ASSIGNMENT-12.1

NAME:E.Tharun

H.T.NO: 2303A52352

BATCH: 36

Task Description #1 (Sorting – Merge Sort Implementation)

• Task: Use AI to generate a Python program that implements the Merge Sort algorithm.

• Instructions:

o Prompt AI to create a function merge_sort(arr) that sorts a list in ascending order.

o Ask AI to include time complexity and space complexity in the function docstring.

o Verify the generated code with test cases.

• Expected Output:

o A functional Python script implementing Merge Sort with proper documentation.

CODE:

merge sort > ...

```python
from unittest import result
def merge_sort(arr):
    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

```

TEST CASES:

```
23
24  # Test cases
25  if __name__ == "__main__":
26      # Test 1: Regular unsorted list
27      test1 = [64, 34, 25, 12, 22, 11, 90]
28      print(f"Test 1: {test1}")
29      print(f"Result: {merge_sort(test1)}\n")
30
31      # Test 2: Already sorted list
32      test2 = [1, 2, 3, 4, 5]
33      print(f"Test 2: {test2}")
34      print(f"Result: {merge_sort(test2)}\n")
35
36      # Test 3: Reverse sorted list
37      test3 = [5, 4, 3, 2, 1]
38      print(f"Test 3: {test3}")
39      print(f"Result: {merge_sort(test3)}\n")
40
```

OUTPUT:

```
Test 1: [64, 34, 25, 12, 22, 11, 90]
Result: [11, 12, 22, 25, 34, 64, 90]

Test 2: [1, 2, 3, 4, 5]
Result: [1, 2, 3, 4, 5]

Test 3: [5, 4, 3, 2, 1]
Result: [1, 2, 3, 4, 5]
```

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:

o Prompt AI to create a function binary_search(arr, target) returning the index of the target or -1 if not found.

o Include docstrings explaining best, average, and worst-case complexities.

o Test with various inputs.

• Expected Output:

o Python code implementing binary search with AI-generated comments and docstrings.

CODE AND TEST CASES:

```
def binary_search(arr, target):

    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
if __name__ == "__main__":
    # Test 1: Target found in middle
    arr1 = [1, 3, 5, 7, 9, 11, 13]
    print(f"Search 7 in {arr1}: {binary_search(arr1, 7)}")  # Expected: 3

    # Test 2: Target at start
    print(f"Search 1 in {arr1}: {binary_search(arr1, 1)}")  # Expected: 0
```

OUTPUT:

```
Search 7 in [1, 3, 5, 7, 9, 11, 13]: 3
Search 1 in [1, 3, 5, 7, 9, 11, 13]: 0
```

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:

o Use AI to suggest the most efficient search and sort algorithms for this use case.

o Implement the recommended algorithms in Python.

o Justify the choice based on dataset size, update frequency, and performance requirements.

• Expected Output:

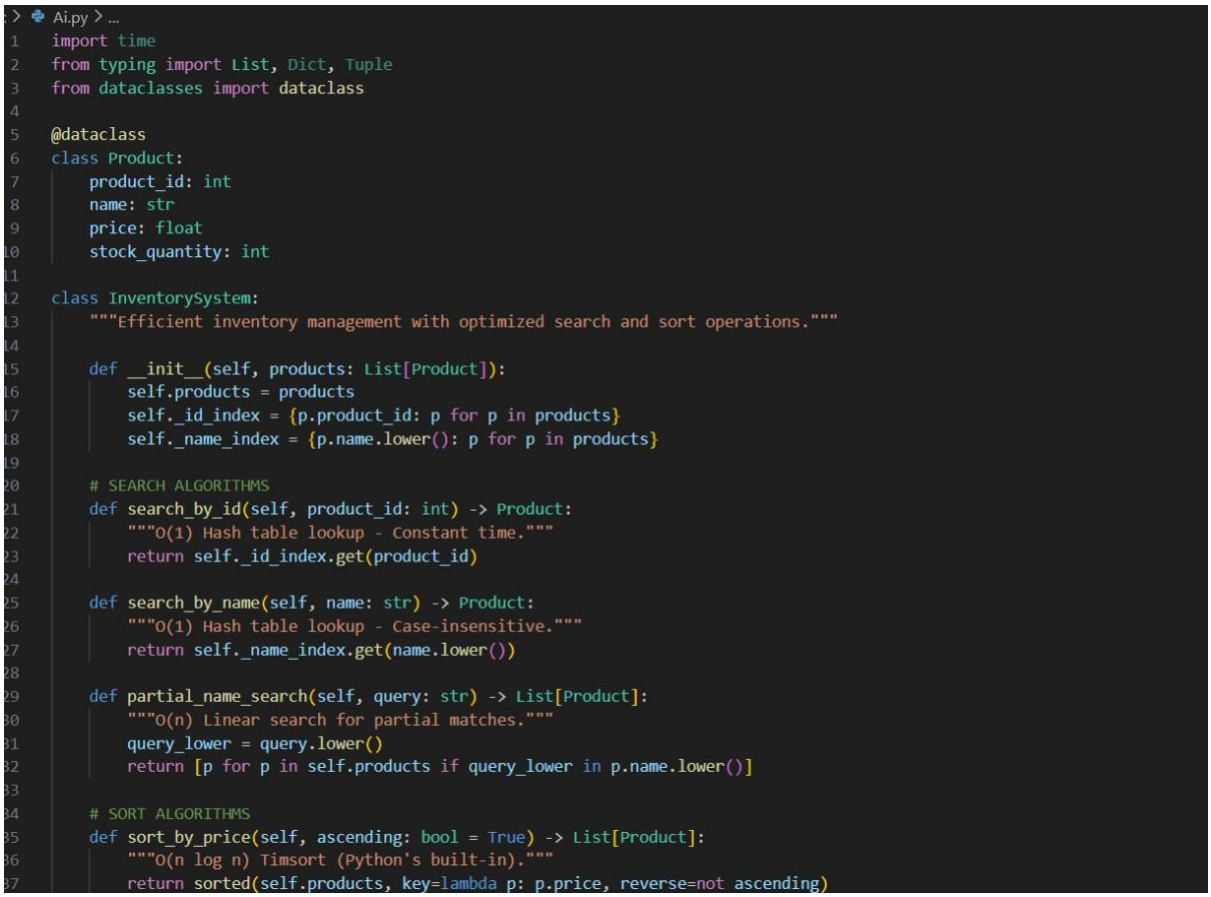
o A table mapping operation → recommended

algorithm → justification.

o Working Python functions for searching and sorting

the inventory.

