list
list6 = [5, 4, 3, 2, 1]
sorted_list6 = merge_sort(list6)
print("Original list:", list6)
print("Sorted list:", sorted_list6)
assert sorted_list6 == [1, 2, 3, 4, 5], "Test case 6 failed"

print("\nAll test cases passed!")

➡ Original list: []
Sorted list: []
Original list: [5]
Sorted list: [5]
Original list: [1, 5, 2, 5, 3, 1]
Sorted list: [1, 1, 2, 3, 5, 5]
Original list: [-3, 0, -1, 2, 1]
Sorted list: [-3, -1, 0, 1, 2]
Original list: [1, 2, 3, 4, 5]
Sorted list: [1, 2, 3, 4, 5]
Original list: [5, 4, 3, 2, 1]
Sorted list: [1, 2, 3, 4, 5]

All test cases passed!

```

OBSERVATION:

The code successfully implements the Merge Sort algorithm and includes test cases that cover various scenarios, including empty lists, single-element lists, lists with duplicates, negative numbers, and sorted/reverse-sorted lists. All test cases passed, indicating the implementation is correct.

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:

- Prompt AI to create a function `binary_search(arr, target)` returning the index of the target or -1 if not found.
- Include docstrings explaining best, average, and worst-case complexities.
- Test with various inputs.
- Expected Output:
 - Python code implementing binary search with AI-generated comments and docstrings.

PROMPT:

Generate a python code by creating a binary search function that finds a target element in a sorted list, and create a function `binary_search(arr, target)` returning the index of the target or -1 if not found also Include docstrings explaining best, average, and worst-case complexities. Test with various inputs

CODE:

```


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

    Args:
        arr: A sorted list of elements.
        target: The element to search for.

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

    Time Complexity Analysis:
        - Best Case: O(1) - When the target element is the middle element in the first comparison.
        - Average Case: O(log n) - On average, the search space is halved in each step.
        - Worst Case: O(log n) - When the target element is at the beginning or end of the list, or not present.
            The number of comparisons is logarithmic with respect to the size of the list (n).
    """
    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
sorted_list = [2, 5, 8, 12, 16, 23, 38, 56, 72, 91]
target1 = 23
target2 = 5
target3 = 100
target4 = 91

print(f"Index of {target1}: {binary_search(sorted_list, target1)}")
print(f"Index of {target2}: {binary_search(sorted_list, target2)}")
print(f"Index of {target3}: {binary_search(sorted_list, target3)}")
print(f"Index of {target4}: {binary_search(sorted_list, target4)}")


```

Index of 23: 5
Index of 5: 1
Index of 100: -1
Index of 91: 9

OBSERVATION:

This code implements a binary search algorithm, which efficiently finds an element in a *sorted* list by repeatedly dividing the search interval in half. Its key advantage is a logarithmic time complexity ($O(\log n)$), making it very fast for large datasets compared to linear search ($O(n)$).

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:

- Use AI to suggest the most efficient search and sort algorithms for this use case.
- Implement the recommended algorithms in Python.
- Justify the choice based on dataset size, update frequency, and performance requirements.
- Expected Output:
 - A table mapping operation → recommended algorithm → justification.
 - Working 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: Quickly search for a product by ID or name. Sort products by price or quantity for stock analysis. now suggest the most efficient search and sort algorithms for this use case. Implement the recommended algorithms in Python. Justify the choice based on dataset size, update frequency, and performance requirements. The output should be in the form of table mapping operation → recommended algorithm → justification.

CODE:

▶ # 1. Create a dictionary named `inventory` with sample data.

```
inventory = {
    'PROD001': {'name': 'Laptop', 'price': 1200.00, 'quantity': 50},
    'PROD002': {'name': 'Mouse', 'price': 25.00, 'quantity': 200},
    'PROD003': {'name': 'Keyboard', 'price': 75.00, 'quantity': 150},
    'PROD004': {'name': 'Monitor', 'price': 300.00, 'quantity': 30},
    'PROD005': {'name': 'Webcam', 'price': 50.00, 'quantity': 100},
    'PROD006': {'name': 'Printer', 'price': 200.00, 'quantity': 20},
    'PROD007': {'name': 'Speakers', 'price': 100.00, 'quantity': 75},
    'PROD008': {'name': 'Desk Lamp', 'price': 35.00, 'quantity': 120}
}
```

2. Write a Python function named `search_product_by_id`.

```
def search_product_by_id(inventory_dict, product_id):
    """Searches for a product by its ID using direct dictionary lookup."""
    return inventory_dict.get(product_id)
```

3. Write a Python function named `search_product_by_name`.

