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.

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

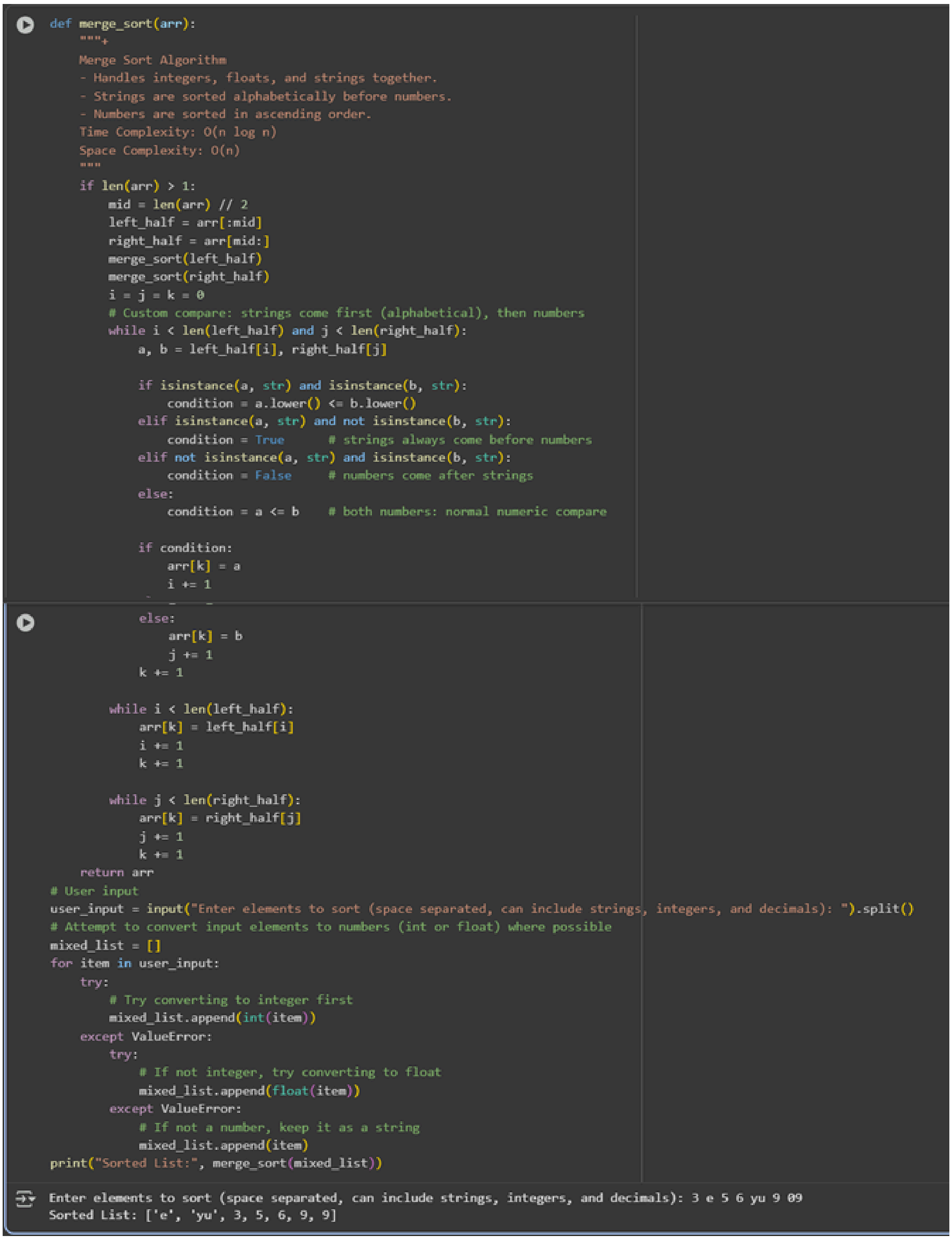
- Task: Use AI to create a binary search function that finds a target element in a sorted list.
- Instructions:
  - Prompt AI to create a function binary_search(arr, target) returning the index of the target or -1 if not found.
  - Include docstrings explaining best, average, and worst-case complexities.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

