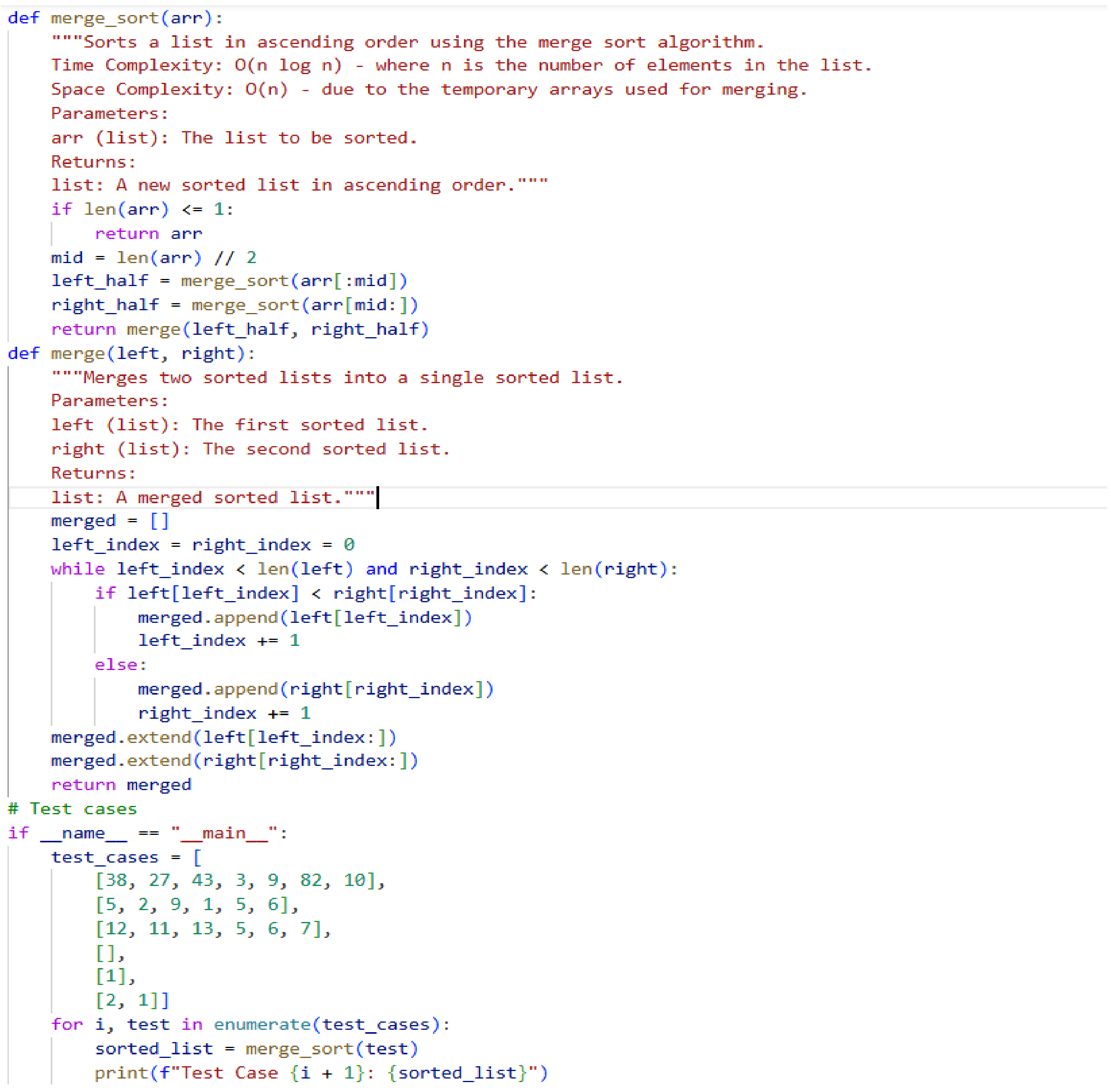
Lab Assignment 12.5

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 : 20-02-2026

Task 1: Sorting – Merge Sort Implementation

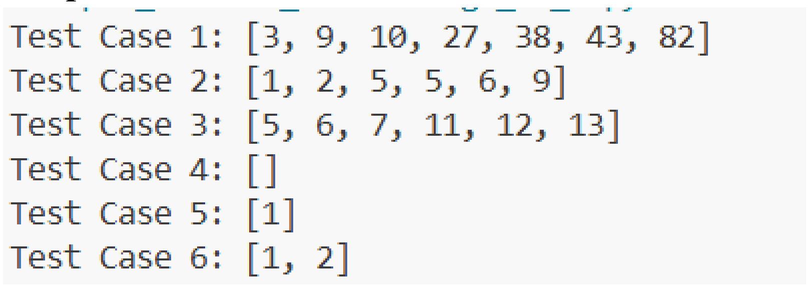
Prompt: Generate a python program that creates a function merge_sort(arr) that sorts a list in ascending order and include time complexity and space complexity in the docstrings.