```
def search_product_by_name(inventory_dict, product_name):
    """Searches for a product by its name using linear search through values."""
    for product_details in inventory_dict.values():
        if product_details['name'].lower() == product_name.lower():
            return product_details
    return None
```

4. Write a Python function named `sort_inventory_by_price`.

```
def sort_inventory_by_price(inventory_dict):
    """Sorts the inventory by product price."""
    # Extract product details (values) into a list and sort
    sorted_products = sorted(inventory_dict.values(), key=lambda x: x['price'])
    return sorted_products
```

5. Write a Python function named `sort_inventory_by_quantity`.

```
def sort_inventory_by_quantity(inventory_dict):
    """Sorts the inventory by product quantity."""
    # Extract product details (values) into a list and sort
    sorted_products = sorted(inventory_dict.values(), key=lambda x: x['quantity'])
    return sorted_products
```

```

# 6. Include example calls to each function and print outputs.

print("--- Search by ID ---")
# Test case: ID found
product_id_found = 'PROD003'
found_product_id = search_product_by_id(inventory, product_id_found)
print(f"Searching for ID '{product_id_found}': {found_product_id}")

# Test case: ID not found
product_id_not_found = 'PROD999'
not_found_product_id = search_product_by_id(inventory, product_id_not_found)
print(f"Searching for ID '{product_id_not_found}': {not_found_product_id}")

print("\n--- Search by Name ---")
# Test case: Name found
product_name_found = 'Monitor'
found_product_name = search_product_by_name(inventory, product_name_found)
print(f"Searching for name '{product_name_found}': {found_product_name}")

# Test case: Name not found
product_name_not_found = 'Tablet'
not_found_product_name = search_product_by_name(inventory, product_name_not_found)
print(f"Searching for name '{product_name_not_found}': {not_found_product_name}")

print("\n--- Sort by Price ---")
sorted_by_price = sort_inventory_by_price(inventory)
print("Inventory sorted by price:")
for product in sorted_by_price:
    print(product)

print("\n--- Sort by Quantity ---")
sorted_by_quantity = sort_inventory_by_quantity(inventory)
print("Inventory sorted by quantity:")
for product in sorted_by_quantity:
    print(product)

```

→ --- Search by ID ---
 Searching for ID 'PROD003': {'name': 'Keyboard', 'price': 75.0, 'quantity': 150}
 Searching for ID 'PROD999': None

--- Search by Name ---
 Searching for name 'Monitor': {'name': 'Monitor', 'price': 300.0, 'quantity': 30}
 Searching for name 'Tablet': None

--- Sort by Price ---
 Inventory sorted by price:
 {'name': 'Mouse', 'price': 25.0, 'quantity': 200}
 {'name': 'Desk Lamp', 'price': 35.0, 'quantity': 120}
 {'name': 'Webcam', 'price': 50.0, 'quantity': 100}
 {'name': 'Keyboard', 'price': 75.0, 'quantity': 150}
 {'name': 'Speakers', 'price': 100.0, 'quantity': 75}
 {'name': 'Printer', 'price': 200.0, 'quantity': 20}
 {'name': 'Monitor', 'price': 300.0, 'quantity': 30}
 {'name': 'Laptop', 'price': 1200.0, 'quantity': 50}

--- Sort by Quantity ---
 Inventory sorted by quantity:
 {'name': 'Printer', 'price': 200.0, 'quantity': 20}
 {'name': 'Monitor', 'price': 300.0, 'quantity': 30}
 {'name': 'Laptop', 'price': 1200.0, 'quantity': 50}
 {'name': 'Speakers', 'price': 100.0, 'quantity': 75}
 {'name': 'Webcam', 'price': 50.0, 'quantity': 100}
 {'name': 'Desk Lamp', 'price': 35.0, 'quantity': 120}
 {'name': 'Keyboard', 'price': 75.0, 'quantity': 150}
 {'name': 'Mouse', 'price': 25.0, 'quantity': 200}

OBSERVATION:

Based on the analysis and implementation, the key observation is that using a dictionary with product ID as the key provides highly efficient $O(1)$ average time complexity for searching by ID, which is crucial for frequent lookups in a retail inventory system. Sorting operations, while less frequent, are efficiently handled by Python's built-in Timsort with $O(n \log n)$ complexity. Searching by name is less efficient ($O(n)$), highlighting the importance of ID-based lookups for performance.

Deliverables (For All Tasks)

1. AI-generated prompts for code and test case generation.
2. At least 3 assert test cases for each task.
3. AI-generated initial code and execution screenshots.
4. Analysis of whether code passes all tests.
5. Improved final version with inline comments and explanation.
6. Compiled report (Word/PDF) with prompts, test cases, assertions, code, and output.

Top of Form