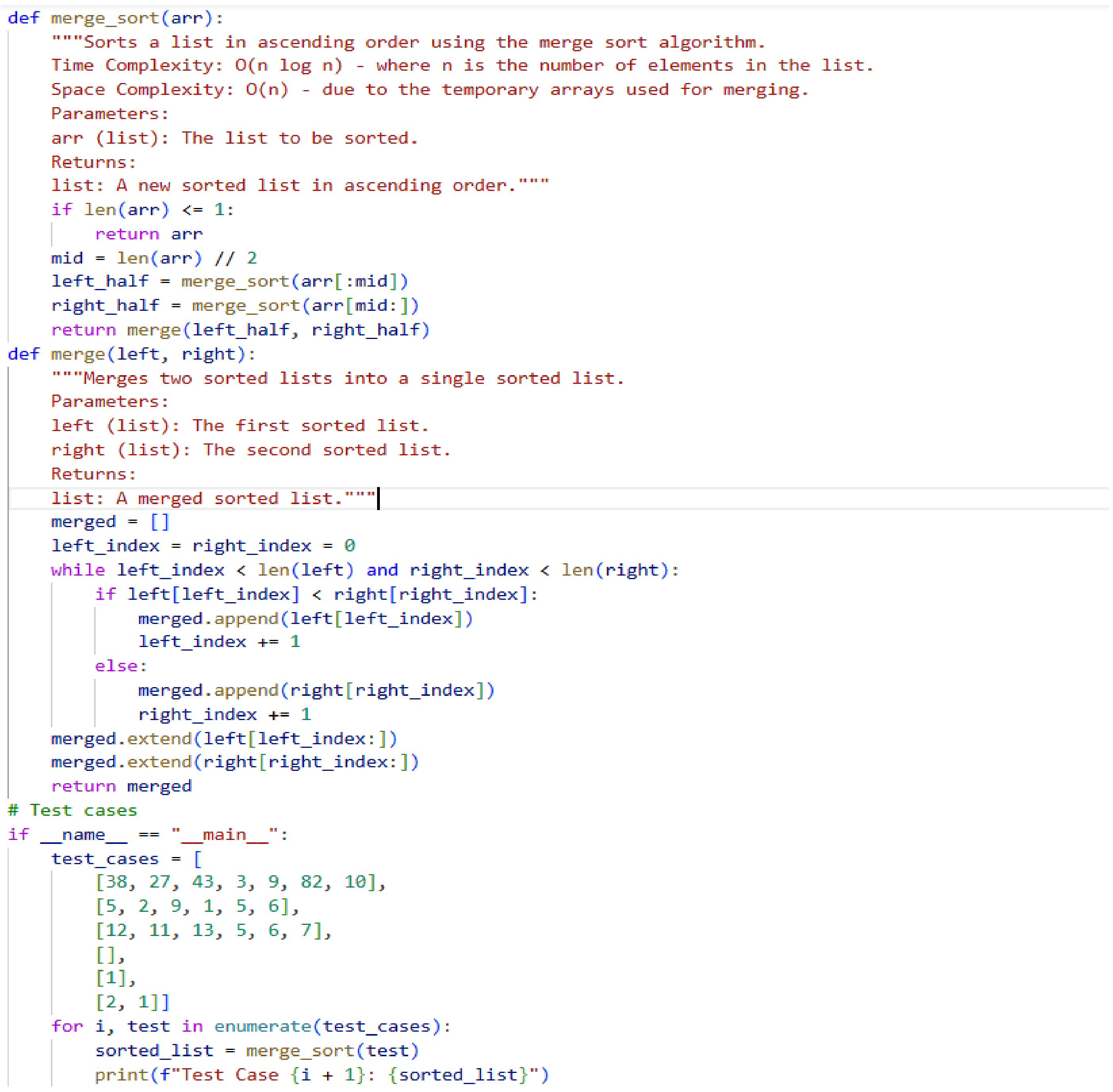
Lab Assignment 12.1

Program : B. Tech (CSE)
Course Title : AI Assisted Coding
Course Code : 23CS002PC304
Semester : III
Academic Session : 2025-2026
Name of Student : Sruthi
Enrollment No. : 2403A51L10
Batch No. : 51
Date : 27-02-2026

Task 1: Sorting – Merge Sort Implementation

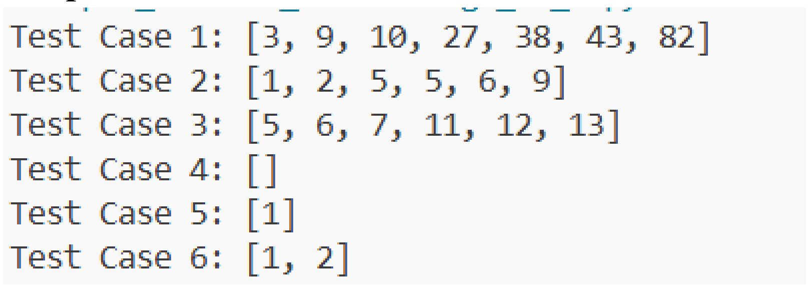
Prompt: Generate a python program that creates a function merge_sort(arr) that sorts a list in ascending order and include time complexity and space complexity in the docstrings.