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

prompts :

create a Python function merge_sort(arr) that sorts a list in ascending order using the Merge Sort algorithm.Include a docstring with time and space complexities and allow the user to input a list of numbers to sort



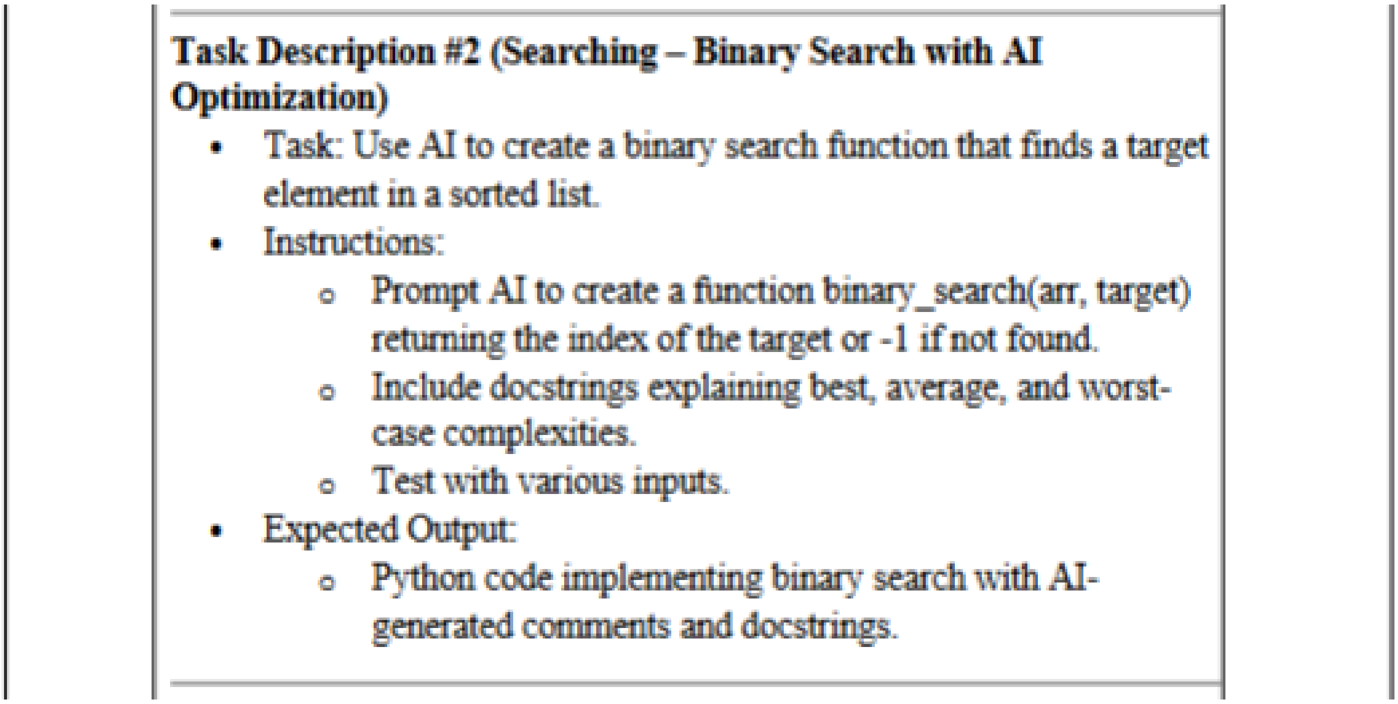
```python
def merge_sort(arr):
    """+
    Merge Sort Algorithm
    - Handles integers, floats, and strings together.
    - Strings are sorted alphabetically before numbers.
    - Numbers are sorted in ascending order.
    Time Complexity: O(n log n)
    Space Complexity: O(n)
    """
    if len(arr) > 1:
        mid = len(arr) // 2
        left_half = arr[:mid]
        right_half = arr[mid:]
        merge_sort(left_half)
        merge_sort(right_half)
        i = j = k = 0
        # Custom compare: strings come first (alphabetical), then numbers
        while i < len(left_half) and j < len(right_half):
            a, b = left_half[i], right_half[j]

            if isinstance(a, str) and isinstance(b, str):
                condition = a.lower() <= b.lower()
            elif isinstance(a, str) and not isinstance(b, str):
                condition = True       # strings always come before numbers
            elif not isinstance(a, str) and isinstance(b, str):
                condition = False      # numbers come after strings
            else:
                condition = a <= b     # both numbers: normal numeric compare

            if condition:
                arr[k] = a
                i += 1
```

```python
            else:
                arr[k] = b
                j += 1
            k += 1

        while i < len(left_half):
            arr[k] = left_half[i]
            i += 1
            k += 1

        while j < len(right_half):
            arr[k] = right_half[j]
            j += 1
            k += 1
    return arr
# User input
user_input = input("Enter elements to sort (space separated, can include strings, integers, and decimals): ").split()
# Attempt to convert input elements to numbers (int or float) where possible
mixed_list = []
for item in user_input:
    try:
        # Try converting to integer first
        mixed_list.append(int(item))
    except ValueError:
        try:
            # If not integer, try converting to float
            mixed_list.append(float(item))
        except ValueError:
            # If not a number, keep it as a string
            mixed_list.append(item)
print("Sorted List:", merge_sort(mixed_list))
```

```
Enter elements to sort (space separated, can include strings, integers, and decimals): 3 e 5 6 yu 9 09
Sorted List: ['e', 'yu', 3, 5, 6, 9, 9]
```

