SCHOOL OF COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE		**DEPARTMENT OF COMPUTER SCIENCE ENGINEERING**	
Program Name: B. Tech	**Assignment Type: Lab**	**Academic Year:**2025-2026	
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	24CS002PC215	**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)

Q.No.	**Question**	***Expected Time to complete***
1	**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.

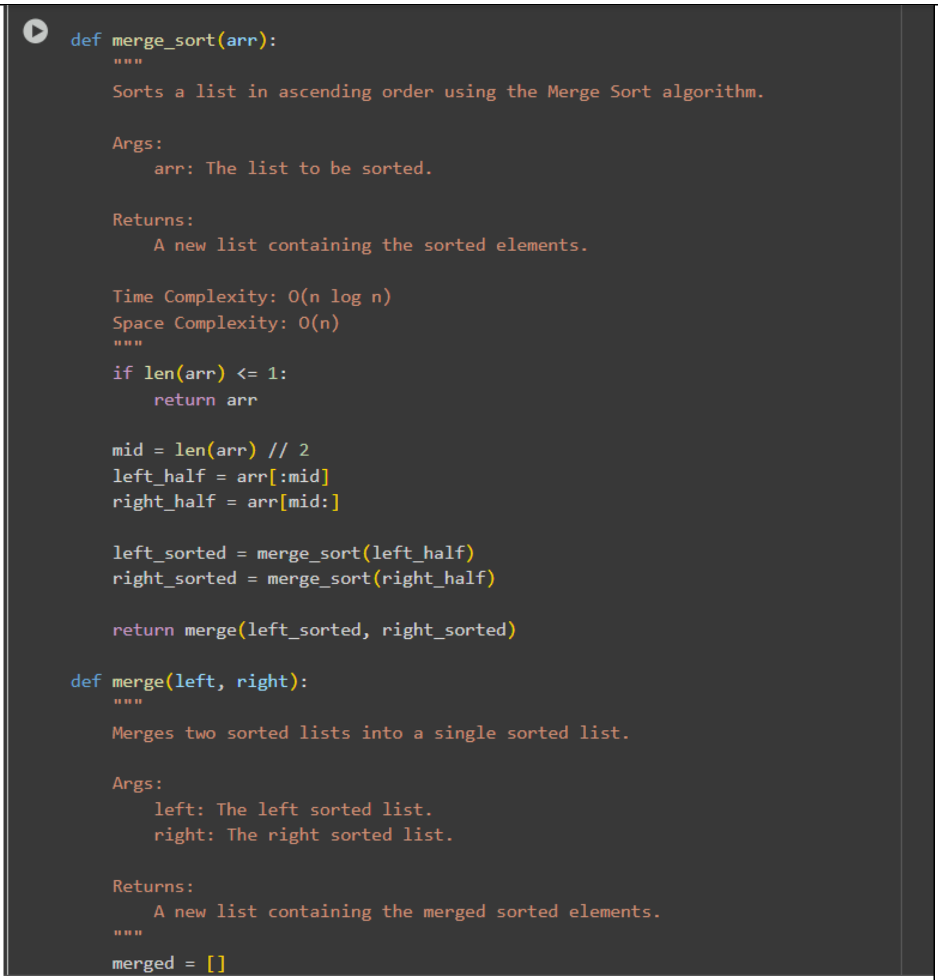
Task Description #1 (Sorting – Merge Sort Implementation)

- 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.