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]
```

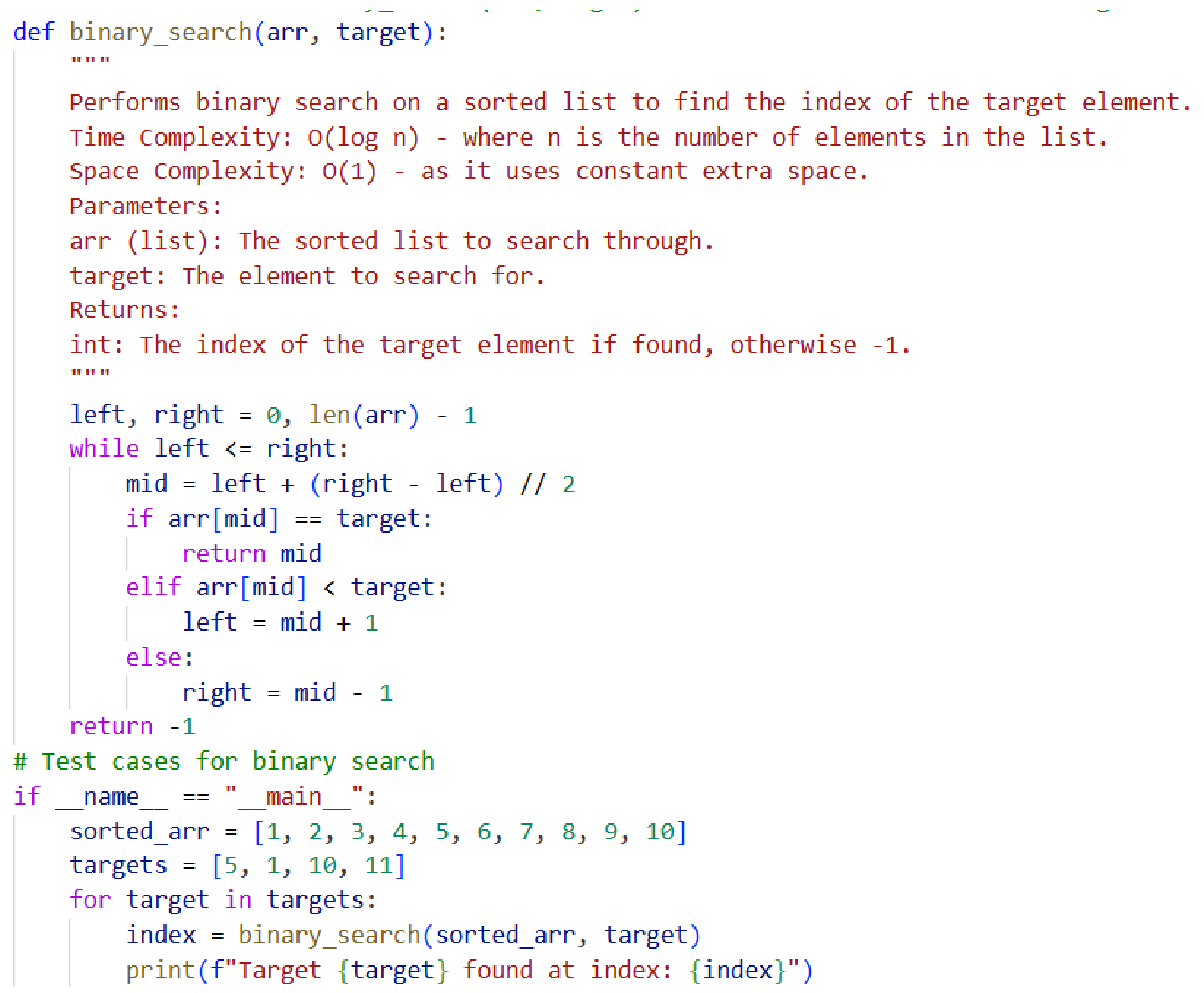
Explanation:

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: Smart Healthcare Appointment Scheduling System

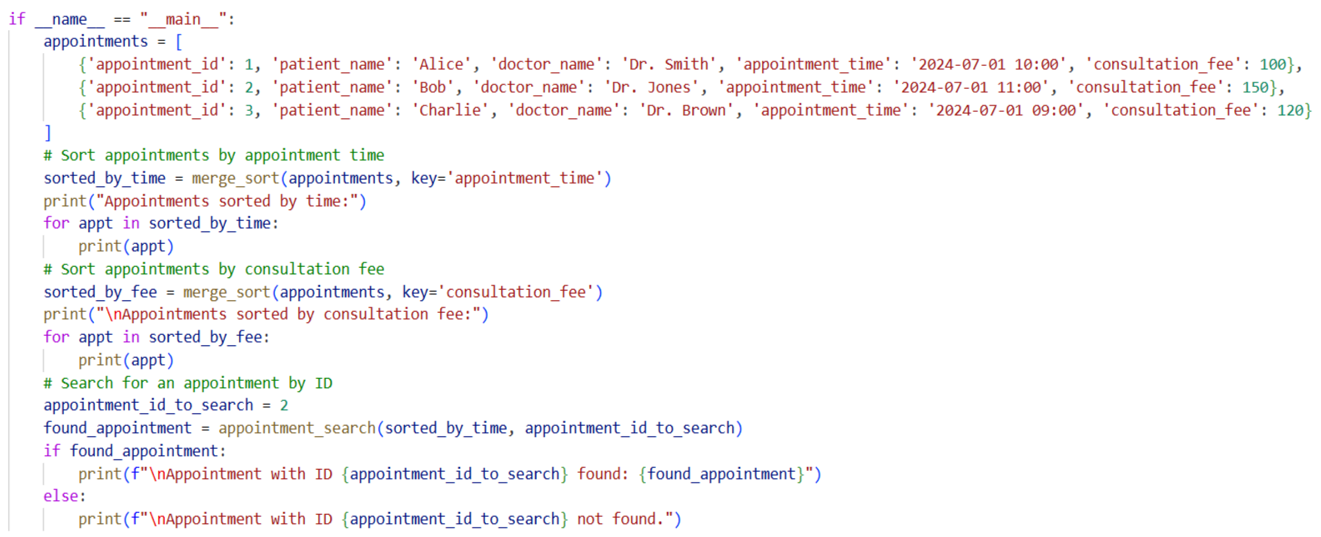
Prompt: A healthcare platform maintains appointment records containing appointment ID, patient name, doctor