observations and code expalnation

- User enters a list of integers separated by spaces.
- The list is divided recursively into halves.
- Each half is sorted using recursive calls.
- Sorted halves are merged into a final sorted list.
- Temporary arrays cause O(n) space complexity.
- Time complexity remains O(n log n).



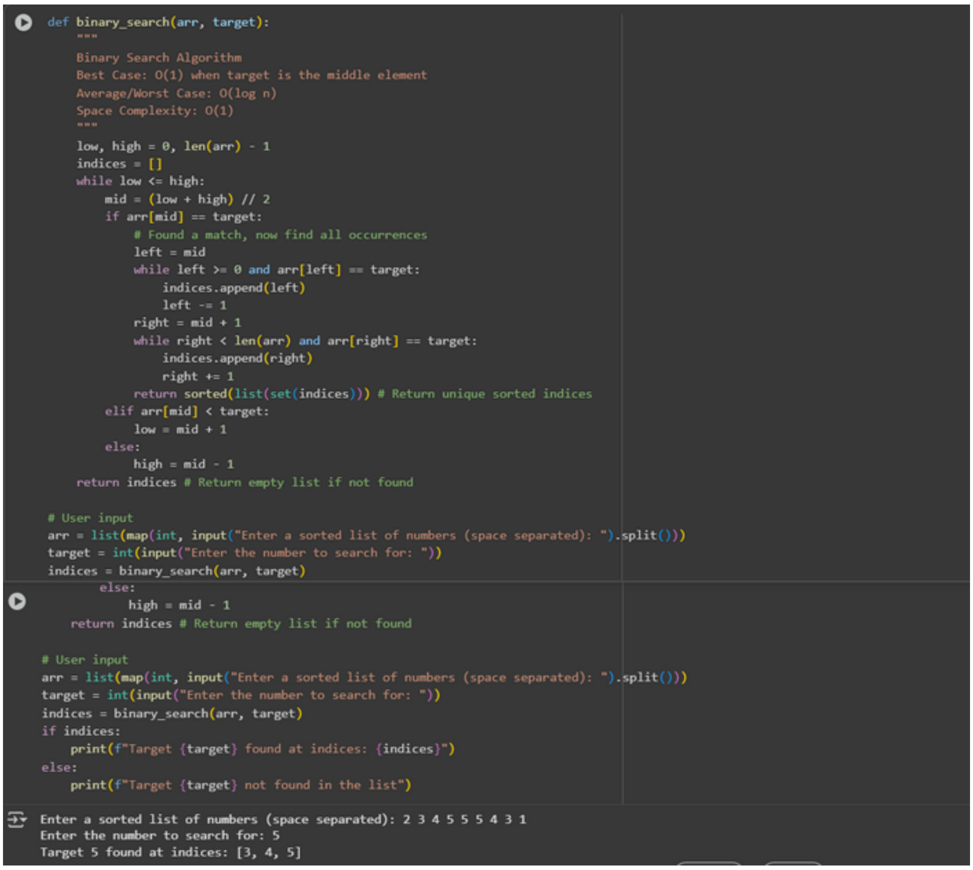
Task Description #2 (Searching – Binary Search with AI Optimization)