Code:



```python
def merge_sort(arr):
    """Sorts a list in ascending order using the merge sort algorithm.
    Time Complexity: O(n log n) - where n is the number of elements in the list.
    Space Complexity: O(n) - due to the temporary arrays used for merging.
    Parameters:
    arr (list): The list to be sorted.
    Returns:
    list: A new sorted list in ascending order."""
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left_half = merge_sort(arr[:mid])
    right_half = merge_sort(arr[mid:])
    return merge(left_half, right_half)
def merge(left, right):
    """Merges two sorted lists into a single sorted list.
    Parameters:
    left (list): The first sorted list.
    right (list): The second sorted list.
    Returns:
    list: A merged sorted list."""
    merged = []
    left_index = right_index = 0
    while left_index < len(left) and right_index < len(right):
        if left[left_index] < right[right_index]:
            merged.append(left[left_index])
            left_index += 1
        else:
            merged.append(right[right_index])
            right_index += 1
    merged.extend(left[left_index:])
    merged.extend(right[right_index:])
    return merged
# Test cases
if __name__ == "__main__":
    test_cases = [
        [38, 27, 43, 3, 9, 82, 10],
        [5, 2, 9, 1, 5, 6],
        [12, 11, 13, 5, 6, 7],
        [],
        [1],
        [2, 1]]
    for i, test in enumerate(test_cases):
        sorted_list = merge_sort(test)
        print(f"Test Case {i + 1}: {sorted_list}")
```

Output:



```
Test Case 1: [3, 9, 10, 27, 38, 43, 82]
Test Case 2: [1, 2, 5, 5, 6, 9]
Test Case 3: [5, 6, 7, 11, 12, 13]
Test Case 4: []
Test Case 5: [1]
Test Case 6: [1, 2]
```

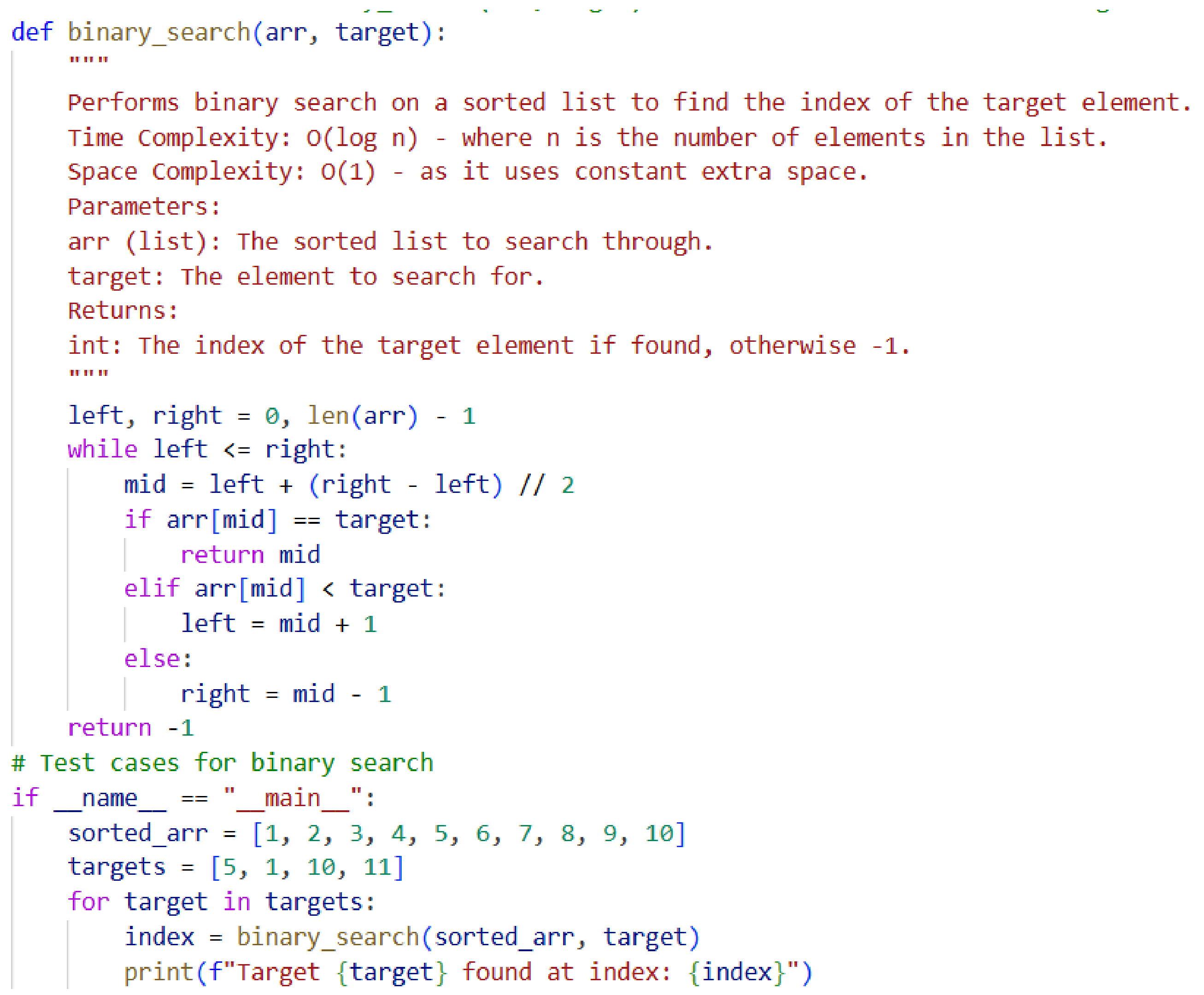
Explanation:

The merge_sort function implements the merge sort algorithm, which is a divide-and-conquer sorting algorithm. It recursively divides the input list into halves until it reaches lists of length 1 or 0 (which are inherently sorted). Then, it merges those sorted halves back together in the correct order using the merge function. The time complexity of O (n log n) arises because each division takes O (log n) and merging takes O(n). The space complexity is O(n) due to the temporary arrays used during the merging process.