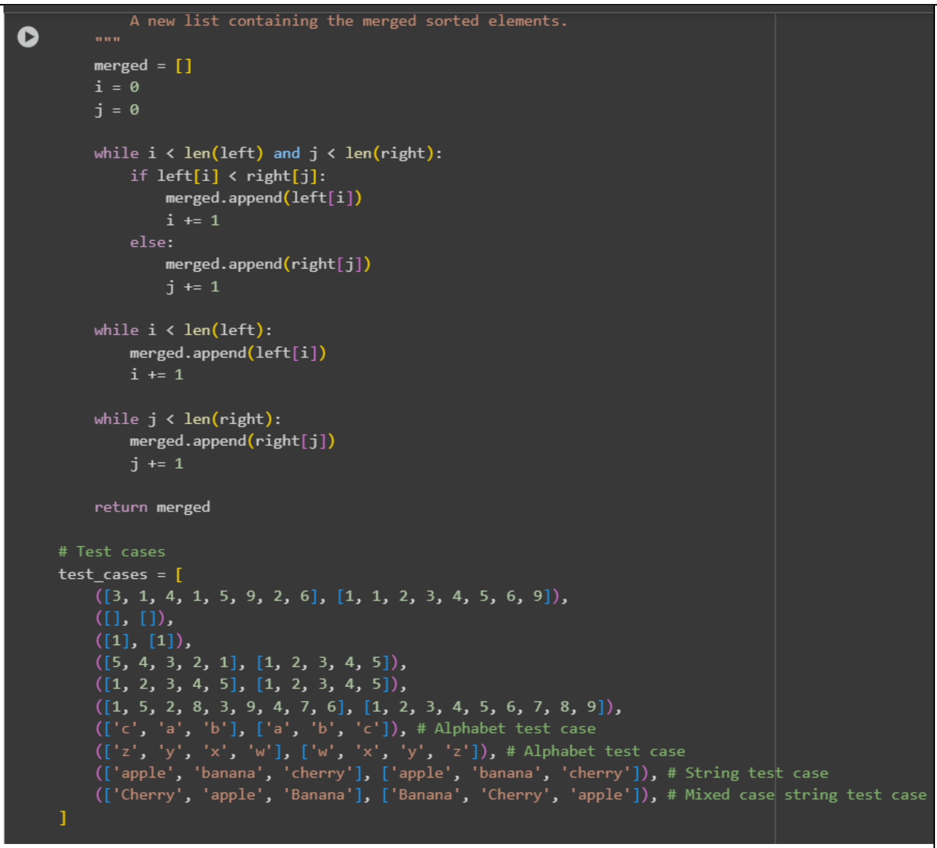
ANSWER:

PROMPT:Write a python program that implements the merge sort algorithms.Create a function merge_sort(arr) that sorts a list in ascending order. Include a doctstring with a time complexity and space complexity.also,add test cases to verify the code.