- 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.

prompts :

write a function binary_search(arr, target) to find a user-specified target in a sorted list. Include best, average, and worst-case complexities.
Allow the user to input the sorted list and the target value.

code:



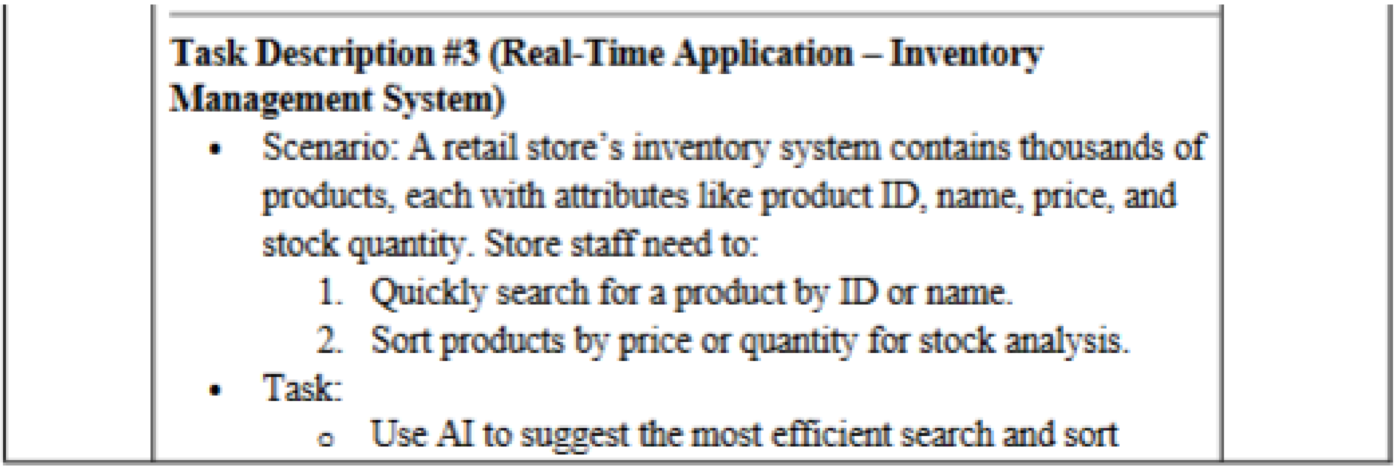
```
def binary_search(arr, target):
    """
    Binary Search Algorithm
    Best Case: O(1) when target is the middle element
    Average/Worst Case: O(log n)
    Space Complexity: O(1)
    """
    low, high = 0, len(arr) - 1
    indices = []
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            # Found a match, now find all occurrences
            left = mid
            while left >= 0 and arr[left] == target:
                indices.append(left)
                left -= 1
            right = mid + 1
            while right < len(arr) and arr[right] == target:
                indices.append(right)
                right += 1
            return sorted(list(set(indices))) # Return unique sorted indices
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return indices # Return empty list if not found

# User input
arr = list(map(int, input("Enter a sorted list of numbers (space separated): ").split()))
target = int(input("Enter the number to search for: "))
indices = binary_search(arr, target)
```

```
        else:
            high = mid - 1
    return indices # Return empty list if not found

# User input
arr = list(map(int, input("Enter a sorted list of numbers (space separated): ").split()))
target = int(input("Enter the number to search for: "))
indices = binary_search(arr, target)
if indices:
    print(f"Target {target} found at indices: {indices}")
else:
    print(f"Target {target} not found in the list")
```

```
Enter a sorted list of numbers (space separated): 2 3 4 5 5 5 4 3 1
Enter the number to search for: 5
Target 5 found at indices: [3, 4, 5]
```

observations and code expalnation :

- User provides a sorted list and a target value.
- The list is divided repeatedly to locate the target.
- Low and high pointers define the search range.
- Complexity is O(log n) for average/worst cases.
- Uses O(1) additional space.
- Returns the index or -1 if not found.