Task 2: Searching – Binary Search with AI Optimization

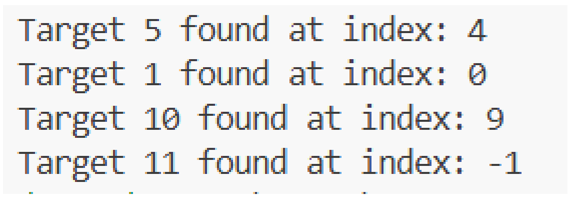
Prompt: Create a function binary_search (arr, target) it returns the index of the target or -1 if not found and include docstring explaining time complexities and test with various inputs.

Code:



```python
def binary_search(arr, target):
    """
    Performs binary search on a sorted list to find the index of the target element.
    Time Complexity: O(log n) - where n is the number of elements in the list.
    Space Complexity: O(1) - as it uses constant extra space.
    Parameters:
    arr (list): The sorted list to search through.
    target: The element to search for.
    Returns:
    int: The index of the target element if found, otherwise -1.
    """
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = left + (right - left) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1
# Test cases for binary search
if __name__ == "__main__":
    sorted_arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    targets = [5, 1, 10, 11]
    for target in targets:
        index = binary_search(sorted_arr, target)
        print(f"Target {target} found at index: {index}")
```

Output:



```
Target 5 found at index: 4
Target 1 found at index: 0
Target 10 found at index: 9
Target 11 found at index: -1
```

Explanation:

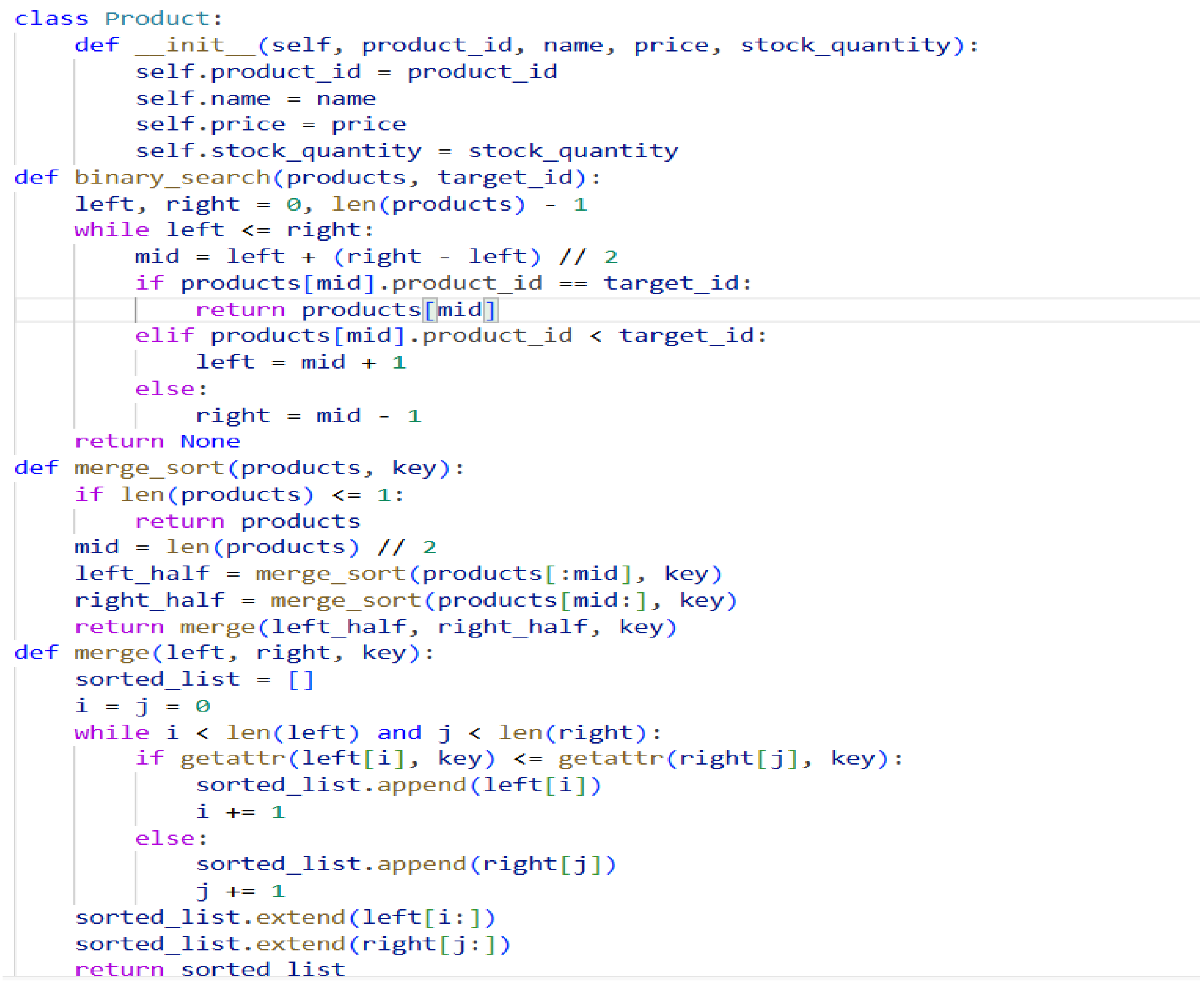
- The binary_search function takes a sorted list and a target element as input.
- It uses two pointers, left and right, to keep track of the current search range.
- The function calculates the middle index and compares the middle element with the target.
- If they match, it returns the index.
- If the middle element is less than the target, it narrows the search to the right half.
- If the middle element is greater than the target, it narrows to the left half.
- If the target is not found after exhausting the search range, it returns -1.