name, appointment time, and consultation fee. The system needs to:

1. Search appointments using appointment ID: To search the appointments use the binary search
2. Sort appointments based on time or consultation fee: To sort the time or consultation fee use merge sort

Generate the python program for the above scenario.

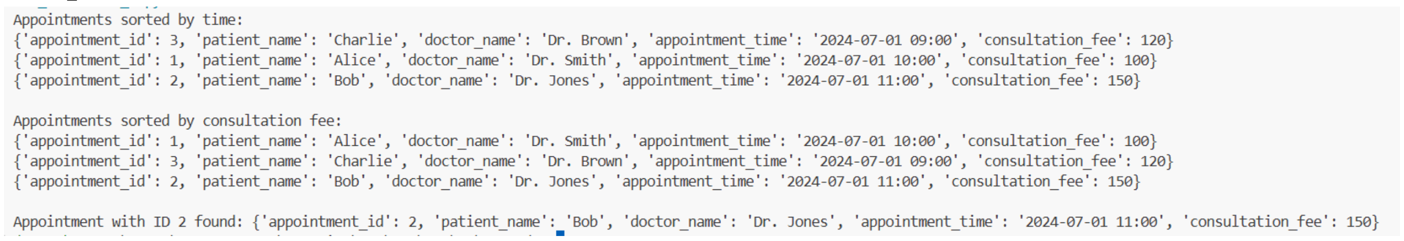
Code:

```python
def appointment_search(appointments, appointment_id):
    left, right = 0, len(appointments) - 1
    while left <= right:
        mid = left + (right - left) // 2
        if appointments[mid]['appointment_id'] == appointment_id:
            return appointments[mid]
        elif appointments[mid]['appointment_id'] < appointment_id:
            left = mid + 1
        else:
            right = mid - 1
    return None
def merge_sort(arr, key):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left_half = merge_sort(arr[:mid], key)
    right_half = merge_sort(arr[mid:], key)
    return merge(left_half, right_half, key)
def merge(left, right, key):
    merged = []
    left_index = right_index = 0
    while left_index < len(left) and right_index < len(right):
        if left[left_index][key] < right[right_index][key]:
            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
```



```python
if __name__ == "__main__":
    appointments = [
        {'appointment_id': 1, 'patient_name': 'Alice', 'doctor_name': 'Dr. Smith', 'appointment_time': '2024-07-01 10:00', 'consultation_fee': 100},
        {'appointment_id': 2, 'patient_name': 'Bob', 'doctor_name': 'Dr. Jones', 'appointment_time': '2024-07-01 11:00', 'consultation_fee': 150},
        {'appointment_id': 3, 'patient_name': 'Charlie', 'doctor_name': 'Dr. Brown', 'appointment_time': '2024-07-01 09:00', 'consultation_fee': 120}
    ]
    # Sort appointments by appointment time
    sorted_by_time = merge_sort(appointments, key='appointment_time')
    print("Appointments sorted by time:")
    for appt in sorted_by_time:
        print(appt)
    # Sort appointments by consultation fee
    sorted_by_fee = merge_sort(appointments, key='consultation_fee')
    print("\nAppointments sorted by consultation fee:")
    for appt in sorted_by_fee:
        print(appt)
    # Search for an appointment by ID
    appointment_id_to_search = 2
    found_appointment = appointment_search(sorted_by_time, appointment_id_to_search)
    if found_appointment:
        print(f"\nAppointment with ID {appointment_id_to_search} found: {found_appointment}")
    else:
        print(f"\nAppointment with ID {appointment_id_to_search} not found.")
```

Output:



```
Appointments sorted by time:
{'appointment_id': 3, 'patient_name': 'Charlie', 'doctor_name': 'Dr. Brown', 'appointment_time': '2024-07-01 09:00', 'consultation_fee': 120}
{'appointment_id': 1, 'patient_name': 'Alice', 'doctor_name': 'Dr. Smith', 'appointment_time': '2024-07-01 10:00', 'consultation_fee': 100}
{'appointment_id': 2, 'patient_name': 'Bob', 'doctor_name': 'Dr. Jones', 'appointment_time': '2024-07-01 11:00', 'consultation_fee': 150}

Appointments sorted by consultation fee:
{'appointment_id': 1, 'patient_name': 'Alice', 'doctor_name': 'Dr. Smith', 'appointment_time': '2024-07-01 10:00', 'consultation_fee': 100}
{'appointment_id': 3, 'patient_name': 'Charlie', 'doctor_name': 'Dr. Brown', 'appointment_time': '2024-07-01 09:00', 'consultation_fee': 120}
{'appointment_id': 2, 'patient_name': 'Bob', 'doctor_name': 'Dr. Jones', 'appointment_time': '2024-07-01 11:00', 'consultation_fee': 150}

