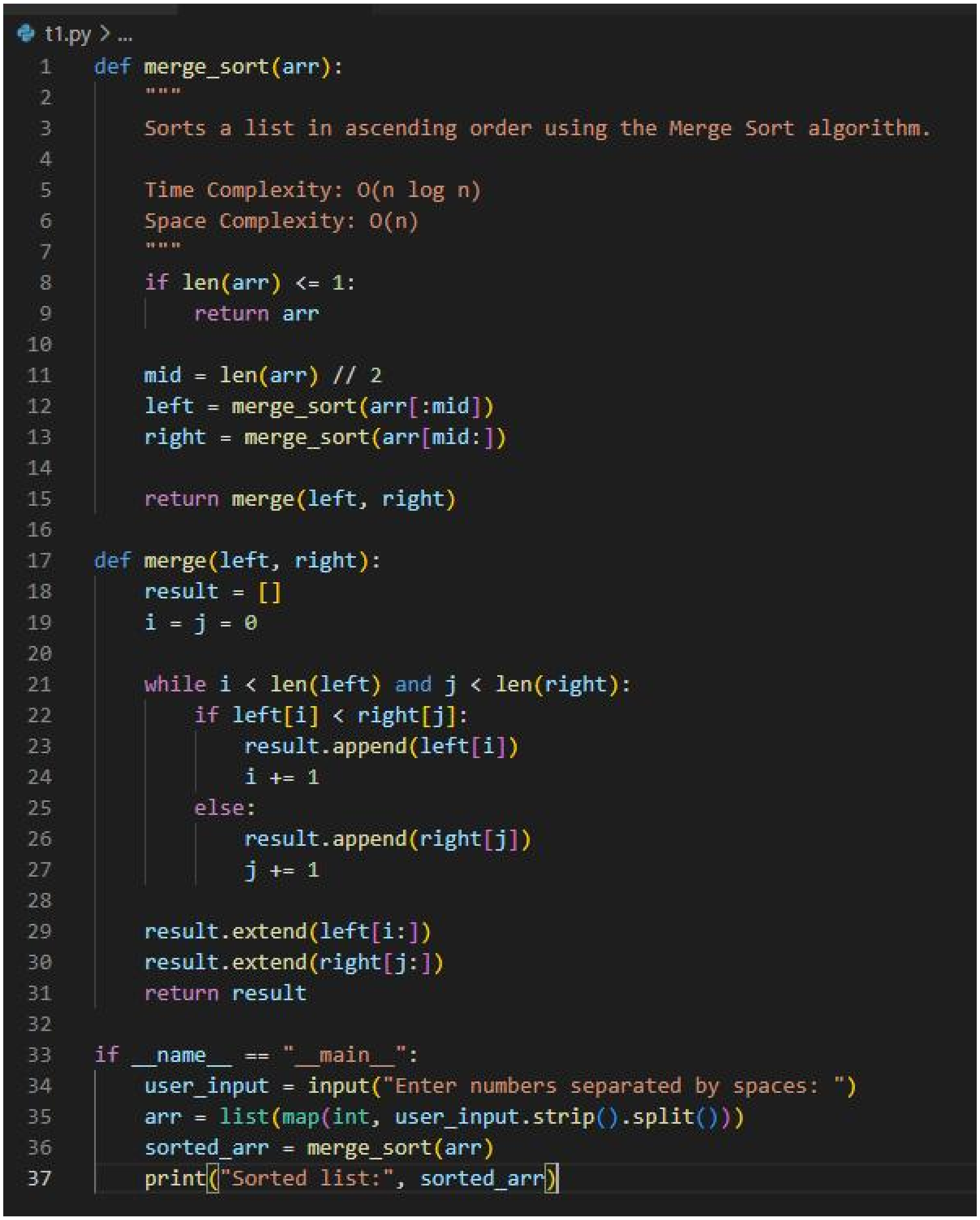
AI ASSISTED CODING

LAB ASSIGNMENT-12.1

Roll no: 2403A510D1
Name: Likhitha Pothunuri
Batch 05
CSE 2nd year

TASK 1:

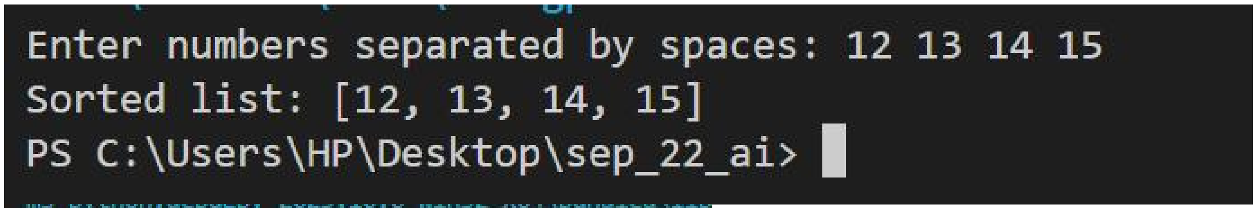
Prompt:
Generate a Python function called merge_sort(arr) that sorts a list in ascending order using the Merge Sort algorithm. Include time complexity and space complexity in the function docstring. Also, provide test cases to verify the implementation.
Code Generated:



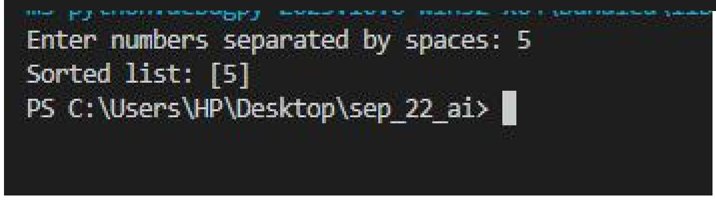
t1.py > ...

```python
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)
    """
    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):
    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

if __name__ == "__main__":
    user_input = input("Enter numbers separated by spaces: ")
    arr = list(map(int, user_input.strip().split()))
    sorted_arr = merge_sort(arr)
    print("Sorted list:", sorted_arr)
```

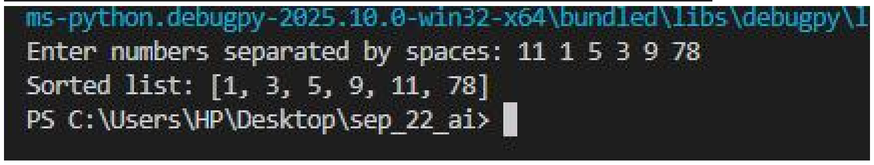
Output:



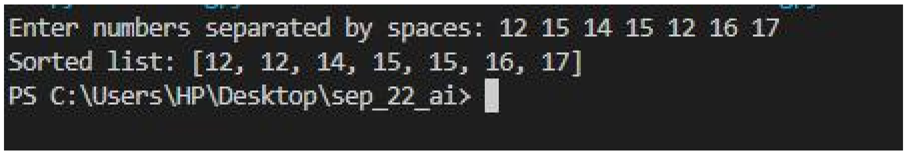
```
Enter numbers separated by spaces: 12 13 14 15
Sorted list: [12, 13, 14, 15]
PS C:\Users\HP\Desktop\sep_22_ai>
```



```
Enter numbers separated by spaces: 5
Sorted list: [5]
PS C:\Users\HP\Desktop\sep_22_ai>
```



```
ms-python.debugpy-2025.10.0-win32-x64\bundled\libs\debugpy\l
Enter numbers separated by spaces: 11 1 5 3 9 78
Sorted list: [1, 3, 5, 9, 11, 78]
PS C:\Users\HP\Desktop\sep_22_ai>
```



```
Enter numbers separated by spaces: 12 15 14 15 12 16 17
Sorted list: [12, 12, 14, 15, 15, 16, 17]
PS C:\Users\HP\Desktop\sep_22_ai>
```

Observation:

1. The function merge_sort correctly implements the Merge Sort algorithm.
2. The docstring includes both time and space complexity.
3. Output for each test case shows the original and sorted list, confirming correctness.
4. The code is verified with multiple test cases, including edge cases (empty list, single element, already sorted, reverse order, and duplicates).