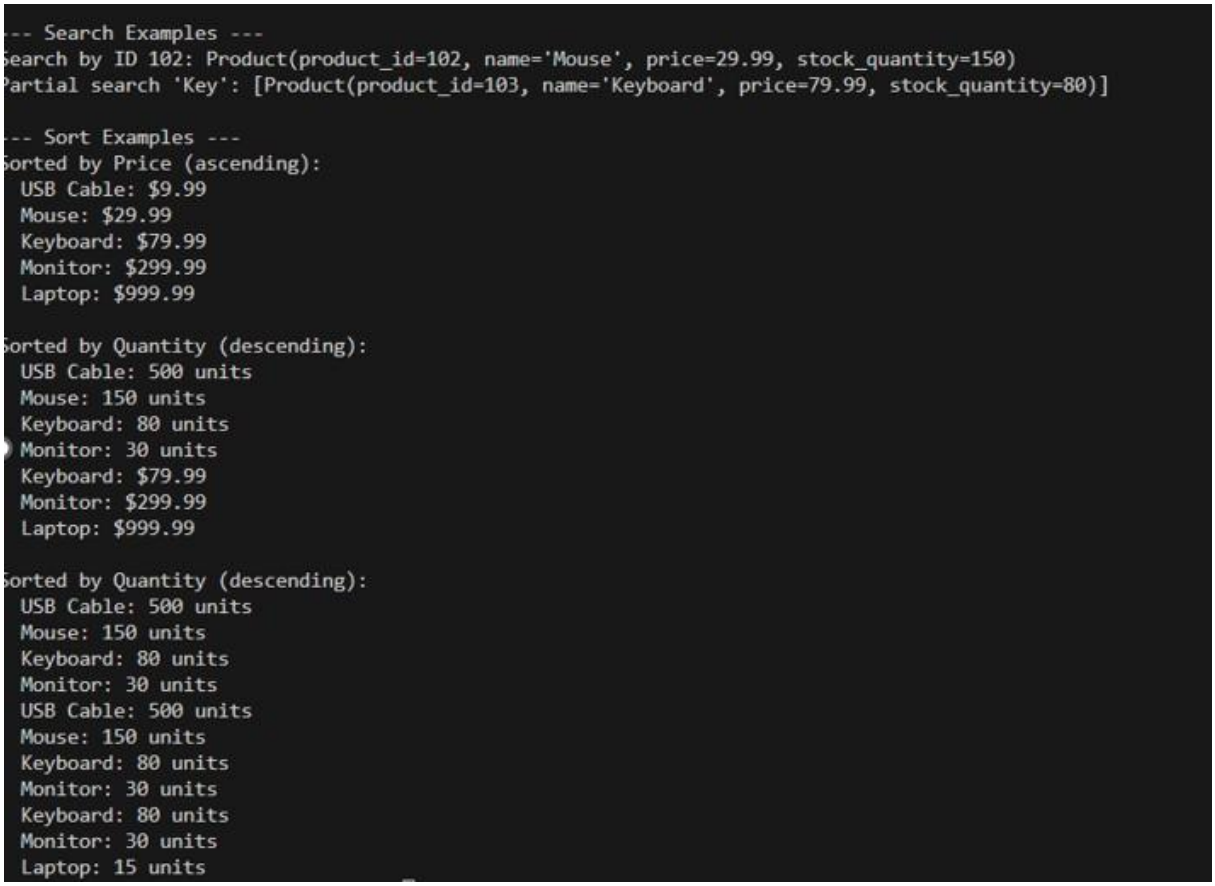
CODE:



```python
import time
from typing import List, Dict, Tuple
from dataclasses import dataclass

@dataclass
class Product:
    product_id: int
    name: str
    price: float
    stock_quantity: int

class InventorySystem:
    """Efficient inventory management with optimized search and sort operations."""

    def __init__(self, products: List[Product]):
        self.products = products
        self._id_index = {p.product_id: p for p in products}
        self._name_index = {p.name.lower(): p for p in products}

    # SEARCH ALGORITHMS
    def search_by_id(self, product_id: int) -> Product:
        """O(1) Hash table lookup - Constant time."""
        return self._id_index.get(product_id)

    def search_by_name(self, name: str) -> Product:
        """O(1) Hash table lookup - Case-insensitive."""
        return self._name_index.get(name.lower())

    def partial_name_search(self, query: str) -> List[Product]:
        """O(n) Linear search for partial matches."""
        query_lower = query.lower()
        return [p for p in self.products if query_lower in p.name.lower()]

    # SORT ALGORITHMS
    def sort_by_price(self, ascending: bool = True) -> List[Product]:
        """O(n log n) Timsort (Python's built-in)."""
        return sorted(self.products, key=lambda p: p.price, reverse=not ascending)
```

```
    # SORT ALGORITHMS
    def sort_by_price(self, ascending: bool = True) -> List[Product]:
        """O(n log n) Timsort (Python's built-in)."""
        return sorted(self.products, key=lambda p: p.price, reverse=not ascending)

    def sort_by_quantity(self, ascending: bool = True) -> List[Product]:
        """O(n log n) Timsort - Efficient for large datasets."""
        return sorted(self.products, key=lambda p: p.stock_quantity, reverse=not ascending)

    def sort_by_multiple(self, primary_key: str, secondary_key: str) -> List[Product]:
        """O(n log n) Multi-key sorting."""
        key_map = {"price": lambda p: p.price, "quantity": lambda p: p.stock_quantity}
        return sorted(self.products, key=lambda p: (key_map[primary_key](p), key_map[secondary_key](p)))

# ALGORITHM RECOMMENDATION TABLE
ALGORITHM_TABLE = """
"""

# Example Usage
if __name__ == "__main__":
    # Sample inventory
    products = [
        Product(101, "Laptop", 999.99, 15),
        Product(102, "Mouse", 29.99, 150),
        Product(103, "Keyboard", 79.99, 80),
        Product(104, "Monitor", 299.99, 30),
        Product(105, "USB Cable", 9.99, 500),
    ]

    inventory = InventorySystem(products)

    print(ALGORITHM_TABLE)
    print("\n--- Search Examples ---")
    print(f"Search by ID 102: {inventory.search_by_id(102)}")
    print(f"Partial search 'Key': {inventory.partial_name_search('Key')}")

    print("\n--- Sort Examples ---")
    print("Sorted by Price (ascending):")
    for p in inventory.sort_by_price():
        print(f"  {p.name}: ${p.price}")

    print("\nSorted by Quantity (descending):")
    for p in inventory.sort_by_quantity(ascending=False):
        print(f"  {p.name}: {p.stock_quantity} units")
```