Task 3: Real-Time Application – Inventory Management System

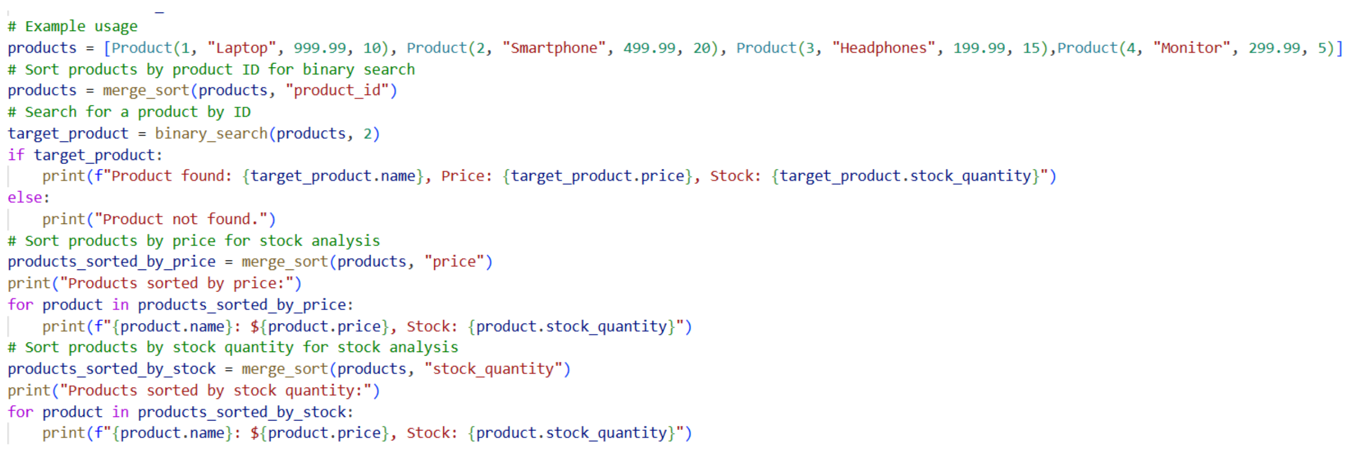
Prompt: A retail store's inventory system contains thousands of products, each with attributes like product ID, name, price, and stock quantity. Store staff need to:

1. Quickly search for a product by ID or name: Use binary search.
2. Sort products by price or quantity for stock analysis: Use merge sort.

Code:



```python
class Product:
    def __init__(self, product_id, name, price, stock_quantity):
        self.product_id = product_id
        self.name = name
        self.price = price
        self.stock_quantity = stock_quantity
def binary_search(products, target_id):
    left, right = 0, len(products) - 1
    while left <= right:
        mid = left + (right - left) // 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 merge_sort(products, key):
    if len(products) <= 1:
        return products
    mid = len(products) // 2
    left_half = merge_sort(products[:mid], key)
    right_half = merge_sort(products[mid:], key)
    return merge(left_half, right_half, key)
def merge(left, right, key):
    sorted_list = []
    i = j = 0
    while i < len(left) and j < len(right):
        if getattr(left[i], key) <= getattr(right[j], key):
            sorted_list.append(left[i])
            i += 1
        else:
            sorted_list.append(right[j])
            j += 1
    sorted_list.extend(left[i:])
    sorted_list.extend(right[j:])
    return sorted_list
```



```python
# Example usage
products = [Product(1, "Laptop", 999.99, 10), Product(2, "Smartphone", 499.99, 20), Product(3, "Headphones", 199.99, 15),Product(4, "Monitor", 299.99, 5)]
# Sort products by product ID for binary search
products = merge_sort(products, "product_id")
# Search for a product by ID
target_product = binary_search(products, 2)
if target_product:
    print(f"Product found: {target_product.name}, Price: {target_product.price}, Stock: {target_product.stock_quantity}")
else:
    print("Product not found.")
# Sort products by price for stock analysis
products_sorted_by_price = merge_sort(products, "price")
print("Products sorted by price:")
for product in products_sorted_by_price:
    print(f"{product.name}: ${product.price}, Stock: {product.stock_quantity}")
# Sort products by stock quantity for stock analysis
products_sorted_by_stock = merge_sort(products, "stock_quantity")
print("Products sorted by stock quantity:")
for product in products_sorted_by_stock:
    print(f"{product.name}: ${product.price}, Stock: {product.stock_quantity}")
```

Output:

```
Product found: Smartphone, Price: 499.99, Stock: 20
Products sorted by price:
Headphones: $199.99, Stock: 15
Monitor: $299.99, Stock: 5
Smartphone: $499.99, Stock: 20
Laptop: $999.99, Stock: 10
Products sorted by stock quantity:
Monitor: $299.99, Stock: 5
Laptop: $999.99, Stock: 10
Headphones: $199.99, Stock: 15
Smartphone: $499.99, Stock: 20
```