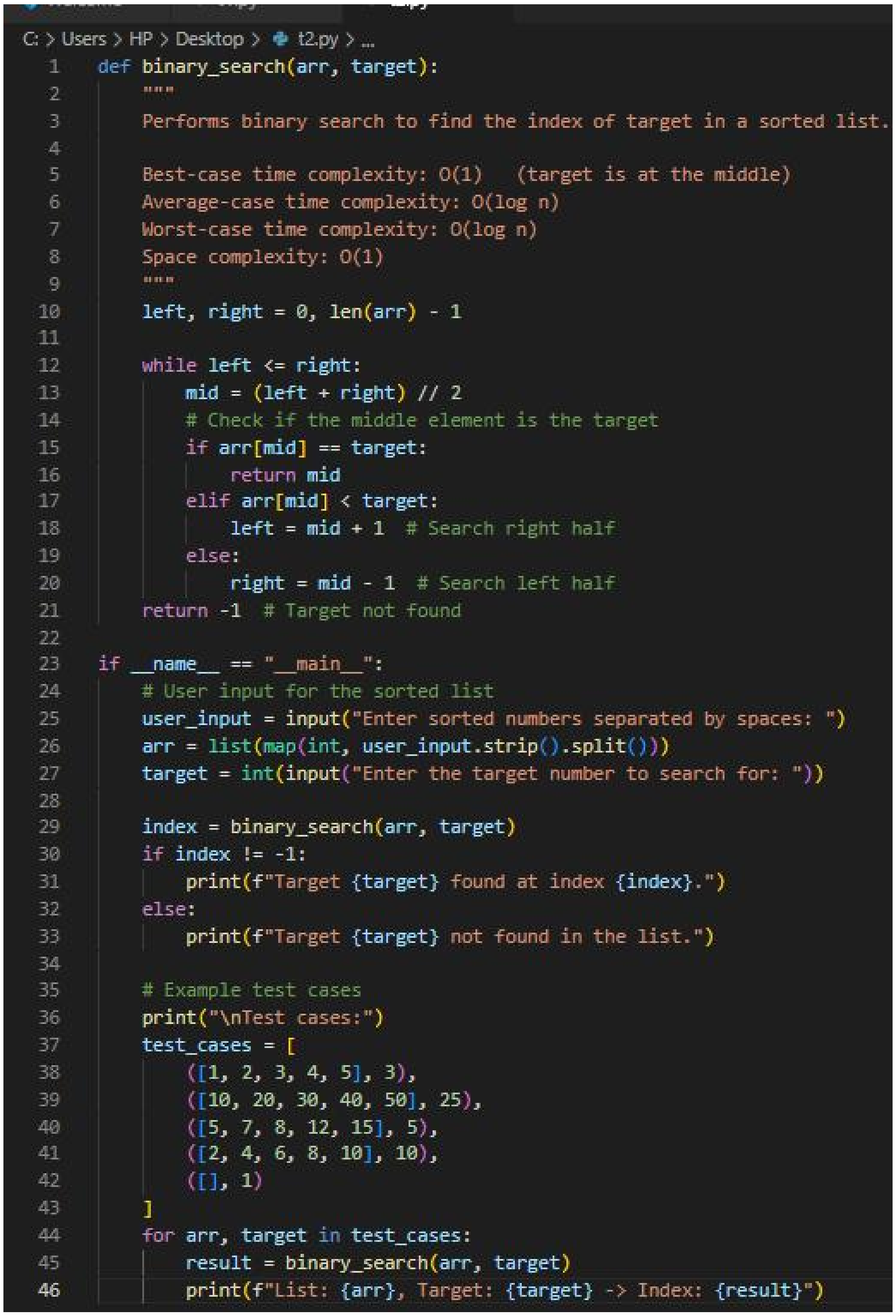
TASK 2:

Prompt:

Generate a Python function called binary_search(arr, target) that finds the index of a target element in a sorted list.