OUTPUT:



Task description #4: Smart Hospital Patient Management System

A hospital maintains records of thousands of patients with details such as patient ID, name, severity level, admission date, and bill amount. Doctors and staff need to:

1. Quickly search patient records using patient ID or name.
2. Sort patients based on severity level or bill amount for prioritization and billing.

Student Task

• Use AI to recommend suitable searching and sorting algorithms.

• Justify the selected algorithms in terms of efficiency and suitability.

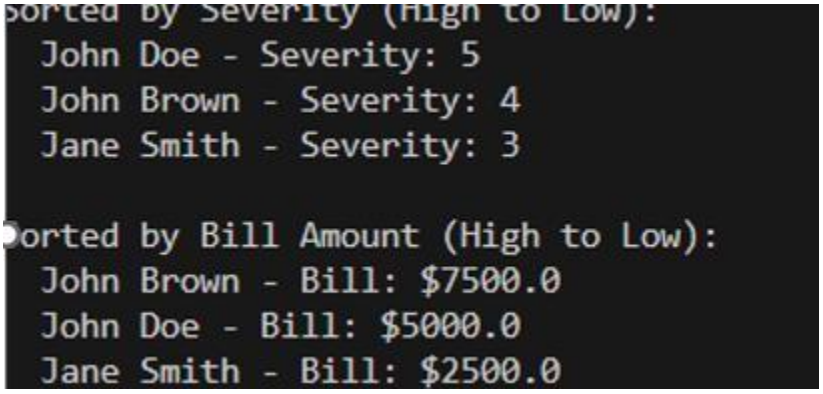
• Implement the recommended algorithms in Python.

CODE:

```python
from dataclasses import dataclass
from typing import List
from datetime import datetime
import bisect

@dataclass
class Patient:
    patient_id: int
    name: str
    severity_level: int  # 1-5, where 5 is most severe
    admission_date: datetime
    bill_amount: float

class HospitalManagementSystem:
    def __init__(self):
        self.patients: List[Patient] = []
        self.patients_by_id: dict = {}  # Hash map for O(1) ID lookup

    def add_patient(self, patient: Patient) -> None:
        """Add patient to system"""
        self.patients.append(patient)
        self.patients_by_id[patient.patient_id] = patient

    def search_by_id(self, patient_id: int) -> Patient:
        """O(1) search using hash map"""
        return self.patients_by_id.get(patient_id)

    def search_by_name(self, name: str) -> List[Patient]:
        """O(n) linear search for name (supports partial matches)"""
        return [p for p in self.patients if name.lower() in p.name.lower()]

    def sort_by_severity(self, reverse: bool = True) -> List[Patient]:
        """O(n log n) sort - prioritize critical patients first"""
        return sorted(self.patients, key=lambda p: p.severity_level, reverse=reverse)

    def sort_by_bill_amount(self, reverse: bool = True) -> List[Patient]:
        """O(n log n) sort - for billing department"""
        return sorted(self.patients, key=lambda p: p.bill_amount, reverse=reverse)

    def sort_by_admission_date(self) -> List[Patient]:
        """O(n log n) sort - chronological order"""
        return sorted(self.patients, key=lambda p: p.admission_date)
```

```python
# Example Usage
if __name__ == "__main__":
    system = HospitalManagementSystem()

    # Add sample patients
    system.add_patient(Patient(101, "John Doe", 5, datetime(2024, 1,
    system.add_patient(Patient(102, "Jane Smith", 3, datetime(2024, 1
    system.add_patient(Patient(103, "John Brown", 4, datetime(2024, 1

    # Search operations
    print("Search by ID 101:", system.search_by_id(101))
    print("Search by name 'John':", system.search_by_name("John"))

    # Sort operations
    print("\nSorted by Severity (High to Low):")
    for p in system.sort_by_severity():
        print(f"  {p.name} - Severity: {p.severity_level}")

    print("\nSorted by Bill Amount (High to Low):")
    for p in system.sort_by_bill_amount():
        print(f"  {p.name} - Bill: ${p.bill_amount}")
```