Appointment with ID 2 found: {'appointment_id': 2, 'patient_name': 'Bob', 'doctor_name': 'Dr. Jones', 'appointment_time': '2024-07-01 11:00', 'consultation_fee': 150}
```

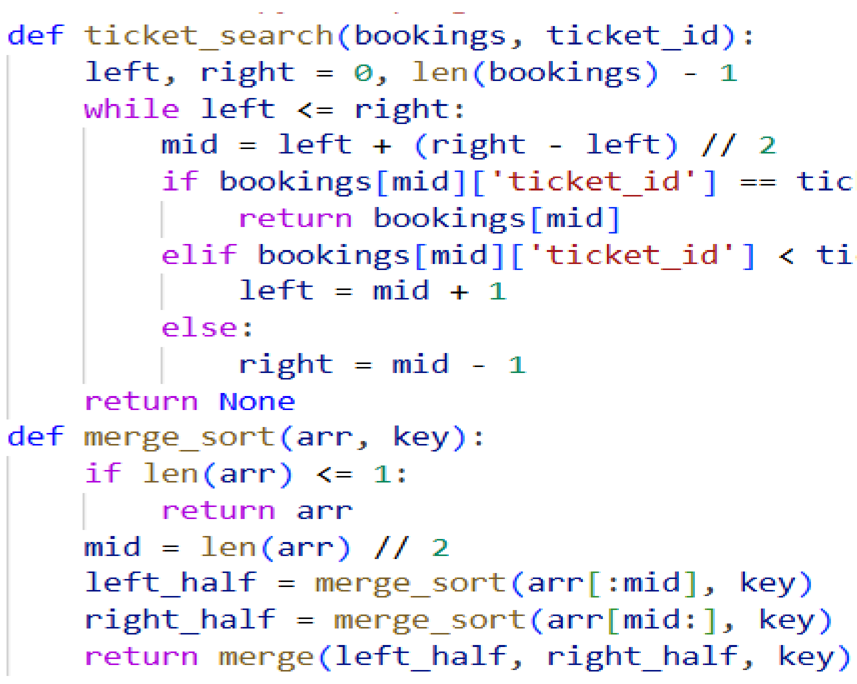
Justification:

Binary Search is used to search appointment records because it is fast and efficient with a time complexity of **O (log n)**. It works well when the data is sorted and is suitable for large healthcare databases.
Merge Sort is used for sorting appointments by time and fee because it has a time complexity of **O (n log n)** and is stable. It performs better than simple sorting algorithms like Bubble Sort for large datasets.