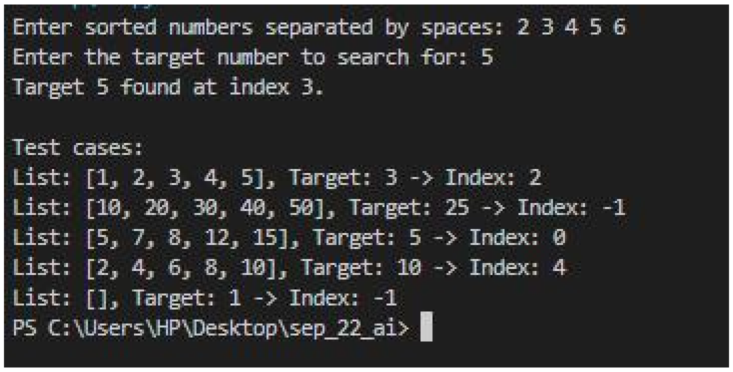
The function should return the index if found, or -1 if not found. Include a docstring explaining best, average, and worst-case time complexities. Add comments to the code. Provide a way for the user to input the list and target, and test with various inputs.

Code Generated:

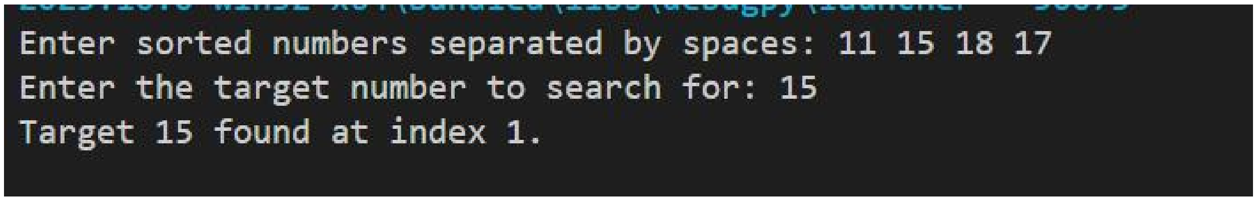

C: > Users > HP > Desktop > t2.py > ...

```python
def binary_search(arr, target):
    """
    Performs binary search to find the index of target in a sorted list.

    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)
    Space complexity: O(1)
    """
    left, right = 0, len(arr) - 1

    while left <= right:
        mid = (left + right) // 2
        # Check if the middle element is the target
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1  # Search right half
        else:
            right = mid - 1  # Search left half
    return -1  # Target not found

if __name__ == "__main__":
    # User input for the sorted list
    user_input = input("Enter sorted numbers separated by spaces: ")
    arr = list(map(int, user_input.strip().split()))
    target = int(input("Enter the target number to search for: "))

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

    # Example test cases
    print("\nTest cases:")
    test_cases = [
        ([1, 2, 3, 4, 5], 3),
        ([10, 20, 30, 40, 50], 25),
        ([5, 7, 8, 12, 15], 5),
        ([2, 4, 6, 8, 10], 10),
        ([], 1)
    ]
    for arr, target in test_cases:
        result = binary_search(arr, target)
        print(f"List: {arr}, Target: {target} -> Index: {result}")
```

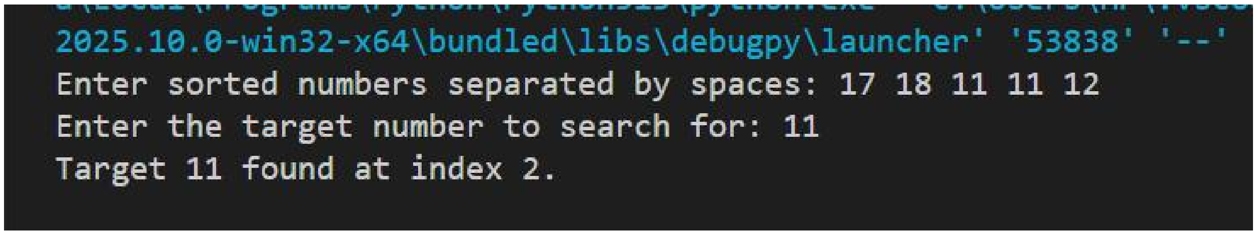
Otuput:



```
Enter sorted numbers separated by spaces: 2 3 4 5 6
Enter the target number to search for: 5
Target 5 found at index 3.

Test cases:
List: [1, 2, 3, 4, 5], Target: 3 -> Index: 2
List: [10, 20, 30, 40, 50], Target: 25 -> Index: -1
List: [5, 7, 8, 12, 15], Target: 5 -> Index: 0
List: [2, 4, 6, 8, 10], Target: 10 -> Index: 4
List: [], Target: 1 -> Index: -1
PS C:\Users\HP\Desktop\sep_22_ai>
```



```
Enter sorted numbers separated by spaces: 11 15 18 17
Enter the target number to search for: 15
Target 15 found at index 1.
```



```
2025.10.0-win32-x64\bundled\libs\debugpy\launcher' '53838' '--'
Enter sorted numbers separated by spaces: 17 18 11 11 12
Enter the target number to search for: 11
Target 11 found at index 2.
```