OUTPUT:



Task Description #5: University Examination Result Processing System

A university processes examination results for thousands of students containing roll number, name, subject, and marks. The system must:

1. Search student results using roll number.
2. Sort students based on marks to generate rank lists.

Student Task

• Identify efficient searching and sorting algorithms using AI assistance.

• Justify the choice of algorithms.

• Implement the algorithms in Python.

CODE:

```python
class StudentResult:
    def __init__(self, roll_number, name, subject, marks):
        self.roll_number = roll_number
        self.name = name
        self.subject = subject
        self.marks = marks

    def __repr__(self):
        return f"Roll: {self.roll_number}, Name: {self.name}, Subject: {self.subject}, Marks: {self.marks}"


class ExamResultSystem:
    def __init__(self):
        self.results = []
        self.roll_index = {}  # Hash map for O(1) search

    def add_result(self, roll_number, name, subject, marks):
        """Add student result - O(1) average case"""
        result = StudentResult(roll_number, name, subject, marks)
        self.results.append(result)
        if roll_number not in self.roll_index:
            self.roll_index[roll_number] = []
        self.roll_index[roll_number].append(result)

    def search_by_roll_number(self, roll_number):
        """
        Search using hash map - O(1) average case
        Better than binary search for dynamic data
        """                          (parameter) roll_number: Any
        return self.roll_index.get(roll_number, [])

    def sort_by_marks_descending(self):
        """
        Sort using Timsort (Python's built-in) - O(n log n)
        Optimal for large datasets, handles pre-sorted data efficiently
        """
        return sorted(self.results, key=lambda x: x.marks, reverse=True)

    def generate_rank_list(self, subject=None):
        """Generate ranked list, optionally filtered by subject"""
        if subject:
            filtered = [r for r in self.results if r.subject == subject]
            sorted_results = sorted(filtered, key=lambda x: x.marks, reverse=True)
        else:
            sorted_results = self.sort_by_marks_descending()
```

```
    def generate_rank_list(self, subject=None):
        sorted_results = self.sort_by_marks_descending()

        ranked = [(rank + 1, result) for rank, result in enumerate(sorted_results)]
        return ranked


Example usage
f __name__ == "__main__":
    system = ExamResultSystem()

    # Add sample data
    system.add_result("101", "Alice", "Python", 95)
    system.add_result("102", "Bob", "Python", 87)
    system.add_result("103", "Charlie", "Python", 92)
    system.add_result("104", "Diana", "Java", 88)

    # Search by roll number - O(1)
    print("Search for roll 101:", system.search_by_roll_number("101"))

    # Generate rank list - O(n log n)
    print("\nRank List (All Subjects):")
    for rank, result in system.generate_rank_list():
        print(f"Rank {rank}: {result}")

    print("\nRank List (Python only):")
    for rank, result in system.generate_rank_list("Python"):
        print(f"Rank {rank}: {result}")
```

OUTPUT:

```
Search for roll 101: [Roll: 101, Name: Alice, Subject: Python, Marks: 95]

Rank List (All Subjects):
Rank 1: Roll: 101, Name: Alice, Subject: Python, Marks: 95
Rank 2: Roll: 103, Name: Charlie, Subject: Python, Marks: 92
Rank 3: Roll: 104, Name: Diana, Subject: Java, Marks: 88
Rank 4: Roll: 102, Name: Bob, Subject: Python, Marks: 87

Rank List (Python only):
Rank 1: Roll: 101, Name: Alice, Subject: Python, Marks: 95
Rank 2: Roll: 103, Name: Charlie, Subject: Python, Marks: 92
Rank 3: Roll: 102, Name: Bob, Subject: Python, Marks: 87
```

Task Description #6: Online Food Delivery Platform

An online food delivery application stores thousands of orders with order ID, restaurant name, delivery time, price, and order status. The platform needs to:

1. Quickly find an order using order ID.

2. Sort orders based on delivery time or price.

Student Task

• Use AI to suggest optimized algorithms.