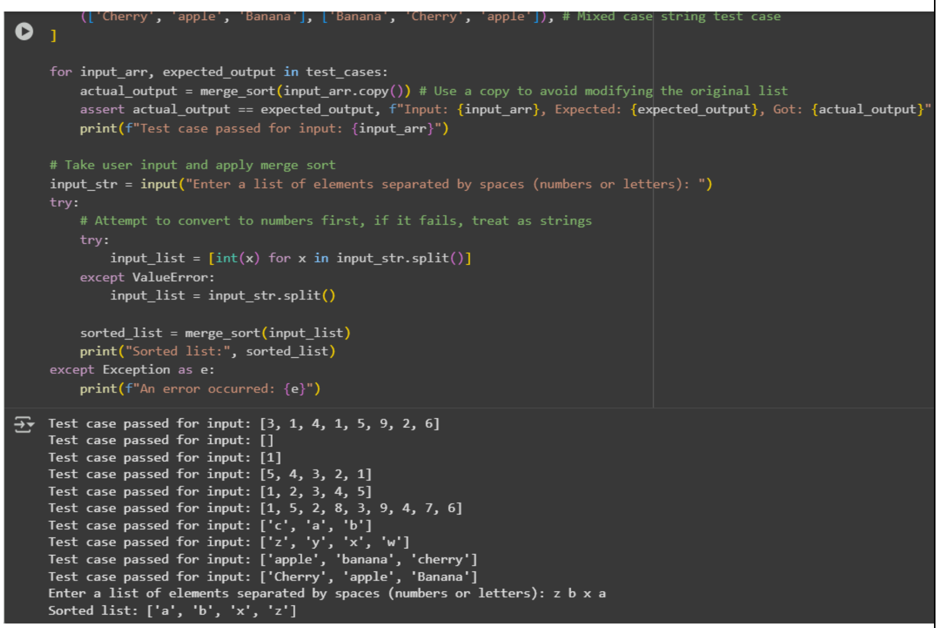
SCREENSHOTS:



```python
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 = []
```



```
        A new list containing the merged sorted elements.
    """
    merged = []
    i = 0
    j = 0

    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

    while i < len(left):
        merged.append(left[i])
        i += 1

    while j < len(right):
        merged.append(right[j])
        j += 1

    return merged

# Test cases
test_cases = [
    ([3, 1, 4, 1, 5, 9, 2, 6], [1, 1, 2, 3, 4, 5, 6, 9]),
    ([], []),
    ([1], [1]),
    ([5, 4, 3, 2, 1], [1, 2, 3, 4, 5]),
    ([1, 2, 3, 4, 5], [1, 2, 3, 4, 5]),
    ([1, 5, 2, 8, 3, 9, 4, 7, 6], [1, 2, 3, 4, 5, 6, 7, 8, 9]),
    (['c', 'a', 'b'], ['a', 'b', 'c']), # Alphabet test case
    (['z', 'y', 'x', 'w'], ['w', 'x', 'y', 'z']), # Alphabet test case
    (['apple', 'banana', 'cherry'], ['apple', 'banana', 'cherry']), # String test case
    (['Cherry', 'apple', 'Banana'], ['Banana', 'Cherry', 'apple']), # Mixed case string test case
]
```



```
    (['Cherry', 'apple', 'Banana'], ['Banana', 'Cherry', 'apple']), # Mixed case string test case
]

for input_arr, expected_output in test_cases:
    actual_output = merge_sort(input_arr.copy()) # Use a copy to avoid modifying the original list
    assert actual_output == expected_output, f"Input: {input_arr}, Expected: {expected_output}, Got: {actual_output}"
    print(f"Test case passed for input: {input_arr}")

# Take user input and apply merge sort
input_str = input("Enter a list of elements separated by spaces (numbers or letters): ")
try:
    # Attempt to convert to numbers first, if it fails, treat as strings
    try:
        input_list = [int(x) for x in input_str.split()]
    except ValueError:
        input_list = input_str.split()

    sorted_list = merge_sort(input_list)
    print("Sorted list:", sorted_list)
except Exception as e:
    print(f"An error occurred: {e}")
```

```
Test case passed for input: [3, 1, 4, 1, 5, 9, 2, 6]
Test case passed for input: []
Test case passed for input: [1]
Test case passed for input: [5, 4, 3, 2, 1]
Test case passed for input: [1, 2, 3, 4, 5]
Test case passed for input: [1, 5, 2, 8, 3, 9, 4, 7, 6]
Test case passed for input: ['c', 'a', 'b']
Test case passed for input: ['z', 'y', 'x', 'w']
Test case passed for input: ['apple', 'banana', 'cherry']
Test case passed for input: ['Cherry', 'apple', 'Banana']
Enter a list of elements separated by spaces (numbers or letters): z b x a
Sorted list: ['a', 'b', 'x', 'z']
```

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

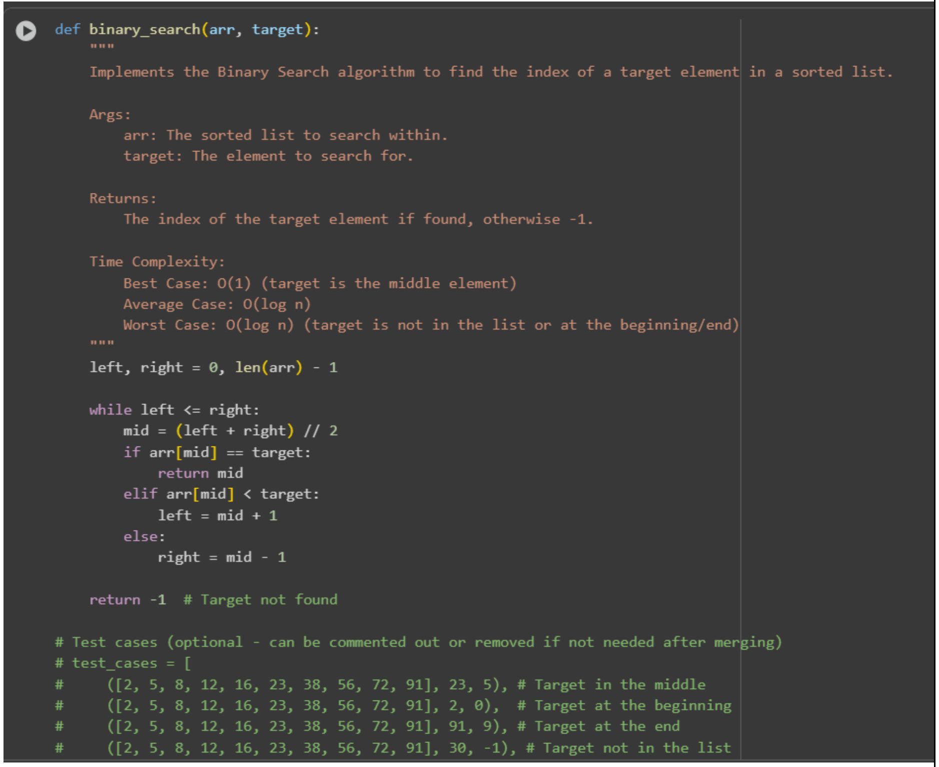
- Task: Use AI to create a binary search function that finds a target element in a sorted list.
- Instructions:

  - 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

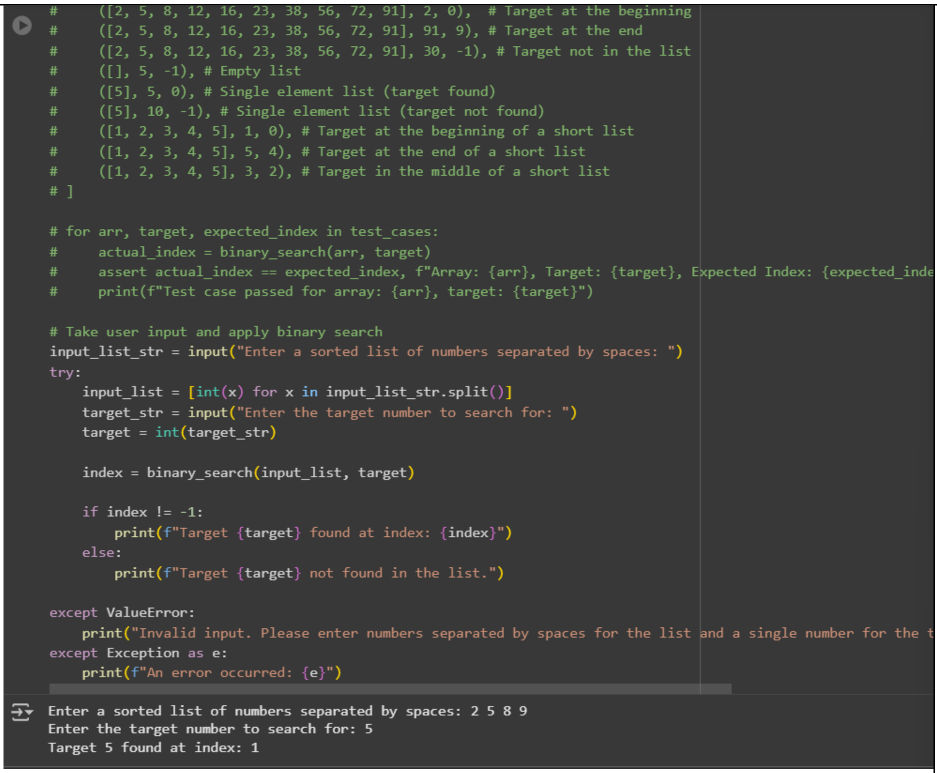
ANSWER:

PROMPT:Write a python program that implements binary search . create a function binary_search(arr,target) that returns the index of the target element in a sorted list,or -1 if not found. Include a docstring with best,average and worst-case time complexities.