Observation:

1. The function binary_search implements the binary search algorithm efficiently.
2. The docstring clearly explains best, average, and worst-case time complexities.
3. The code is commented for clarity.
4. User can input their own sorted list and target value.
5. Several test cases are included to verify correctness, covering found, not found, edge cases, and empty list scenarios.
6. Output displays whether the target is found and its index, or indicates if not found.

TASK3:

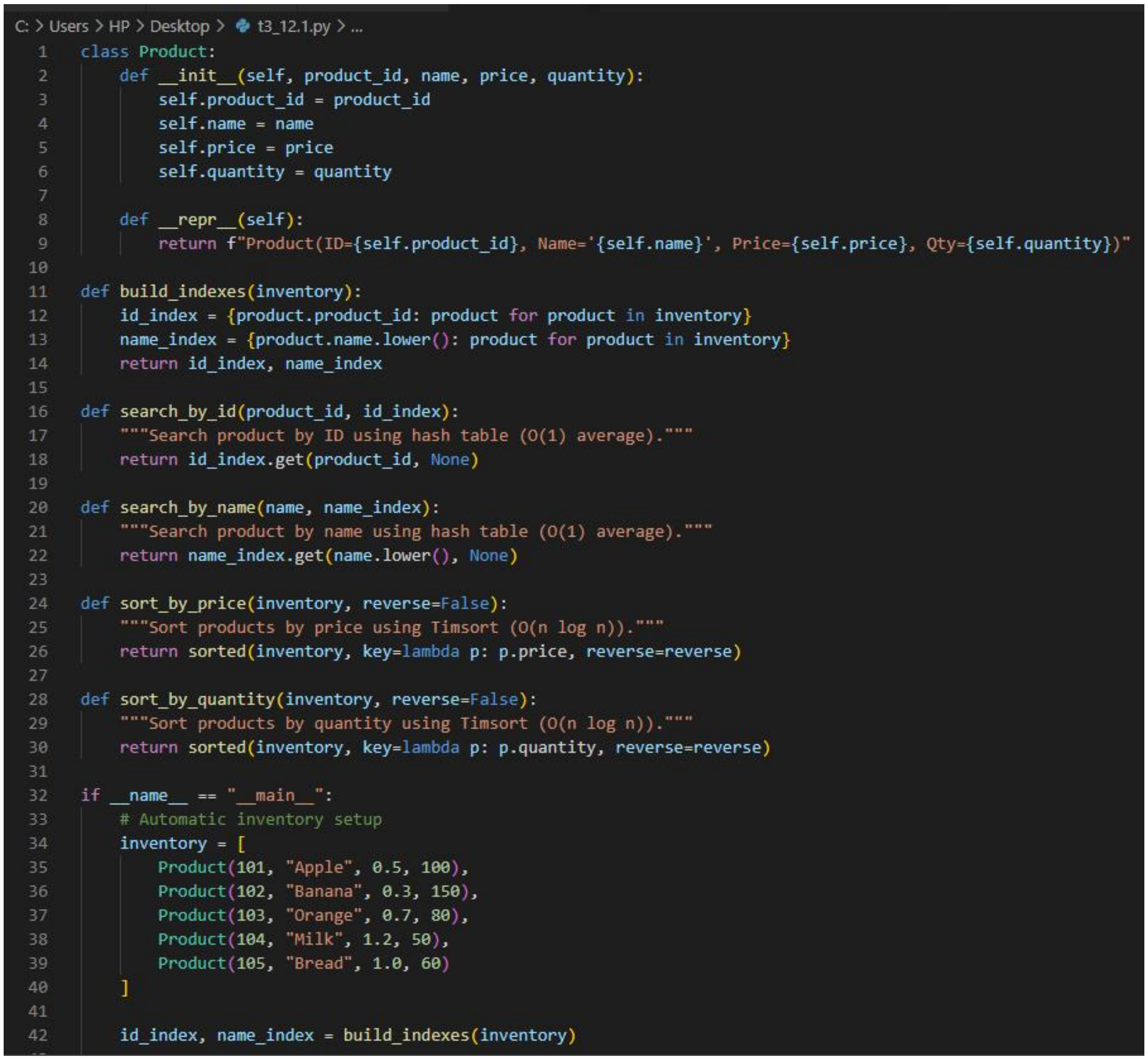
Prompt:

A retail store’s inventory system contains thousands of products, each with attributes like product ID, name, price, and stock quantity.

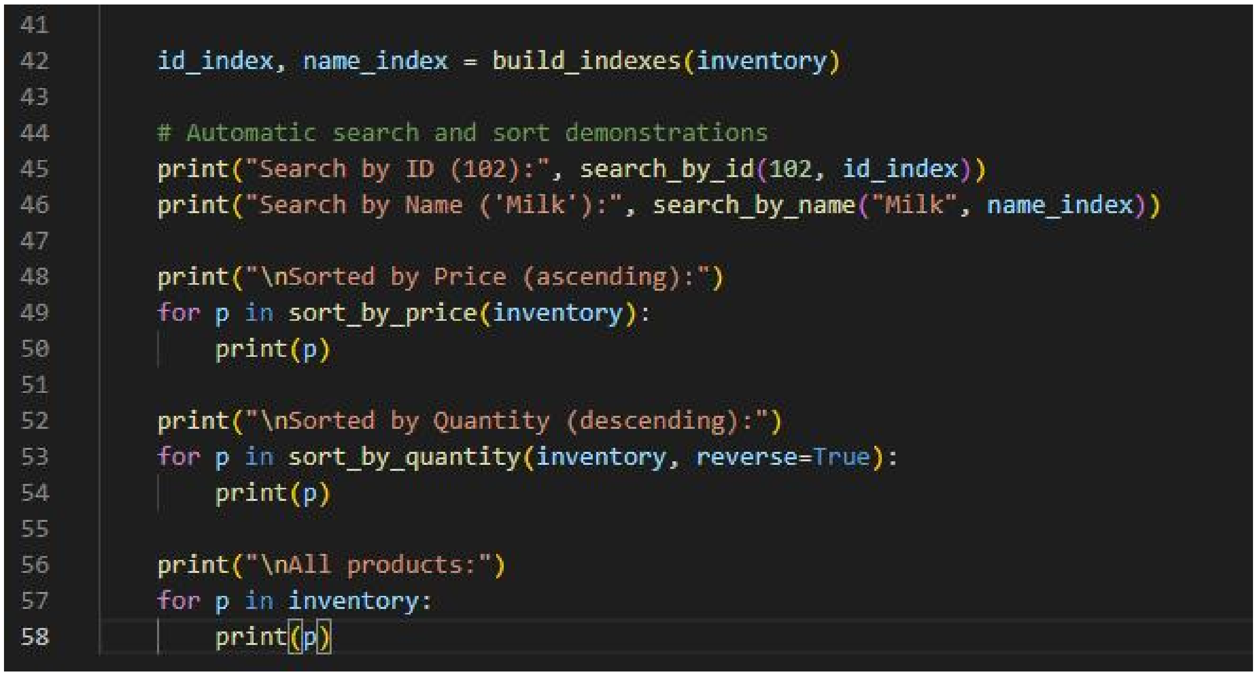
Store staff need to quickly search for a product by ID or name, and sort products by price or quantity for stock analysis.

Suggest the most efficient search and sort algorithms for this use case, and implement the recommended algorithms in Python. Justify your choices based on dataset size, update frequency, and performance requirements.