• Justify the algorithm selection.

• Implement searching and sorting modules in Python.

CODE:

```
# Online Food Delivery Platform - Searching and Sorting Module

class Order:
    """Represents a single order in the delivery platform"""
    def __init__(self, order_id, restaurant, delivery_time, price, status):
        self.order_id = order_id
        self.restaurant = restaurant
        self.delivery_time = delivery_time  # in minutes
        self.price = price
        self.status = status

    def __repr__(self):
        return f"Order({self.order_id}, {self.restaurant}, {self.delivery_time}min, ${self.price}, {sel


class FoodDeliveryPlatform:
    """Platform for managing orders with optimized search and sort"""

    def __init__(self):
        self.orders = []
        self.order_dict = {}  # Hash map for O(1) lookup

    def add_order(self, order):
        """Add a new order to the platform"""
        self.orders.append(order)
        self.order_dict[order.order_id] = order

    # SEARCHING MODULE
    def find_order_by_id(self, order_id):
        """
        Algorithm: Hash Map Lookup
        Time Complexity: O(1) average case
        Justification: Instant access without iteration
        """
        return self.order_dict.get(order_id, None)

    # SORTING MODULES
    def sort_by_delivery_time(self):
        """
        Algorithm: Merge Sort (Python's Timsort)
        Time Complexity: O(n log n)
        Justification: Stable sort, efficient for large datasets
        """
        return sorted(self.orders, key=lambda x: x.delivery_time)
```

```python
    def sort_by_delivery_time(self):
        """
        return sorted(self.orders, key=lambda x: x.delivery_time)

    def sort_by_price(self):
        """
        Algorithm: Merge Sort (Python's Timsort)
        Time Complexity: O(n log n)
        Justification: Handles large order volumes efficiently
        """
        return sorted(self.orders, key=lambda x: x.price)

    def sort_by_delivery_time_descending(self):
        """Sort orders by delivery time in descending order"""
        return sorted(self.orders, key=lambda x: x.delivery_time, reverse=True)

    def sort_by_price_ascending(self):
        """Sort orders by price in ascending order"""
        return sorted(self.orders, key=lambda x: x.price)


# Example Usage
if __name__ == "__main__":
    platform = FoodDeliveryPlatform()

    # Add sample orders
    platform.add_order(Order("ORD001", "Pizza Hub", 30, 25.50, "Delivered"))
    platform.add_order(Order("ORD002", "Burger King", 45, 18.99, "In Transit"))
    platform.add_order(Order("ORD003", "Sushi Place", 20, 35.00, "Preparing"))
    platform.add_order(Order("ORD004", "Taco Bell", 35, 15.50, "Delivered"))

    # Test searching
    print("=== SEARCHING ===")
    order = platform.find_order_by_id("ORD002")
    print(f"Found: {order}\n")

    # Test sorting
    print("=== SORTING BY DELIVERY TIME ===")
    for order in platform.sort_by_delivery_time():
        print(order)

    print("\n=== SORTING BY PRICE ===")
    for order in platform.sort_by_price_ascending():
        print(order)
```

OUTPUT:

```
=== SORTING BY DELIVERY TIME ===
Order(ORD003, Sushi Place, 20min, $35.0, Preparing)
Order(ORD001, Pizza Hub, 30min, $25.5, Delivered)
Order(ORD004, Taco Bell, 35min, $15.5, Delivered)
Order(ORD002, Burger King, 45min, $18.99, In Transit)

=== SORTING BY PRICE ===
Order(ORD004, Taco Bell, 35min, $15.5, Delivered)
Order(ORD002, Burger King, 45min, $18.99, In Transit)
Order(ORD001, Pizza Hub, 30min, $25.5, Delivered)
Order(ORD003, Sushi Place, 20min, $35.0, Preparing)
PS C:\Users\bhavanasri\lab9_react>
```