SCREENSHOTS:



```
def binary_search(arr, target):
    """
    Implements the Binary Search algorithm to find the index of a target element in a sorted list.

    Args:
        arr: The sorted list to search within.
        target: The element to search for.

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

    Time Complexity:
        Best Case: O(1) (target is the middle element)
        Average Case: O(log n)
        Worst Case: O(log n) (target is not in the list or at the beginning/end)
    """
    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  # Target not found

# Test cases (optional - can be commented out or removed if not needed after merging)
# test_cases = [
#     ([2, 5, 8, 12, 16, 23, 38, 56, 72, 91], 23, 5), # Target in the middle
#     ([2, 5, 8, 12, 16, 23, 38, 56, 72, 91], 2, 0),  # Target at the beginning
#     ([2, 5, 8, 12, 16, 23, 38, 56, 72, 91], 91, 9), # Target at the end
#     ([2, 5, 8, 12, 16, 23, 38, 56, 72, 91], 30, -1), # Target not in the list
```



```
#     ([2, 5, 8, 12, 16, 23, 38, 56, 72, 91], 2, 0),  # Target at the beginning
#     ([2, 5, 8, 12, 16, 23, 38, 56, 72, 91], 91, 9), # Target at the end
#     ([2, 5, 8, 12, 16, 23, 38, 56, 72, 91], 30, -1), # Target not in the list
#     ([], 5, -1), # Empty list
#     ([5], 5, 0), # Single element list (target found)
#     ([5], 10, -1), # Single element list (target not found)
#     ([1, 2, 3, 4, 5], 1, 0), # Target at the beginning of a short list
#     ([1, 2, 3, 4, 5], 5, 4), # Target at the end of a short list
#     ([1, 2, 3, 4, 5], 3, 2), # Target in the middle of a short list
# ]

# for arr, target, expected_index in test_cases:
#     actual_index = binary_search(arr, target)
#     assert actual_index == expected_index, f"Array: {arr}, Target: {target}, Expected Index: {expected_inde
#     print(f"Test case passed for array: {arr}, target: {target}")

# Take user input and apply binary search
input_list_str = input("Enter a sorted list of numbers separated by spaces: ")
try:
    input_list = [int(x) for x in input_list_str.split()]
    target_str = input("Enter the target number to search for: ")
    target = int(target_str)

    index = binary_search(input_list, target)

    if index != -1:
        print(f"Target {target} found at index: {index}")
    else:
        print(f"Target {target} not found in the list.")

except ValueError:
    print("Invalid input. Please enter numbers separated by spaces for the list and a single number for the t
except Exception as e:
    print(f"An error occurred: {e}")
```

```
Enter a sorted list of numbers separated by spaces: 2 5 8 9
Enter the target number to search for: 5
Target 5 found at index: 1
```

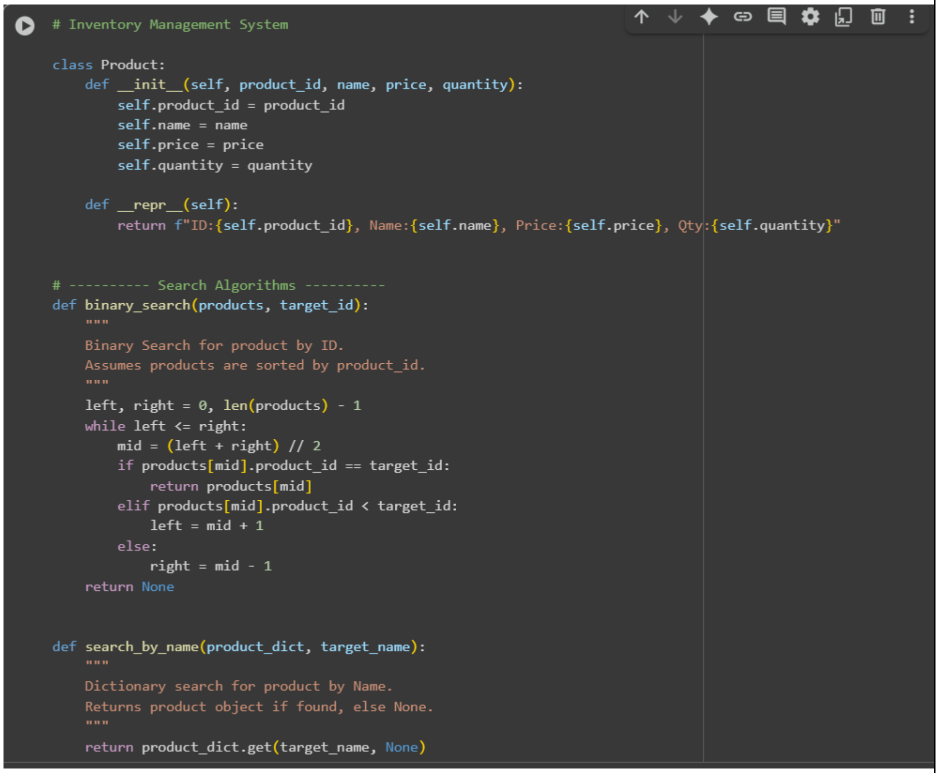
Task Description #3 (Real-Time Application – Inventory Management System)