Justification:

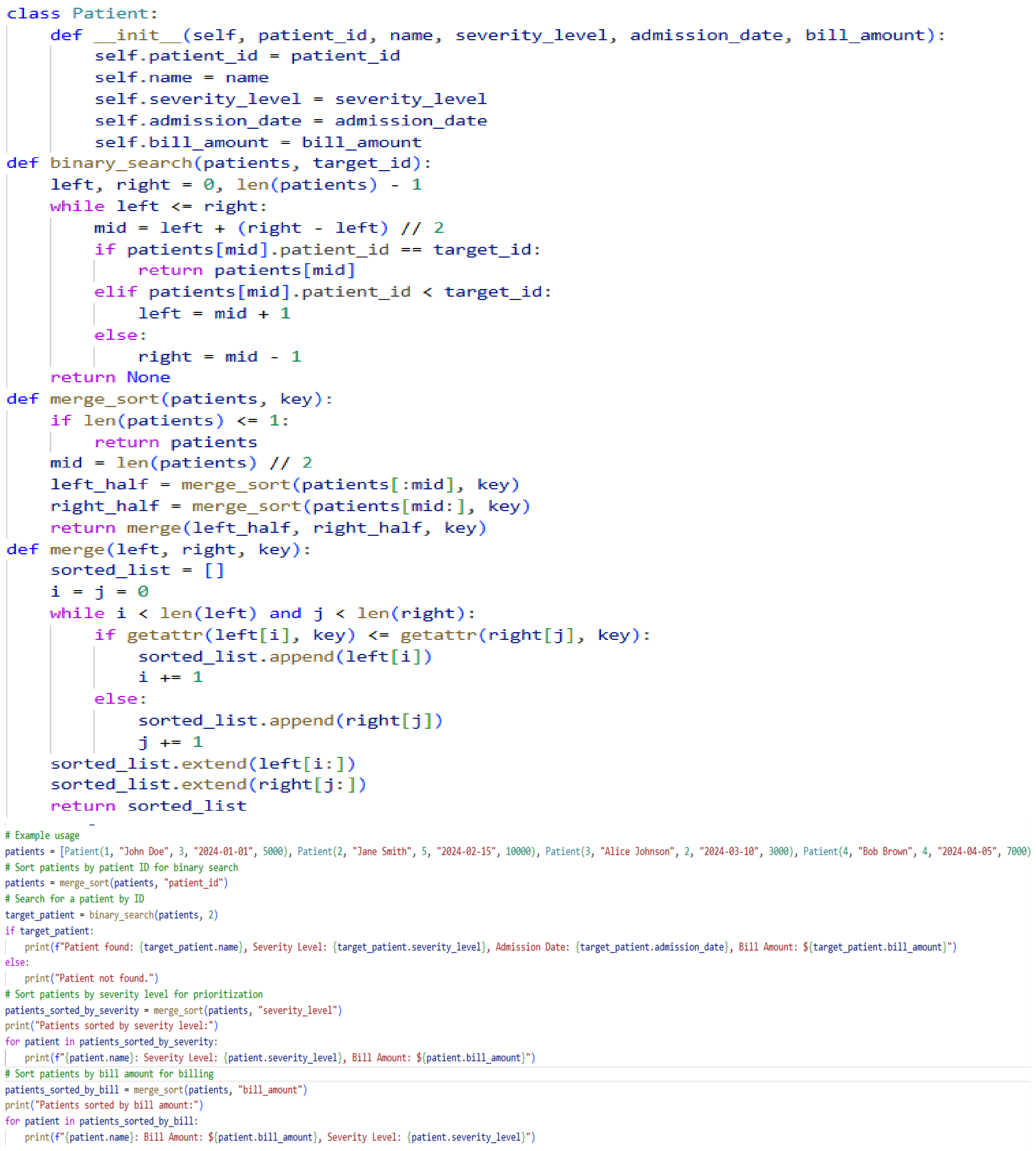
- Binary Search is used to quickly find a product by its unique product ID in O (log n) time after sorting.
- Merge Sort is used to efficiently sort products by price or stock quantity in O (n log n) time, which is suitable for large inventories and ensures stable ordering.

Task 4: Smart Hospital Patient Management System

Prompt: 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:

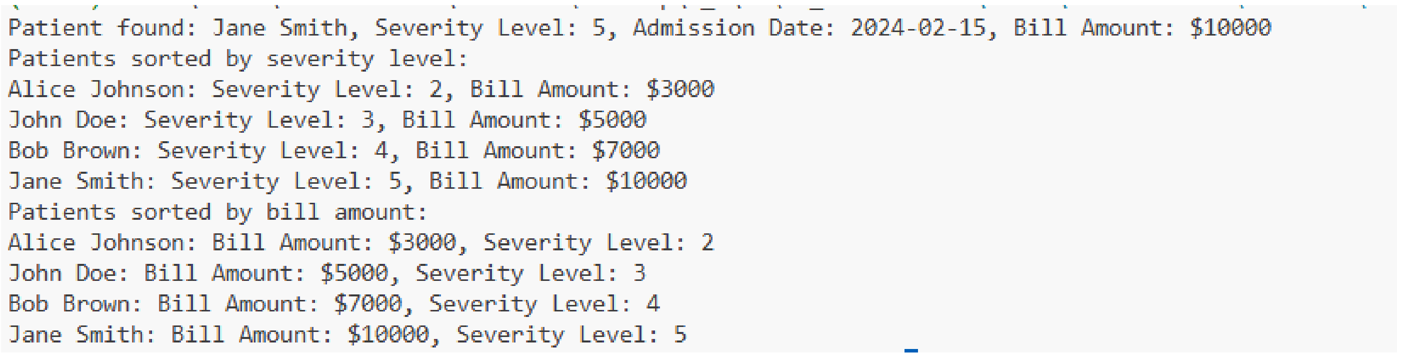
1. Quickly search patient records using patient ID or name: Use binary search.
2. Sort patients based on severity level or bill amount for prioritization and billing: Use merge sort.