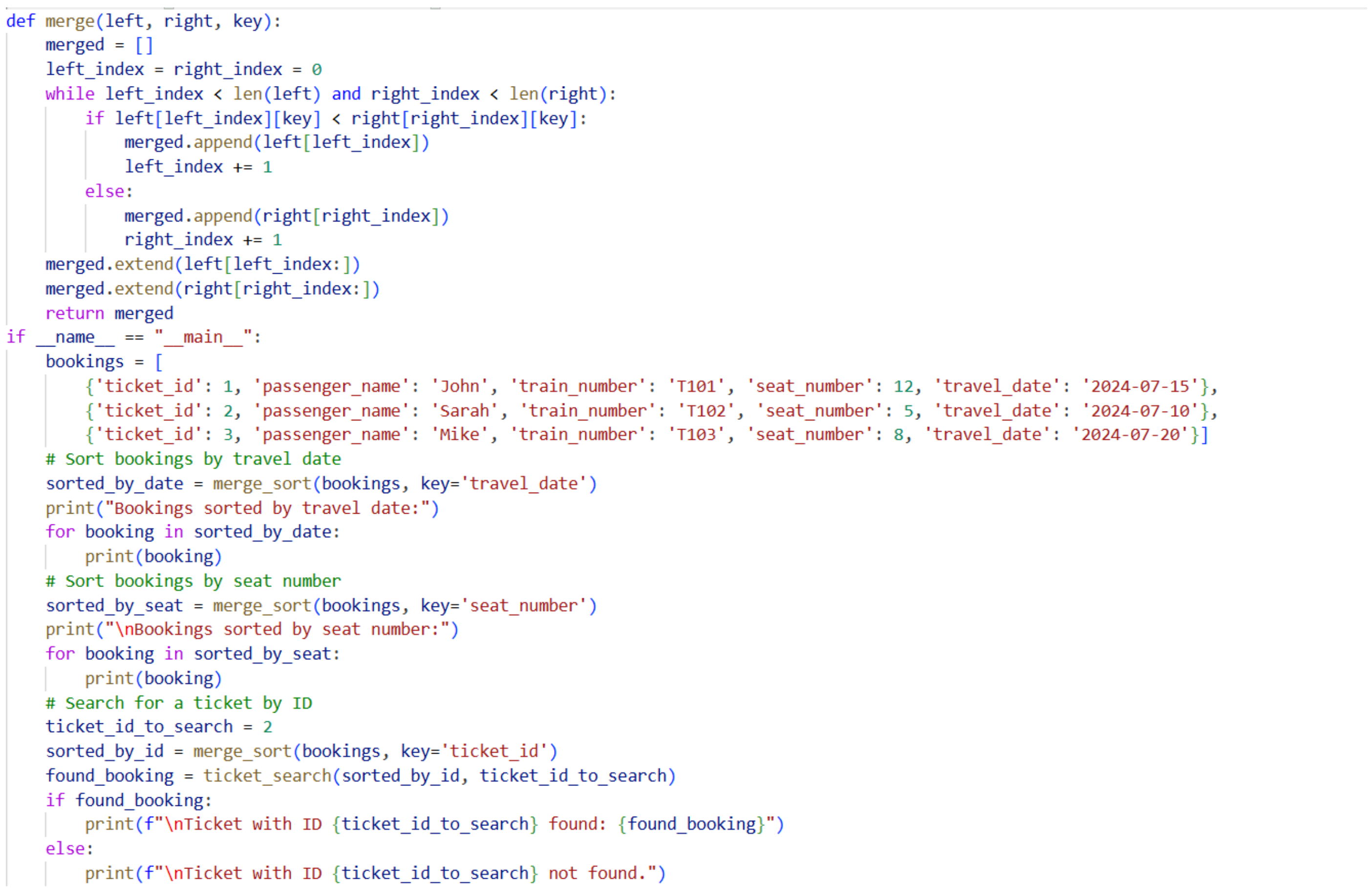
Task 4: Railway Ticket Reservation System

Prompt: A railway reservation system stores booking details containing ticket ID, passenger name, train number, seat number, and travel date. The system needs to:
1. Search tickets using ticket ID: To search tickets use binary search.
2. Sort bookings based on travel date or seat number: To sort use merge sort.
Generate the python program for the above scenario.

Code:

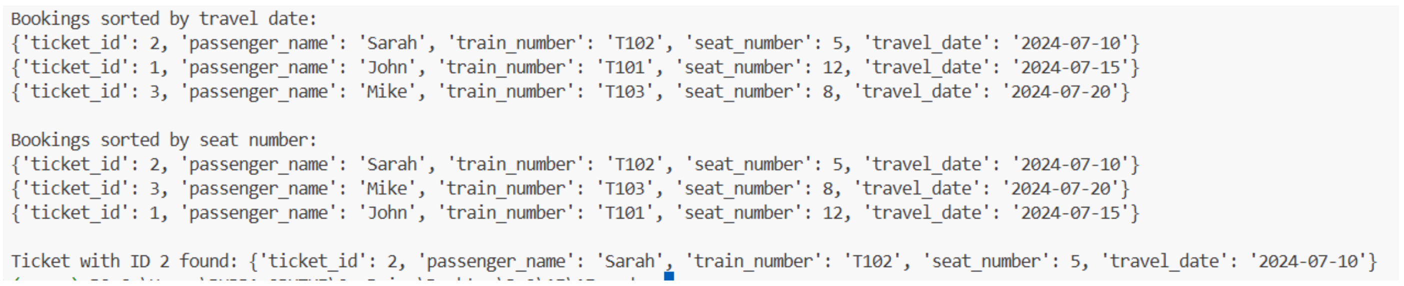