- 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.

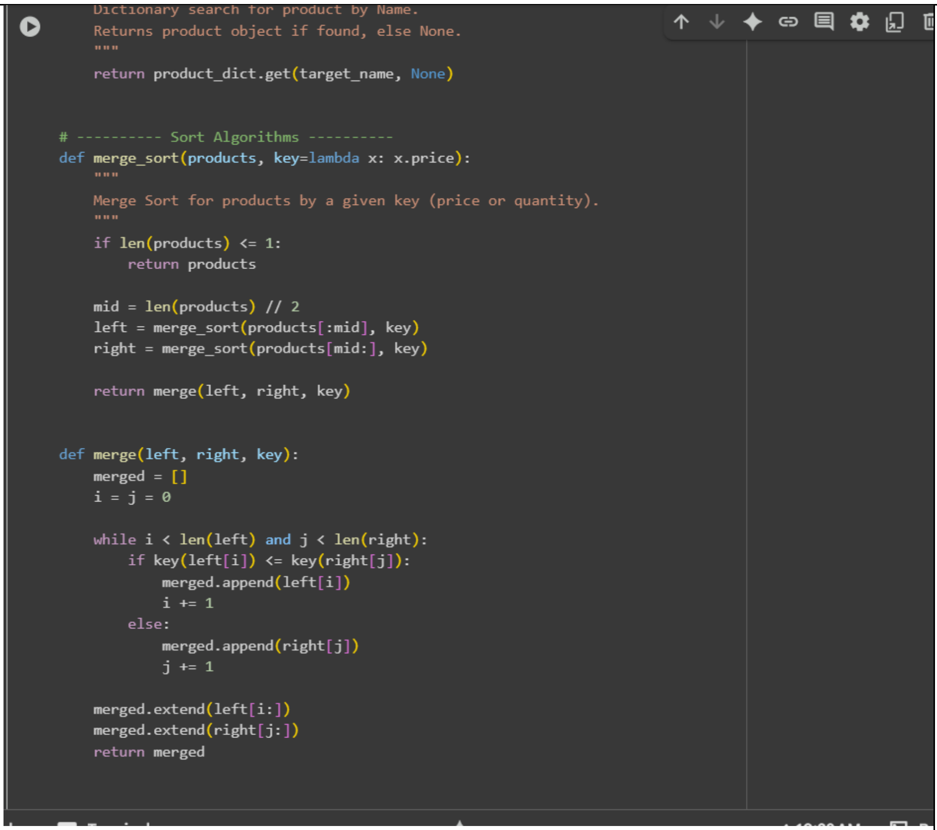
ANSWERS:

PROMPT: Write a python program for an inventory management system.Each product has an id,name,price and quantity.use binary search to find a product by ID,use a dictionary to search by name, and use merge sort to sort products by price or quantity.include a table that maps operations to recommended algortihms with justifications.Add test cases to show searching and sorting results.