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

Prompts :

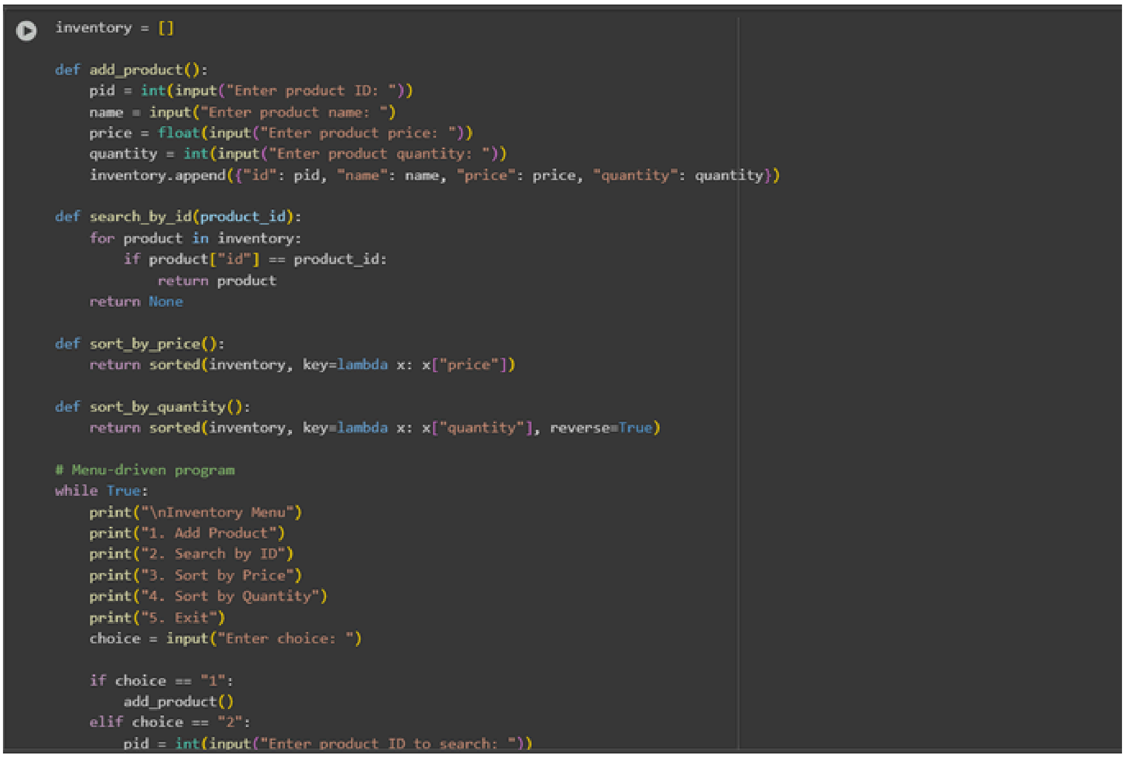
Hello AI, help me create a Python inventory management system.

The user can:

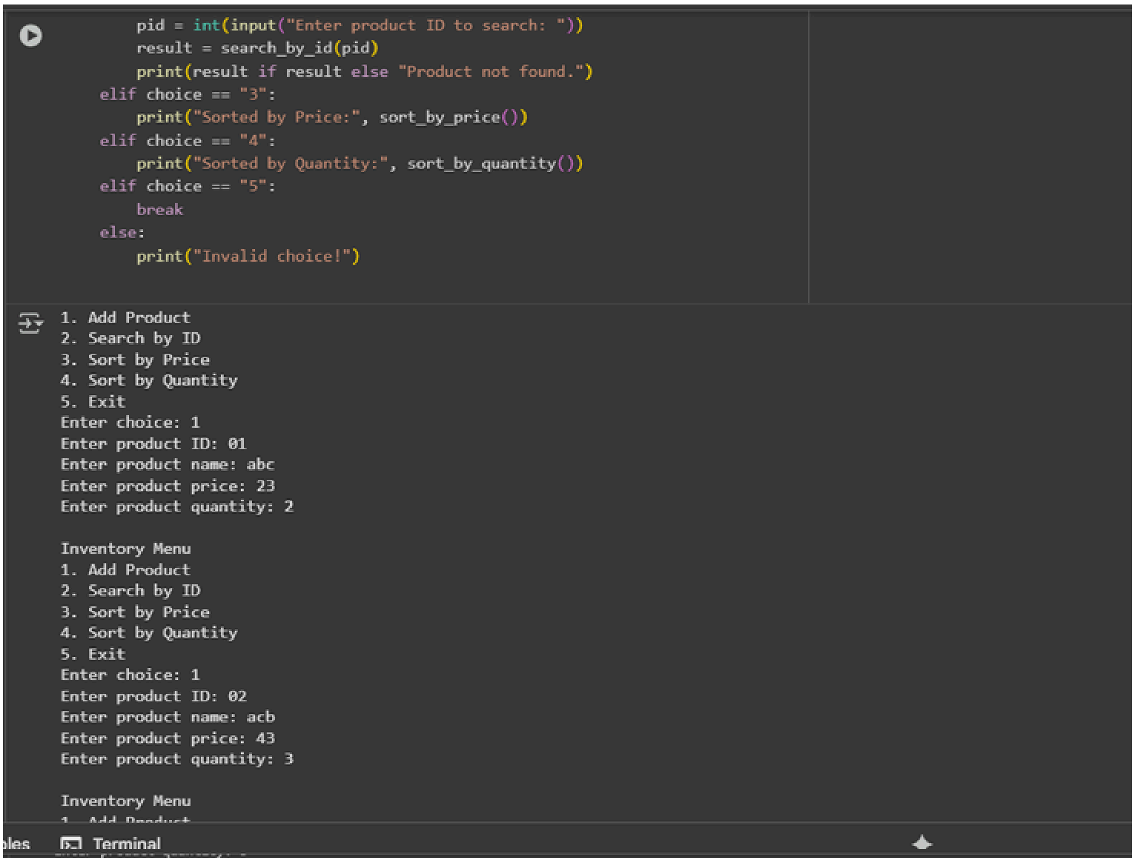
- Add products (ID, name, price, quantity).
- Search for a product by ID.
- Sort the inventory by price or quantity.

Suggest the best search and sort algorithms for large datasets

Code:

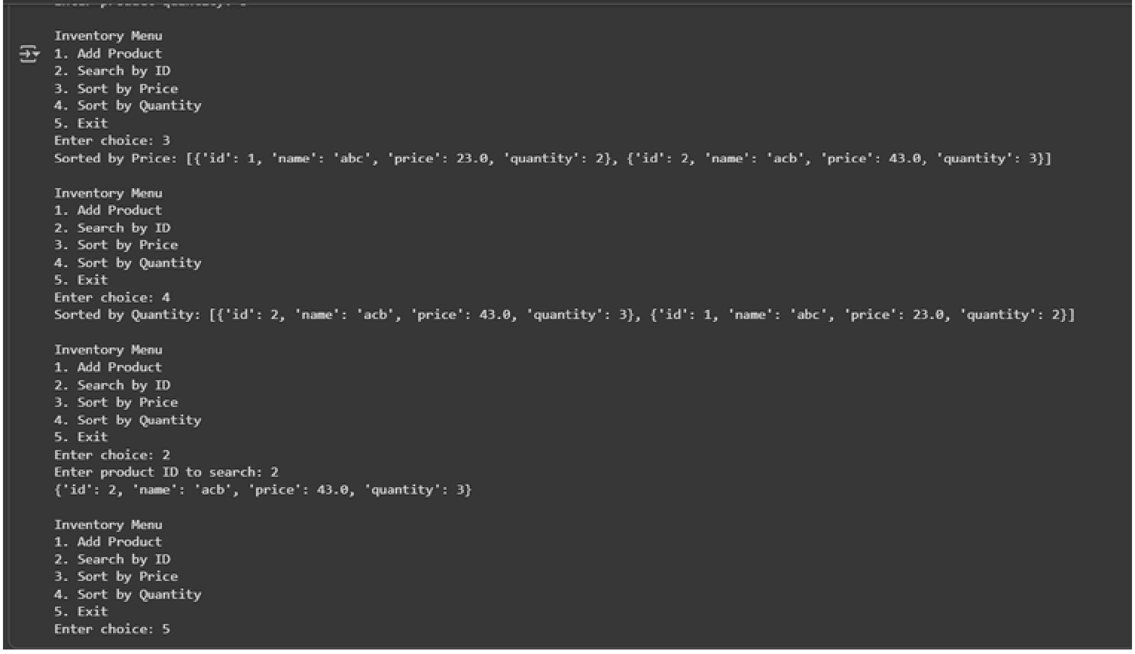


```python
inventory = []

def add_product():
    pid = int(input("Enter product ID: "))
    name = input("Enter product name: ")
    price = float(input("Enter product price: "))
    quantity = int(input("Enter product quantity: "))
    inventory.append({"id": pid, "name": name, "price": price, "quantity": quantity})

def search_by_id(product_id):
    for product in inventory:
        if product["id"] == product_id:
            return product
    return None

def sort_by_price():
    return sorted(inventory, key=lambda x: x["price"])

def sort_by_quantity():
    return sorted(inventory, key=lambda x: x["quantity"], reverse=True)

# Menu-driven program
while True:
    print("\nInventory Menu")
    print("1. Add Product")
    print("2. Search by ID")
    print("3. Sort by Price")
    print("4. Sort by Quantity")
    print("5. Exit")
    choice = input("Enter choice: ")

    if choice == "1":
        add_product()
    elif choice == "2":
        pid = int(input("Enter product ID to search: "))
```



```python
        pid = int(input("Enter product ID to search: "))
        result = search_by_id(pid)
        print(result if result else "Product not found.")
    elif choice == "3":
        print("Sorted by Price:", sort_by_price())
    elif choice == "4":
        print("Sorted by Quantity:", sort_by_quantity())
    elif choice == "5":
        break
    else:
        print("Invalid choice!")
```

```
1. Add Product
2. Search by ID
3. Sort by Price
4. Sort by Quantity
5. Exit
Enter choice: 1
Enter product ID: 01
Enter product name: abc
Enter product price: 23
Enter product quantity: 2

Inventory Menu
1. Add Product
2. Search by ID
3. Sort by Price
4. Sort by Quantity
5. Exit
Enter choice: 1
Enter product ID: 02
Enter product name: acb
Enter product price: 43
Enter product quantity: 3

Inventory Menu
1. Add Product
```



```
Inventory Menu
1. Add Product
2. Search by ID
3. Sort by Price
4. Sort by Quantity
5. Exit
Enter choice: 3
Sorted by Price: [{'id': 1, 'name': 'abc', 'price': 23.0, 'quantity': 2}, {'id': 2, 'name': 'acb', 'price': 43.0, 'quantity': 3}]

Inventory Menu
1. Add Product
2. Search by ID
3. Sort by Price
4. Sort by Quantity
5. Exit
Enter choice: 4
Sorted by Quantity: [{'id': 2, 'name': 'acb', 'price': 43.0, 'quantity': 3}, {'id': 1, 'name': 'abc', 'price': 23.0, 'quantity': 2}]

Inventory Menu
1. Add Product
2. Search by ID
3. Sort by Price
4. Sort by Quantity
5. Exit
Enter choice: 2
Enter product ID to search: 2
{'id': 2, 'name': 'acb', 'price': 43.0, 'quantity': 3}

Inventory Menu
1. Add Product
2. Search by ID
3. Sort by Price
4. Sort by Quantity
5. Exit
Enter choice: 5
```

Code explanation and observations :

- User can add multiple products with details like ID, name, price, and quantity.
- Search by ID uses a simple linear search for clarity (HashMap recommended for large datasets).
- Sorting uses Python's Timsort for efficiency (merge sort hybrid).
- Sorting can be done by price or quantity.
- Menu-driven interface allows repeated operations.
- Scalable to handle larger inventories.