AI ASSISTED CODING

Name:Dugyala Ashmitha

Enrollno:2403a510g5

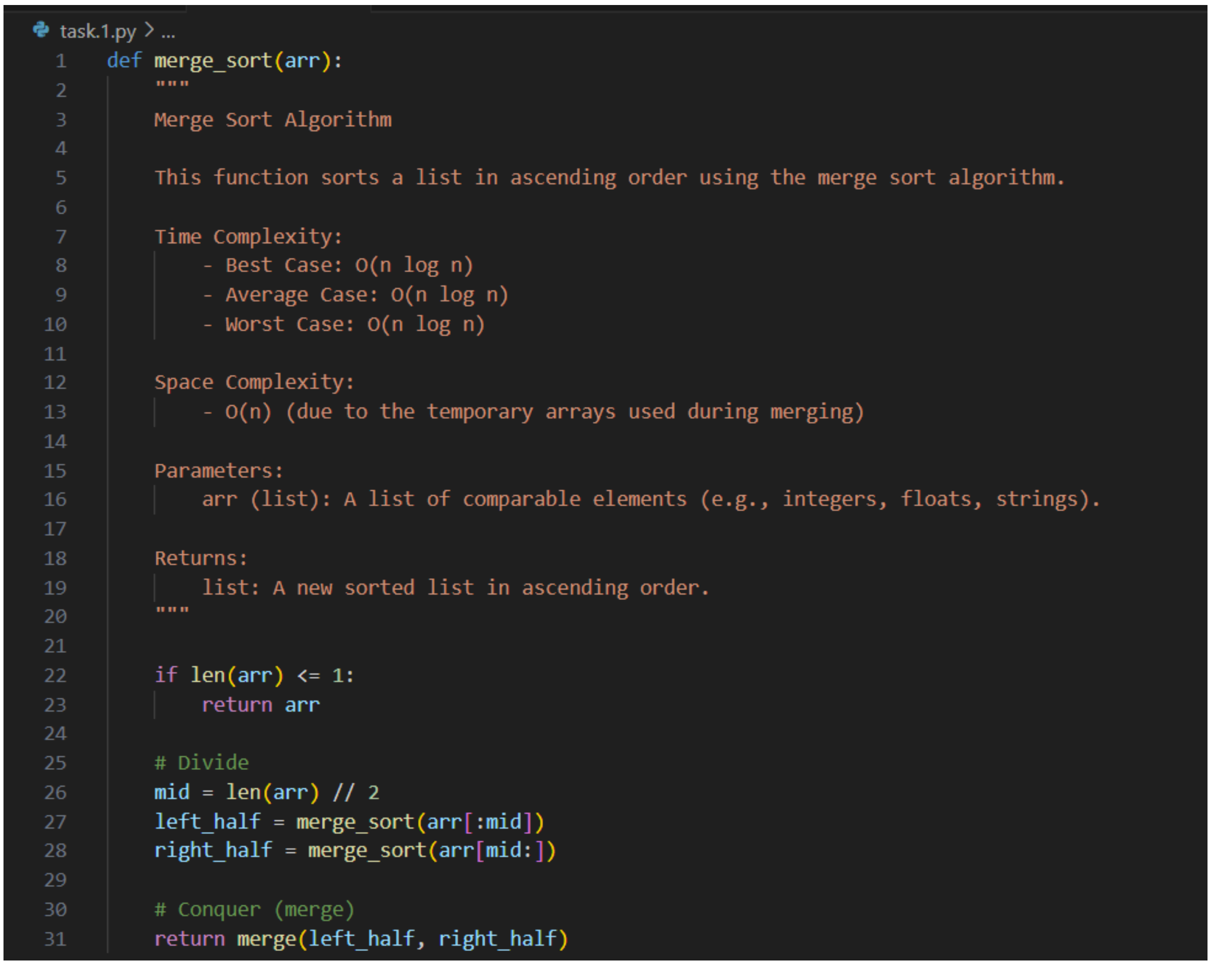
Assignment:12.1

Task1:

Prompt:

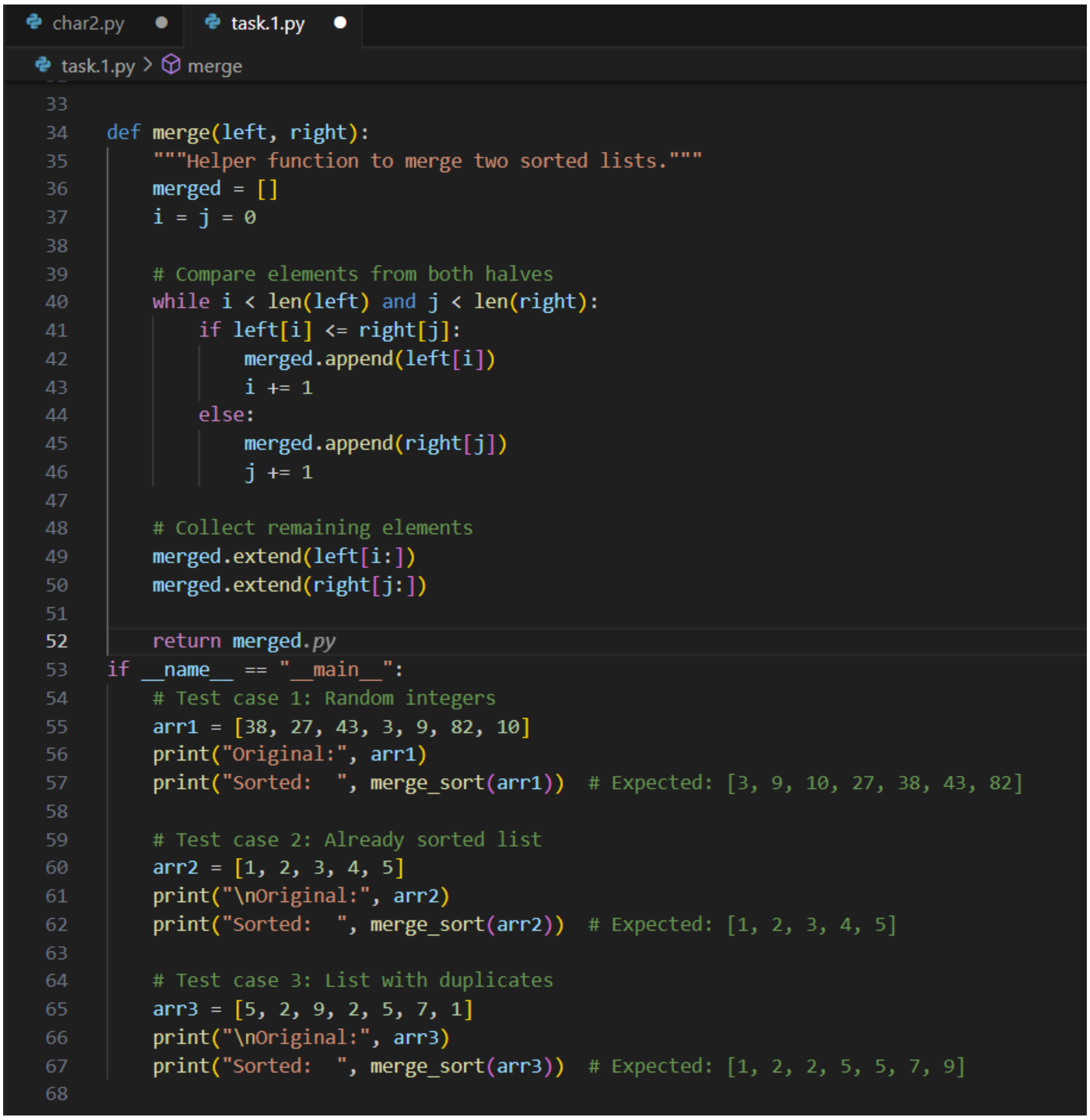
Write a Python program to implement merge sort. Create a function merge_sort(arr) that sorts a list in ascending order. Add a docstring with time complexity and space complexity. Finally, test the function with 3 test cases.

Code:



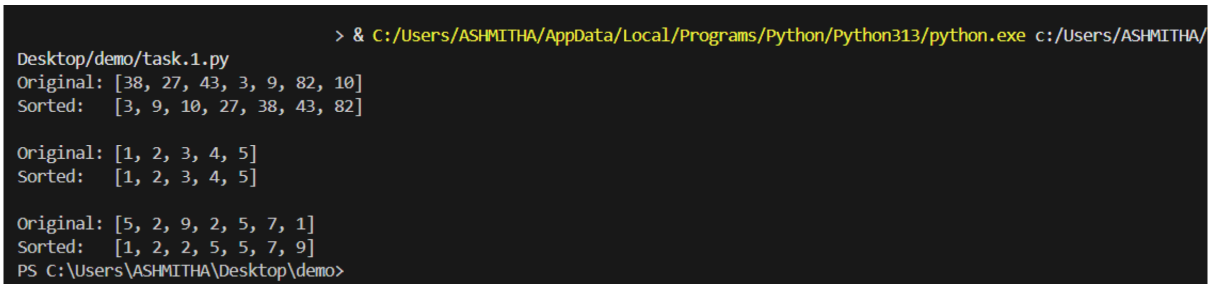
task.1.py > ...

```python
def merge_sort(arr):
    """
    Merge Sort Algorithm

    This function sorts a list in ascending order using the merge sort algorithm.

    Time Complexity:
        - Best Case: O(n log n)
        - Average Case: O(n log n)
        - Worst Case: O(n log n)

    Space Complexity:
        - O(n) (due to the temporary arrays used during merging)

    Parameters:
        arr (list): A list of comparable elements (e.g., integers, floats, strings).

    Returns:
        list: A new sorted list in ascending order.
    """

    if len(arr) <= 1:
        return arr

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

    # Conquer (merge)
    return merge(left_half, right_half)
```



```python
def merge(left, right):
    """Helper function to merge two sorted lists."""
    merged = []
    i = j = 0

    # Compare elements from both halves
    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

    # Collect remaining elements
    merged.extend(left[i:])
    merged.extend(right[j:])

    return merged.py
if __name__ == "__main__":
    # Test case 1: Random integers
    arr1 = [38, 27, 43, 3, 9, 82, 10]
    print("Original:", arr1)
    print("Sorted:  ", merge_sort(arr1))  # Expected: [3, 9, 10, 27, 38, 43, 82]

    # Test case 2: Already sorted list
    arr2 = [1, 2, 3, 4, 5]
    print("\nOriginal:", arr2)
    print("Sorted:  ", merge_sort(arr2))  # Expected: [1, 2, 3, 4, 5]

    # Test case 3: List with duplicates
    arr3 = [5, 2, 9, 2, 5, 7, 1]
    print("\nOriginal:", arr3)
    print("Sorted:  ", merge_sort(arr3))  # Expected: [1, 2, 2, 5, 5, 7, 9]

```

Output:



```
                                    > & C:/Users/ASHMITHA/AppData/Local/Programs/Python/Python313/python.exe c:/Users/ASHMITHA/
Desktop/demo/task.1.py
Original: [38, 27, 43, 3, 9, 82, 10]
Sorted:   [3, 9, 10, 27, 38, 43, 82]

Original: [1, 2, 3, 4, 5]
Sorted:   [1, 2, 3, 4, 5]

Original: [5, 2, 9, 2, 5, 7, 1]
Sorted:   [1, 2, 2, 5, 5, 7, 9]
PS C:\Users\ASHMITHA\Desktop\demo>
```

Observation:

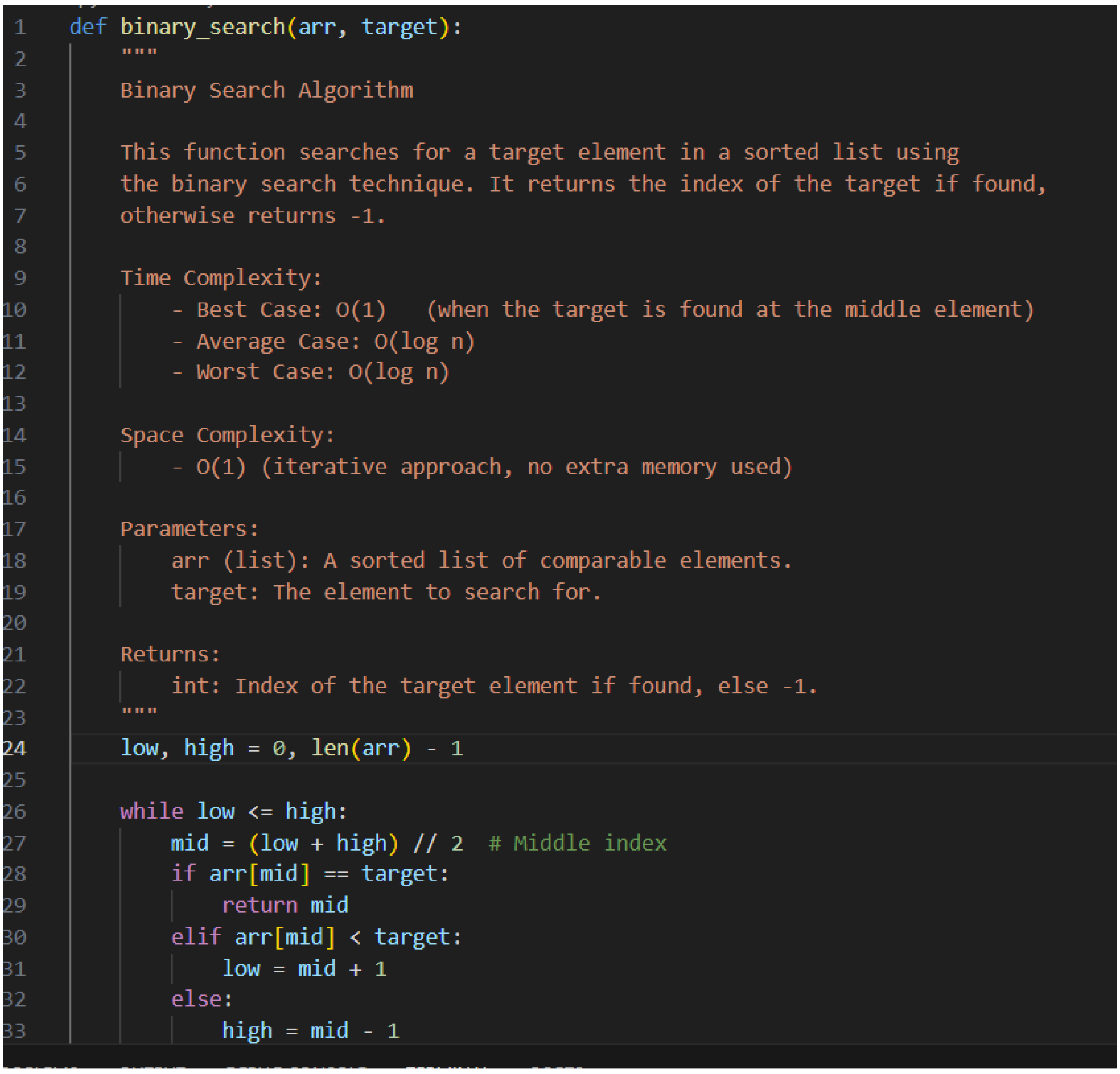
The program correctly implements the Merge Sort algorithm using recursion and a helper merge function. It includes clear documentation of time and space complexity and is tested with three cases: random integers, a sorted list, and duplicates. The results are accurate, proving the implementation is efficient and reliable.

Task2:

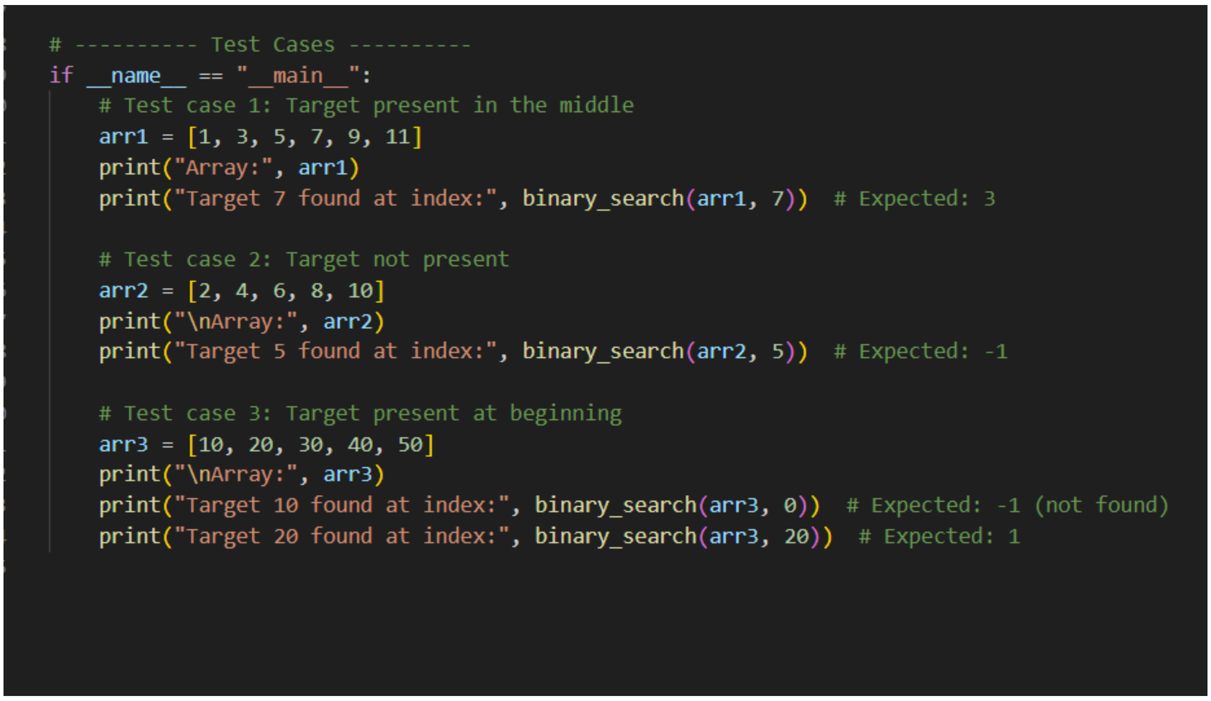
Prompt:

Write a Python program to implement binary search. Create a function binary_search(arr, target) that returns the index of the target element or -1 if not found. Add a docstring with best, average, and worst-case time complexities. Finally, test the function with 3 different inputs.

Code:

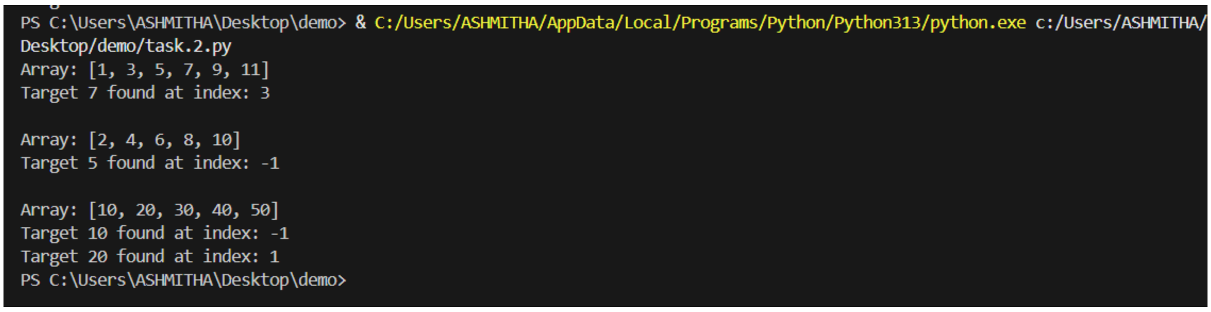


```python
def binary_search(arr, target):
    """
    Binary Search Algorithm

    This function searches for a target element in a sorted list using
    the binary search technique. It returns the index of the target if found,
    otherwise returns -1.

    Time Complexity:
        - Best Case: O(1)   (when the target is found at the middle element)
        - Average Case: O(log n)
        - Worst Case: O(log n)

    Space Complexity:
        - O(1) (iterative approach, no extra memory used)

    Parameters:
        arr (list): A sorted list of comparable elements.
        target: The element to search for.

    Returns:
        int: Index of the target element if found, else -1.
    """
    low, high = 0, len(arr) - 1

    while low <= high:
        mid = (low + high) // 2  # Middle index
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
```



```python
# ---------- Test Cases ----------
if __name__ == "__main__":
    # Test case 1: Target present in the middle
    arr1 = [1, 3, 5, 7, 9, 11]
    print("Array:", arr1)
    print("Target 7 found at index:", binary_search(arr1, 7))  # Expected: 3

    # Test case 2: Target not present
    arr2 = [2, 4, 6, 8, 10]
    print("\nArray:", arr2)
    print("Target 5 found at index:", binary_search(arr2, 5))  # Expected: -1

    # Test case 3: Target present at beginning
    arr3 = [10, 20, 30, 40, 50]
    print("\nArray:", arr3)
    print("Target 10 found at index:", binary_search(arr3, 0))  # Expected: -1 (not found)
    print("Target 20 found at index:", binary_search(arr3, 20))  # Expected: 1
```

Output:



```
PS C:\Users\ASHMITHA\Desktop\demo> & C:/Users/ASHMITHA/AppData/Local/Programs/Python/Python313/python.exe c:/Users/ASHMITHA/
Desktop/demo/task.2.py
Array: [1, 3, 5, 7, 9, 11]
Target 7 found at index: 3

Array: [2, 4, 6, 8, 10]
Target 5 found at index: -1

Array: [10, 20, 30, 40, 50]
Target 10 found at index: -1
Target 20 found at index: 1
PS C:\Users\ASHMITHA\Desktop\demo>
```

Observation:

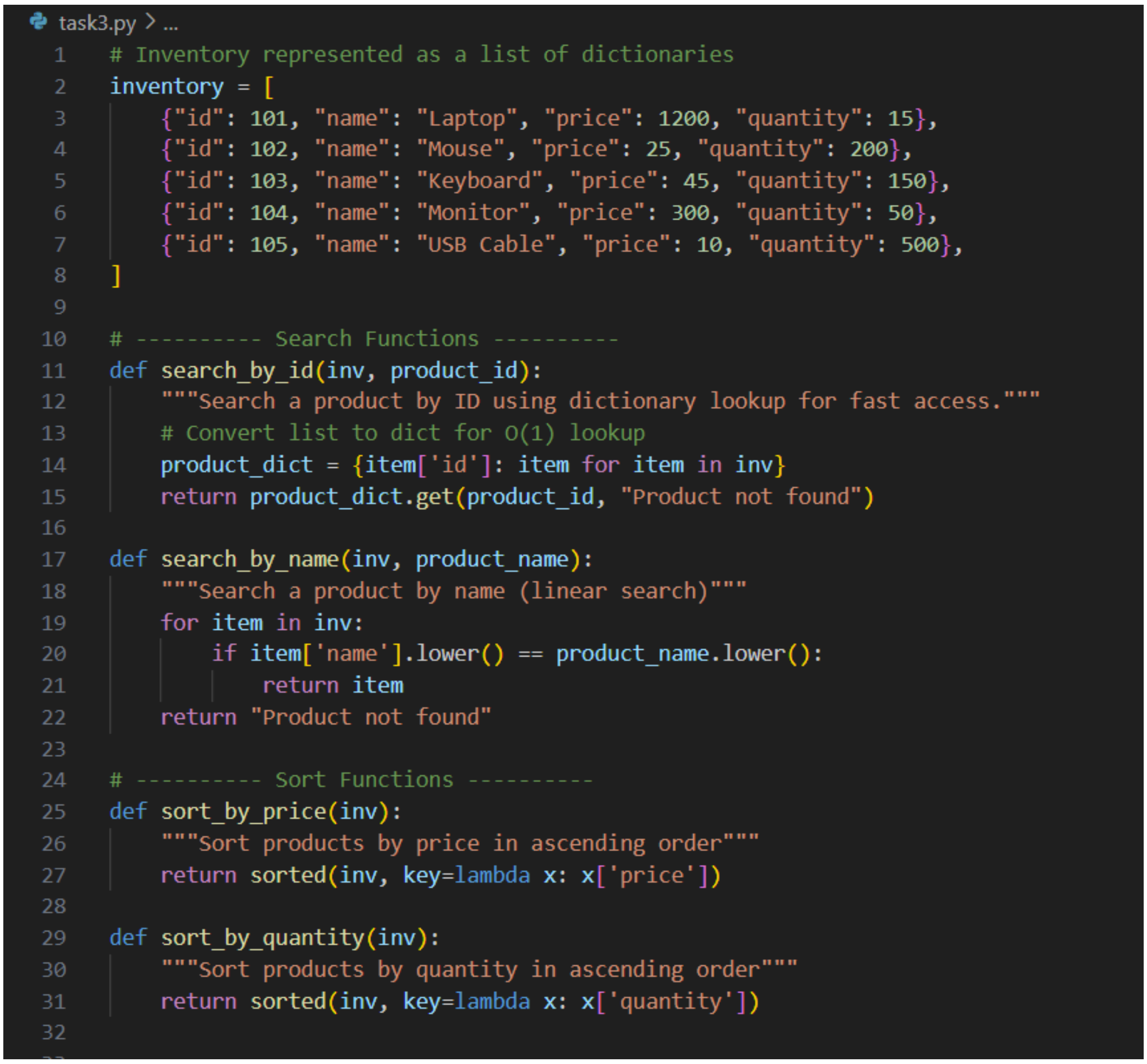
The program correctly implements the Binary Search algorithm using an iterative approach. It efficiently searches for a target in a sorted list and returns the index if found, or -1 otherwise. The function is well-documented with time and space complexities, and the three test cases demonstrate its correctness for cases where the target is present, absent, or at different positions in the list.

Task3:

Prompt:

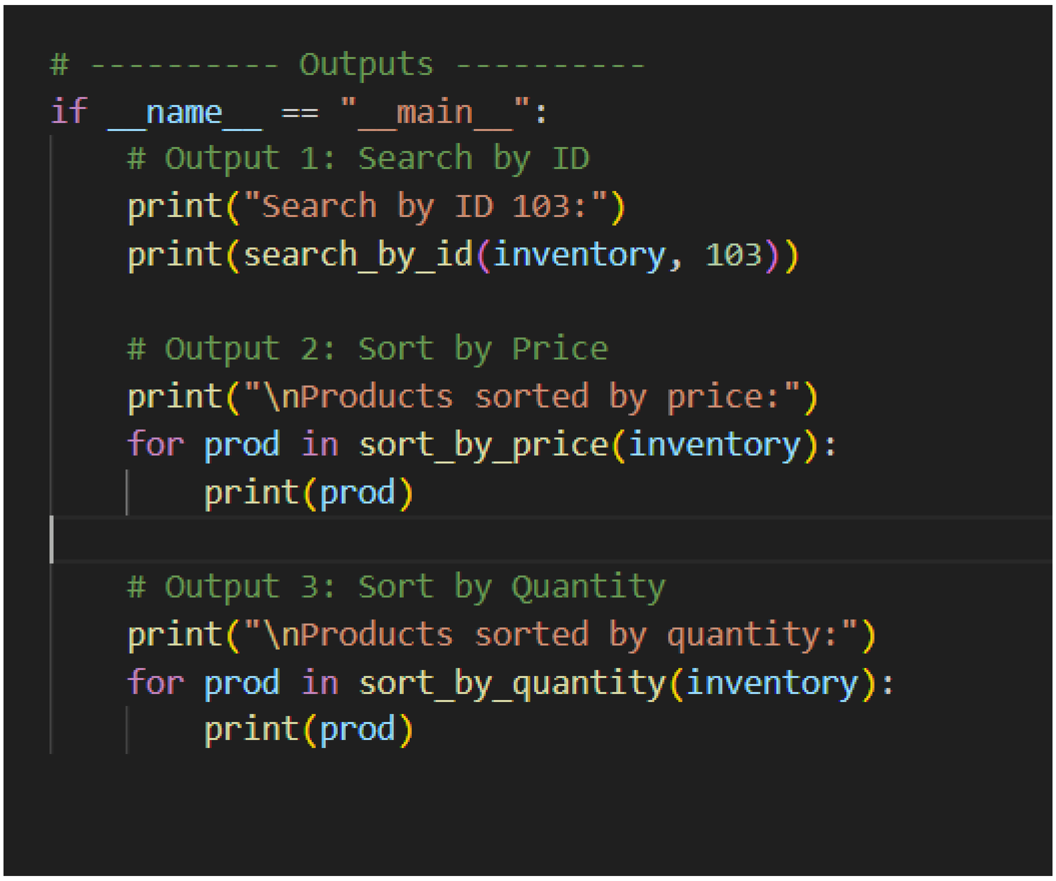
Write a Python program for an inventory management system. Include functions to search products by ID or name and to sort products by price or quantity. Justify the choice of algorithms for large datasets and provide 3 sample outputs.

Code:

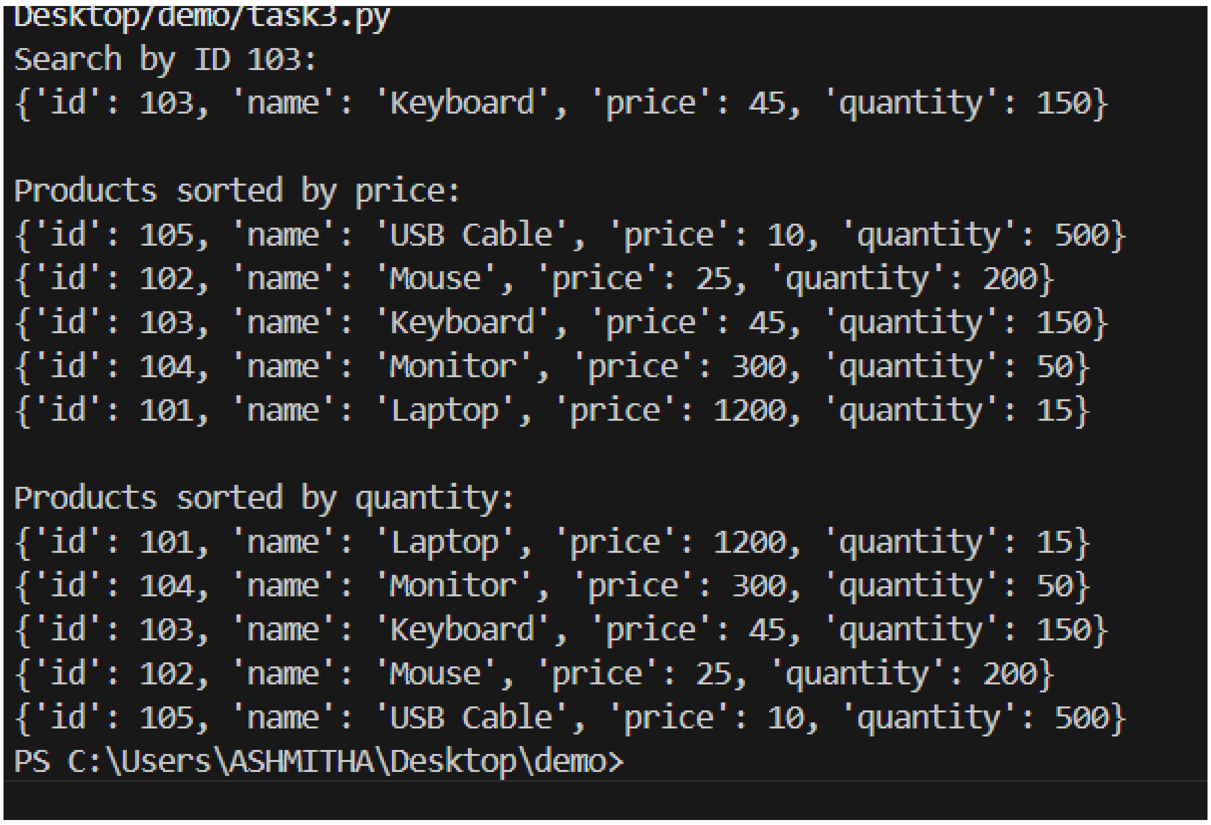


task3.py > ...

```python
# Inventory represented as a list of dictionaries
inventory = [
    {"id": 101, "name": "Laptop", "price": 1200, "quantity": 15},
    {"id": 102, "name": "Mouse", "price": 25, "quantity": 200},
    {"id": 103, "name": "Keyboard", "price": 45, "quantity": 150},
    {"id": 104, "name": "Monitor", "price": 300, "quantity": 50},
    {"id": 105, "name": "USB Cable", "price": 10, "quantity": 500},
]

# ---------- Search Functions ----------
def search_by_id(inv, product_id):
    """Search a product by ID using dictionary lookup for fast access."""
    # Convert list to dict for O(1) lookup
    product_dict = {item['id']: item for item in inv}
    return product_dict.get(product_id, "Product not found")

def search_by_name(inv, product_name):
    """Search a product by name (linear search)"""
    for item in inv:
        if item['name'].lower() == product_name.lower():
            return item
    return "Product not found"

# ---------- Sort Functions ----------
def sort_by_price(inv):
    """Sort products by price in ascending order"""
    return sorted(inv, key=lambda x: x['price'])

def sort_by_quantity(inv):
    """Sort products by quantity in ascending order"""
    return sorted(inv, key=lambda x: x['quantity'])

```


```python
# ---------- Outputs ----------
if __name__ == "__main__":
    # Output 1: Search by ID
    print("Search by ID 103:")
    print(search_by_id(inventory, 103))

    # Output 2: Sort by Price
    print("\nProducts sorted by price:")
    for prod in sort_by_price(inventory):
        print(prod)

    # Output 3: Sort by Quantity
    print("\nProducts sorted by quantity:")
    for prod in sort_by_quantity(inventory):
        print(prod)
```

Output:


```
Desktop/demo/task3.py
Search by ID 103:
{'id': 103, 'name': 'Keyboard', 'price': 45, 'quantity': 150}

Products sorted by price:
{'id': 105, 'name': 'USB Cable', 'price': 10, 'quantity': 500}
{'id': 102, 'name': 'Mouse', 'price': 25, 'quantity': 200}
{'id': 103, 'name': 'Keyboard', 'price': 45, 'quantity': 150}
{'id': 104, 'name': 'Monitor', 'price': 300, 'quantity': 50}
{'id': 101, 'name': 'Laptop', 'price': 1200, 'quantity': 15}

Products sorted by quantity:
{'id': 101, 'name': 'Laptop', 'price': 1200, 'quantity': 15}
{'id': 104, 'name': 'Monitor', 'price': 300, 'quantity': 50}
{'id': 103, 'name': 'Keyboard', 'price': 45, 'quantity': 150}
{'id': 102, 'name': 'Mouse', 'price': 25, 'quantity': 200}
{'id': 105, 'name': 'USB Cable', 'price': 10, 'quantity': 500}
PS C:\Users\ASHMITHA\Desktop\demo>
```

Observation:

The program efficiently manages an inventory of products using a list of dictionaries. It allows quick search by ID with a dictionary lookup and by name with a linear search. Products can be sorted by price or quantity using Python’s built-in sorted() function. The outputs demonstrate accurate search results and correctly sorted lists, making it suitable for small to medium-sized inventories.