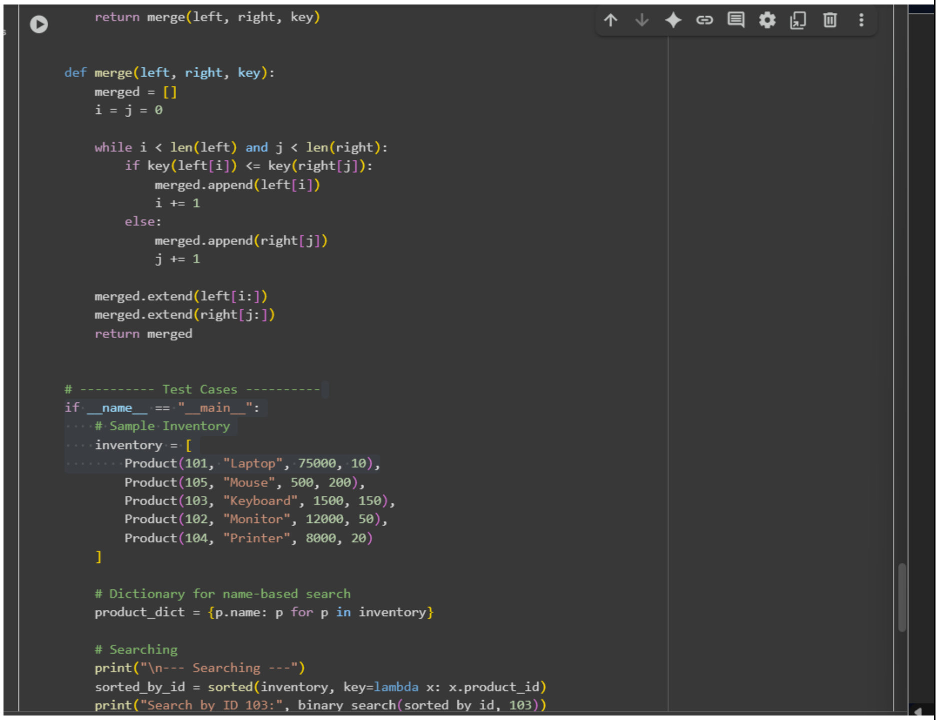
SCREENSHOTS:



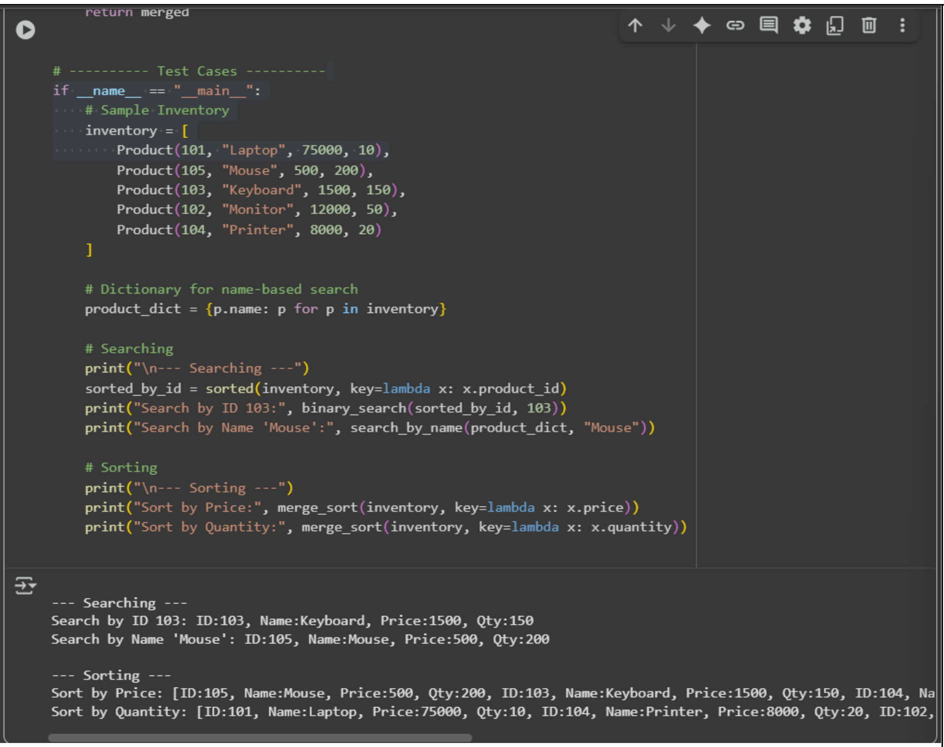
```
# Inventory Management System

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"ID:{self.product_id}, Name:{self.name}, Price:{self.price}, Qty:{self.quantity}"


# ---------- Search Algorithms ----------
def binary_search(products, target_id):
    """
    Binary Search for product by ID.
    Assumes products are sorted by product_id.
    """
    left, right = 0, len(products) - 1
    while left <= right:
        mid = (left + right) // 2
        if products[mid].product_id == target_id:
            return products[mid]
        elif products[mid].product_id < target_id:
            left = mid + 1
        else:
            right = mid - 1
    return None


def search_by_name(product_dict, target_name):
    """
    Dictionary search for product by Name.
    Returns product object if found, else None.
    """
    return product_dict.get(target_name, None)
```



```
    Dictionary search for product by Name.
    Returns product object if found, else None.
    """
    return product_dict.get(target_name, None)


# ---------- Sort Algorithms ----------
def merge_sort(products, key=lambda x: x.price):
    """
    Merge Sort for products by a given key (price or quantity).
    """
    if len(products) <= 1:
        return products

    mid = len(products) // 2
    left = merge_sort(products[:mid], key)
    right = merge_sort(products[mid:], key)

    return merge(left, right, key)


def merge(left, right, key):
    merged = []
    i = j = 0

    while i < len(left) and j < len(right):
        if key(left[i]) <= key(right[j]):
            merged.append(left[i])
            i += 1
        else:
            merged.append(right[j])
            j += 1

    merged.extend(left[i:])
    merged.extend(right[j:])
    return merged
```



```
    return merge(left, right, key)


def merge(left, right, key):
    merged = []
    i = j = 0

    while i < len(left) and j < len(right):
        if key(left[i]) <= key(right[j]):
            merged.append(left[i])
            i += 1
        else:
            merged.append(right[j])
            j += 1

    merged.extend(left[i:])
    merged.extend(right[j:])
    return merged


# ---------- Test Cases ----------
if __name__ == "__main__":
    # Sample Inventory
    inventory = [
        Product(101, "Laptop", 75000, 10),
        Product(105, "Mouse", 500, 200),
        Product(103, "Keyboard", 1500, 150),
        Product(102, "Monitor", 12000, 50),
        Product(104, "Printer", 8000, 20)
    ]

    # Dictionary for name-based search
    product_dict = {p.name: p for p in inventory}

    # Searching
    print("\n--- Searching ---")
    sorted_by_id = sorted(inventory, key=lambda x: x.product_id)
    print("Search by ID 103:", binary search(sorted by id, 103))
```



```
    return merged

# ---------- Test Cases ----------
if __name__ == "__main__":
    # Sample Inventory
    inventory = [
        Product(101, "Laptop", 75000, 10),
        Product(105, "Mouse", 500, 200),
        Product(103, "Keyboard", 1500, 150),
        Product(102, "Monitor", 12000, 50),
        Product(104, "Printer", 8000, 20)
    ]

    # Dictionary for name-based search
    product_dict = {p.name: p for p in inventory}

    # Searching
    print("\n--- Searching ---")
    sorted_by_id = sorted(inventory, key=lambda x: x.product_id)
    print("Search by ID 103:", binary_search(sorted_by_id, 103))
    print("Search by Name 'Mouse':", search_by_name(product_dict, "Mouse"))

    # Sorting
    print("\n--- Sorting ---")
    print("Sort by Price:", merge_sort(inventory, key=lambda x: x.price))
    print("Sort by Quantity:", merge_sort(inventory, key=lambda x: x.quantity))
```

```
--- Searching ---
Search by ID 103: ID:103, Name:Keyboard, Price:1500, Qty:150
Search by Name 'Mouse': ID:105, Name:Mouse, Price:500, Qty:200

--- Sorting ---
Sort by Price: [ID:105, Name:Mouse, Price:500, Qty:200, ID:103, Name:Keyboard, Price:1500, Qty:150, ID:104, Na
Sort by Quantity: [ID:101, Name:Laptop, Price:75000, Qty:10, ID:104, Name:Printer, Price:8000, Qty:20, ID:102,
```

☑ 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.