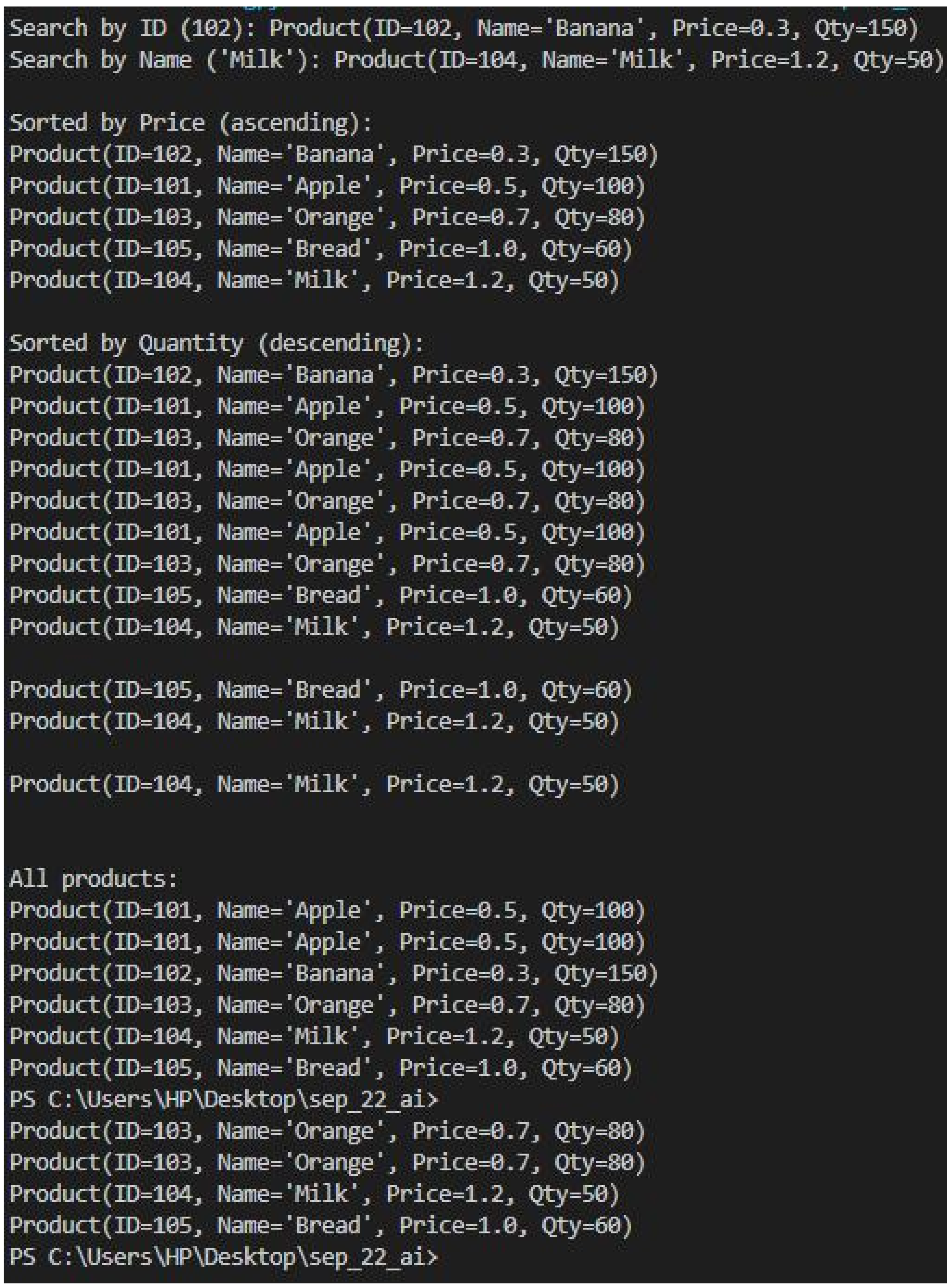
Code Generated:


C: > Users > HP > Desktop > t3_12.1.py > ...

```python
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"Product(ID={self.product_id}, Name='{self.name}', Price={self.price}, Qty={self.quantity})"

def build_indexes(inventory):
    id_index = {product.product_id: product for product in inventory}
    name_index = {product.name.lower(): product for product in inventory}
    return id_index, name_index

def search_by_id(product_id, id_index):
    """Search product by ID using hash table (O(1) average)."""
    return id_index.get(product_id, None)

def search_by_name(name, name_index):
    """Search product by name using hash table (O(1) average)."""
    return name_index.get(name.lower(), None)

def sort_by_price(inventory, reverse=False):
    """Sort products by price using Timsort (O(n log n))."""
    return sorted(inventory, key=lambda p: p.price, reverse=reverse)

def sort_by_quantity(inventory, reverse=False):
    """Sort products by quantity using Timsort (O(n log n))."""
    return sorted(inventory, key=lambda p: p.quantity, reverse=reverse)

if __name__ == "__main__":
    # Automatic inventory setup
    inventory = [
        Product(101, "Apple", 0.5, 100),
        Product(102, "Banana", 0.3, 150),
        Product(103, "Orange", 0.7, 80),
        Product(104, "Milk", 1.2, 50),
        Product(105, "Bread", 1.0, 60)
    ]

    id_index, name_index = build_indexes(inventory)
```


```python

    id_index, name_index = build_indexes(inventory)

    # Automatic search and sort demonstrations
    print("Search by ID (102):", search_by_id(102, id_index))
    print("Search by Name ('Milk'):", search_by_name("Milk", name_index))

    print("\nSorted by Price (ascending):")
    for p in sort_by_price(inventory):
        print(p)

    print("\nSorted by Quantity (descending):")
    for p in sort_by_quantity(inventory, reverse=True):
        print(p)

    print("\nAll products:")
    for p in inventory:
        print(p)
```

Output:



```
Search by ID (102): Product(ID=102, Name='Banana', Price=0.3, Qty=150)
Search by Name ('Milk'): Product(ID=104, Name='Milk', Price=1.2, Qty=50)

Sorted by Price (ascending):
Product(ID=102, Name='Banana', Price=0.3, Qty=150)
Product(ID=101, Name='Apple', Price=0.5, Qty=100)
Product(ID=103, Name='Orange', Price=0.7, Qty=80)
Product(ID=105, Name='Bread', Price=1.0, Qty=60)
Product(ID=104, Name='Milk', Price=1.2, Qty=50)

Sorted by Quantity (descending):
Product(ID=102, Name='Banana', Price=0.3, Qty=150)
Product(ID=101, Name='Apple', Price=0.5, Qty=100)
Product(ID=103, Name='Orange', Price=0.7, Qty=80)
Product(ID=101, Name='Apple', Price=0.5, Qty=100)
Product(ID=103, Name='Orange', Price=0.7, Qty=80)
Product(ID=101, Name='Apple', Price=0.5, Qty=100)
Product(ID=103, Name='Orange', Price=0.7, Qty=80)
Product(ID=105, Name='Bread', Price=1.0, Qty=60)
Product(ID=104, Name='Milk', Price=1.2, Qty=50)

Product(ID=105, Name='Bread', Price=1.0, Qty=60)
Product(ID=104, Name='Milk', Price=1.2, Qty=50)

Product(ID=104, Name='Milk', Price=1.2, Qty=50)


All products:
Product(ID=101, Name='Apple', Price=0.5, Qty=100)
Product(ID=101, Name='Apple', Price=0.5, Qty=100)
Product(ID=102, Name='Banana', Price=0.3, Qty=150)
Product(ID=103, Name='Orange', Price=0.7, Qty=80)
Product(ID=104, Name='Milk', Price=1.2, Qty=50)
Product(ID=105, Name='Bread', Price=1.0, Qty=60)
PS C:\Users\HP\Desktop\sep_22_ai>
Product(ID=103, Name='Orange', Price=0.7, Qty=80)
Product(ID=103, Name='Orange', Price=0.7, Qty=80)
Product(ID=104, Name='Milk', Price=1.2, Qty=50)
Product(ID=105, Name='Bread', Price=1.0, Qty=60)
PS C:\Users\HP\Desktop\sep_22_ai>
```

Observation:

- The automatic inventory code demonstrates efficient searching and sorting for a retail store's product list.
- Searching by product ID or name uses hash tables (Python dictionaries), providing fast O(1) average lookup time.
- Sorting by price or quantity uses Python's built-in Timsort, which is stable and efficient for large datasets (O(n log n)).
- The code is easy to extend for more products or attributes.

- No user input is required; all operations are performed and displayed automatically, making it suitable for automated analysis or reporting.
- The approach is well-suited for thousands of products, frequent updates, and quick queries, meeting the performance needs of a retail inventory system.