Code:



```
class Patient:
    def __init__(self, patient_id, name, severity_level, admission_date, bill_amount):
        self.patient_id = patient_id
        self.name = name
        self.severity_level = severity_level
        self.admission_date = admission_date
        self.bill_amount = bill_amount
def binary_search(patients, target_id):
    left, right = 0, len(patients) - 1
    while left <= right:
        mid = left + (right - left) // 2
        if patients[mid].patient_id == target_id:
            return patients[mid]
        elif patients[mid].patient_id < target_id:
            left = mid + 1
        else:
            right = mid - 1
    return None
def merge_sort(patients, key):
    if len(patients) <= 1:
        return patients
    mid = len(patients) // 2
    left_half = merge_sort(patients[:mid], key)
    right_half = merge_sort(patients[mid:], key)
    return merge(left_half, right_half, key)
def merge(left, right, key):
    sorted_list = []
    i = j = 0
    while i < len(left) and j < len(right):
        if getattr(left[i], key) <= getattr(right[j], key):
            sorted_list.append(left[i])
            i += 1
        else:
            sorted_list.append(right[j])
            j += 1
    sorted_list.extend(left[i:])
    sorted_list.extend(right[j:])
    return sorted_list
# Example usage
patients = [Patient(1, "John Doe", 3, "2024-01-01", 5000), Patient(2, "Jane Smith", 5, "2024-02-15", 10000), Patient(3, "Alice Johnson", 2, "2024-03-10", 3000), Patient(4, "Bob Brown", 4, "2024-04-05", 7000)
# Sort patients by patient ID for binary search
patients = merge_sort(patients, "patient_id")
# Search for a patient by ID
target_patient = binary_search(patients, 2)
if target_patient:
    print(f"Patient found: {target_patient.name}, Severity Level: {target_patient.severity_level}, Admission Date: {target_patient.admission_date}, Bill Amount: ${target_patient.bill_amount}")
else:
    print("Patient not found.")
# Sort patients by severity level for prioritization
patients_sorted_by_severity = merge_sort(patients, "severity_level")
print("Patients sorted by severity level:")
for patient in patients_sorted_by_severity:
    print(f"{patient.name}: Severity Level: {patient.severity_level}, Bill Amount: ${patient.bill_amount}")
# Sort patients by bill amount for billing
patients_sorted_by_bill = merge_sort(patients, "bill_amount")
print("Patients sorted by bill amount:")
for patient in patients_sorted_by_bill:
    print(f"{patient.name}: Bill Amount: ${patient.bill_amount}, Severity Level: {patient.severity_level}")
```

Output:



```
Patient found: Jane Smith, Severity Level: 5, Admission Date: 2024-02-15, Bill Amount: $10000
Patients sorted by severity level:
Alice Johnson: Severity Level: 2, Bill Amount: $3000
John Doe: Severity Level: 3, Bill Amount: $5000
Bob Brown: Severity Level: 4, Bill Amount: $7000
Jane Smith: Severity Level: 5, Bill Amount: $10000
Patients sorted by bill amount:
Alice Johnson: Bill Amount: $3000, Severity Level: 2
John Doe: Bill Amount: $5000, Severity Level: 3
Bob Brown: Bill Amount: $7000, Severity Level: 4
Jane Smith: Bill Amount: $10000, Severity Level: 5
```

Explanation:

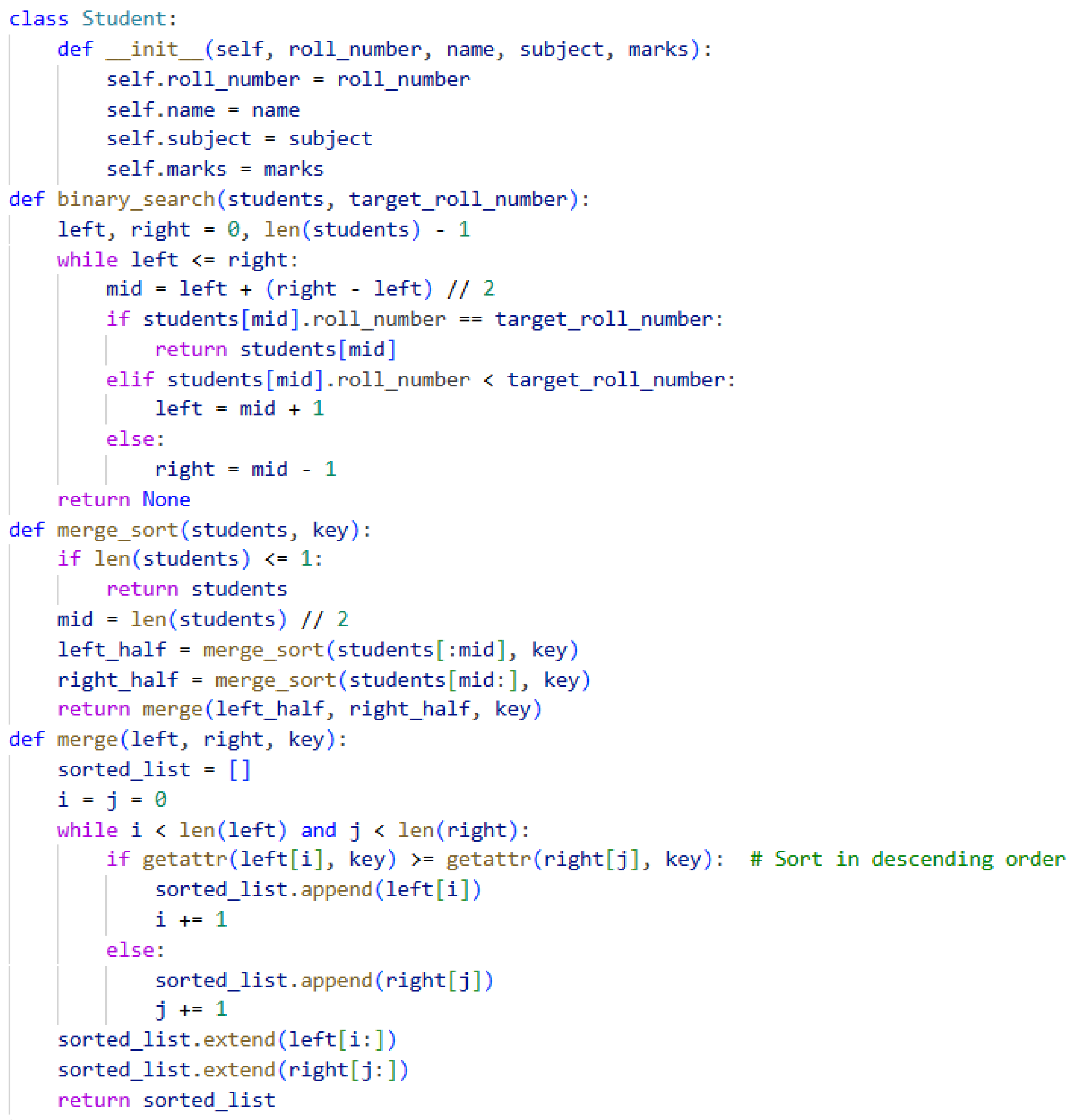
- Binary Search is used to quickly find patient records by ID in O (log n) time since data is sorted.
- Merge Sort is used to efficiently sort patients by severity or bill amount in O (n log n) time while maintaining stability.

Task 5: University Examination Result Processing System

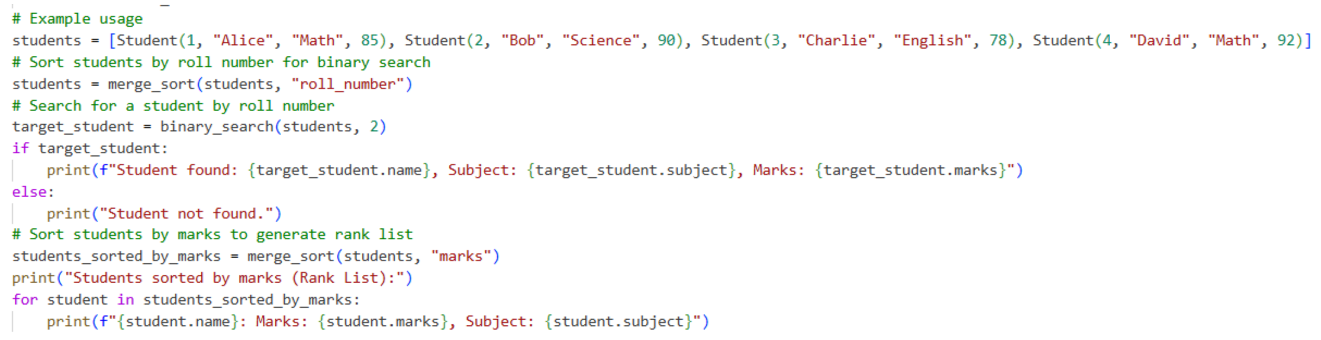
Prompt: A university processes examination results for thousands of students containing roll number, name, subject, and marks. The system must:

1. Search student results using roll number: Use binary search.
2. Sort students based on marks to generate rank lists: Use merge sort.

Code:



```python
class Student:
    def __init__(self, roll_number, name, subject, marks):
        self.roll_number = roll_number
        self.name = name
        self.subject = subject
        self.marks = marks
def binary_search(students, target_roll_number):
    left, right = 0, len(students) - 1
    while left <= right:
        mid = left + (right - left) // 2
        if students[mid].roll_number == target_roll_number:
            return students[mid]
        elif students[mid].roll_number < target_roll_number:
            left = mid + 1
        else:
            right = mid - 1
    return None
def merge_sort(students, key):
    if len(students) <= 1:
        return students
    mid = len(students) // 2
    left_half = merge_sort(students[:mid], key)
    right_half = merge_sort(students[mid:], key)
    return merge(left_half, right_half, key)
def merge(left, right, key):
    sorted_list = []
    i = j = 0
    while i < len(left) and j < len(right):
        if getattr(left[i], key) >= getattr(right[j], key):  # Sort in descending order
            sorted_list.append(left[i])
            i += 1
        else:
            sorted_list.append(right[j])
            j += 1
    sorted_list.extend(left[i:])
    sorted_list.extend(right[j:])
    return sorted_list
```



```python
# Example usage
students = [Student(1, "Alice", "Math", 85), Student(2, "Bob", "Science", 90), Student(3, "Charlie", "English", 78), Student(4, "David", "Math", 92)]
# Sort students by roll number for binary search
students = merge_sort(students, "roll_number")
# Search for a student by roll number
target_student = binary_search(students, 2)
if target_student:
    print(f"Student found: {target_student.name}, Subject: {target_student.subject}, Marks: {target_student.marks}")
else:
    print("Student not found.")
# Sort students by marks to generate rank list
students_sorted_by_marks = merge_sort(students, "marks")
print("Students sorted by marks (Rank List):")
for student in students_sorted_by_marks:
    print(f"{student.name}: Marks: {student.marks}, Subject: {student.subject}")
```

Output:

```
Student not found.
Students sorted by marks (Rank List):
David: Marks: 92, Subject: Math
Bob: Marks: 90, Subject: Science
Alice: Marks: 85, Subject: Math
Charlie: Marks: 78, Subject: English
```

Explanation:

- Binary Search allows fast retrieval of student results using roll number in O (log n) time.
- Merge Sort is used to generate rank lists by sorting marks efficiently in O (n log n) time.

Task 6: Online Food Delivery Platform

Prompt: A streaming service maintains movie records with movie ID, title, genre, rating, and release year. The platform needs to:

1. Search movies by movie ID: Use binary search to search the movies.
2. Sort movies based on rating or release year: Use merge sort to sort movies.

Generate the python program for above scenario.

Code:

```python
class Order:
    def __init__(self, order_id, restaurant_name, delivery_time, price, order_status):
        self.order_id = order_id
        self.restaurant_name = restaurant_name
        self.delivery_time = delivery_time
        self.price = price
        self.order_status = order_status
def binary_search(orders, target_order_id):
    left, right = 0, len(orders) - 1
    while left <= right:
        mid = left + (right - left) // 2
        if orders[mid].order_id == target_order_id:
            return orders[mid]
        elif orders[mid].order_id < target_order_id:
            left = mid + 1
        else:
            right = mid - 1
    return None
def merge_sort(orders, key):
    if len(orders) <= 1:
        return orders
    mid = len(orders) // 2
    left_half = merge_sort(orders[:mid], key)
    right_half = merge_sort(orders[mid:], key)
    return merge(left_half, right_half, key)
def merge(left, right, key):
    sorted_list = []
    i = j = 0
    while i < len(left) and j < len(right):
        if getattr(left[i], key) <= getattr(right[j], key):
            sorted_list.append(left[i])
            i += 1
        else:
            sorted_list.append(right[j])
            j += 1
    sorted_list.extend(left[i:])
    sorted_list.extend(right[j:])
    return sorted_list
```

```python
# Example usage
orders = [Order(1, "Pizza Place", "2024-06-01 18:30", 20.99, "Delivered"), Order(2, "Sushi Spot", "2024-06-01 19:00", 35.50, "In Progress"), Order(3, "Burger Joint", "2024-06-01 18:45", 15.75, "Delivered"),
          Order(4, "Pasta House", "2024-06-01 19:15", 25.00, "Pending")]
# Sort orders by order ID for binary search
orders = merge_sort(orders, "order_id")
# Search for an order by order ID
target_order = binary_search(orders, 2)
if target_order:
    print(f"Order found: {target_order.restaurant_name}, Delivery Time: {target_order.delivery_time}, Price: ${target_order.price}, Status: {target_order.order_status}")
else:
    print("Order not found.")
# Sort orders by delivery time
orders_sorted_by_delivery_time = merge_sort(orders, "delivery_time")
print("Orders sorted by delivery time:")
for order in orders_sorted_by_delivery_time:
    print(f"{order.restaurant_name}: Delivery Time: {order.delivery_time}, Price: ${order.price}, Status: {order.order_status}")
# Sort orders by price
orders_sorted_by_price = merge_sort(orders, "price")
print("Orders sorted by price:")
for order in orders_sorted_by_price:
    print(f"{order.restaurant_name}: Price: ${order.price}, Delivery Time: {order.delivery_time}, Status: {order.order_status}")
```

Output:

```
Order found: Sushi Spot, Delivery Time: 2024-06-01 19:00, Price: $35.5, Status: In Progress
Orders sorted by delivery time:
Pizza Place: Delivery Time: 2024-06-01 18:30, Price: $20.99, Status: Delivered
Burger Joint: Delivery Time: 2024-06-01 18:45, Price: $15.75, Status: Delivered
Sushi Spot: Delivery Time: 2024-06-01 19:00, Price: $35.5, Status: In Progress
Pasta House: Delivery Time: 2024-06-01 19:15, Price: $25.0, Status: Pending
Orders sorted by price:
Burger Joint: Price: $15.75, Delivery Time: 2024-06-01 18:45, Status: Delivered
Pizza Place: Price: $20.99, Delivery Time: 2024-06-01 18:30, Status: Delivered
Pasta House: Price: $25.0, Delivery Time: 2024-06-01 19:15, Status: Pending
Sushi Spot: Price: $35.5, Delivery Time: 2024-06-01 19:00, Status: In Progress
```

Explanation:

- Binary Search helps quickly locate an order-by-order ID in O (log n) time.
- Merge Sort efficiently sorts orders by delivery time or price in O (n log n) time for better management.