```python
def ticket_search(bookings, ticket_id):
    left, right = 0, len(bookings) - 1
    while left <= right:
        mid = left + (right - left) // 2
        if bookings[mid]['ticket_id'] == ticket_id:
            return bookings[mid]
        elif bookings[mid]['ticket_id'] < ticket_id:
            left = mid + 1
        else:
            right = mid - 1
    return None
def merge_sort(arr, key):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left_half = merge_sort(arr[:mid], key)
    right_half = merge_sort(arr[mid:], key)
    return merge(left_half, right_half, key)
```



```
def merge(left, right, key):
    merged = []
    left_index = right_index = 0
    while left_index < len(left) and right_index < len(right):
        if left[left_index][key] < right[right_index][key]:
            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
if __name__ == "__main__":
    bookings = [
        {'ticket_id': 1, 'passenger_name': 'John', 'train_number': 'T101', 'seat_number': 12, 'travel_date': '2024-07-15'},
        {'ticket_id': 2, 'passenger_name': 'Sarah', 'train_number': 'T102', 'seat_number': 5, 'travel_date': '2024-07-10'},
        {'ticket_id': 3, 'passenger_name': 'Mike', 'train_number': 'T103', 'seat_number': 8, 'travel_date': '2024-07-20'}]
    # Sort bookings by travel date
    sorted_by_date = merge_sort(bookings, key='travel_date')
    print("Bookings sorted by travel date:")
    for booking in sorted_by_date:
        print(booking)
    # Sort bookings by seat number
    sorted_by_seat = merge_sort(bookings, key='seat_number')
    print("\nBookings sorted by seat number:")
    for booking in sorted_by_seat:
        print(booking)
    # Search for a ticket by ID
    ticket_id_to_search = 2
    sorted_by_id = merge_sort(bookings, key='ticket_id')
    found_booking = ticket_search(sorted_by_id, ticket_id_to_search)
    if found_booking:
        print(f"\nTicket with ID {ticket_id_to_search} found: {found_booking}")
    else:
        print(f"\nTicket with ID {ticket_id_to_search} not found.")
```

Output:



```
Bookings sorted by travel date:
{'ticket_id': 2, 'passenger_name': 'Sarah', 'train_number': 'T102', 'seat_number': 5, 'travel_date': '2024-07-10'}
{'ticket_id': 1, 'passenger_name': 'John', 'train_number': 'T101', 'seat_number': 12, 'travel_date': '2024-07-15'}
{'ticket_id': 3, 'passenger_name': 'Mike', 'train_number': 'T103', 'seat_number': 8, 'travel_date': '2024-07-20'}

Bookings sorted by seat number:
{'ticket_id': 2, 'passenger_name': 'Sarah', 'train_number': 'T102', 'seat_number': 5, 'travel_date': '2024-07-10'}
{'ticket_id': 3, 'passenger_name': 'Mike', 'train_number': 'T103', 'seat_number': 8, 'travel_date': '2024-07-20'}
{'ticket_id': 1, 'passenger_name': 'John', 'train_number': 'T101', 'seat_number': 12, 'travel_date': '2024-07-15'}

Ticket with ID 2 found: {'ticket_id': 2, 'passenger_name': 'Sarah', 'train_number': 'T102', 'seat_number': 5, 'travel_date': '2024-07-10'}
```

Explanation:

Binary Search (For Searching Ticket ID)

- Ticket ID is unique.
- Time Complexity = O (log n).
- It is faster than Linear Search (O(n)).
- Suitable for large railway databases with many bookings.

Merge Sort (For Sorting by Date or Seat Number)

- Time Complexity = O (n log n).
- Efficient for large number of records.

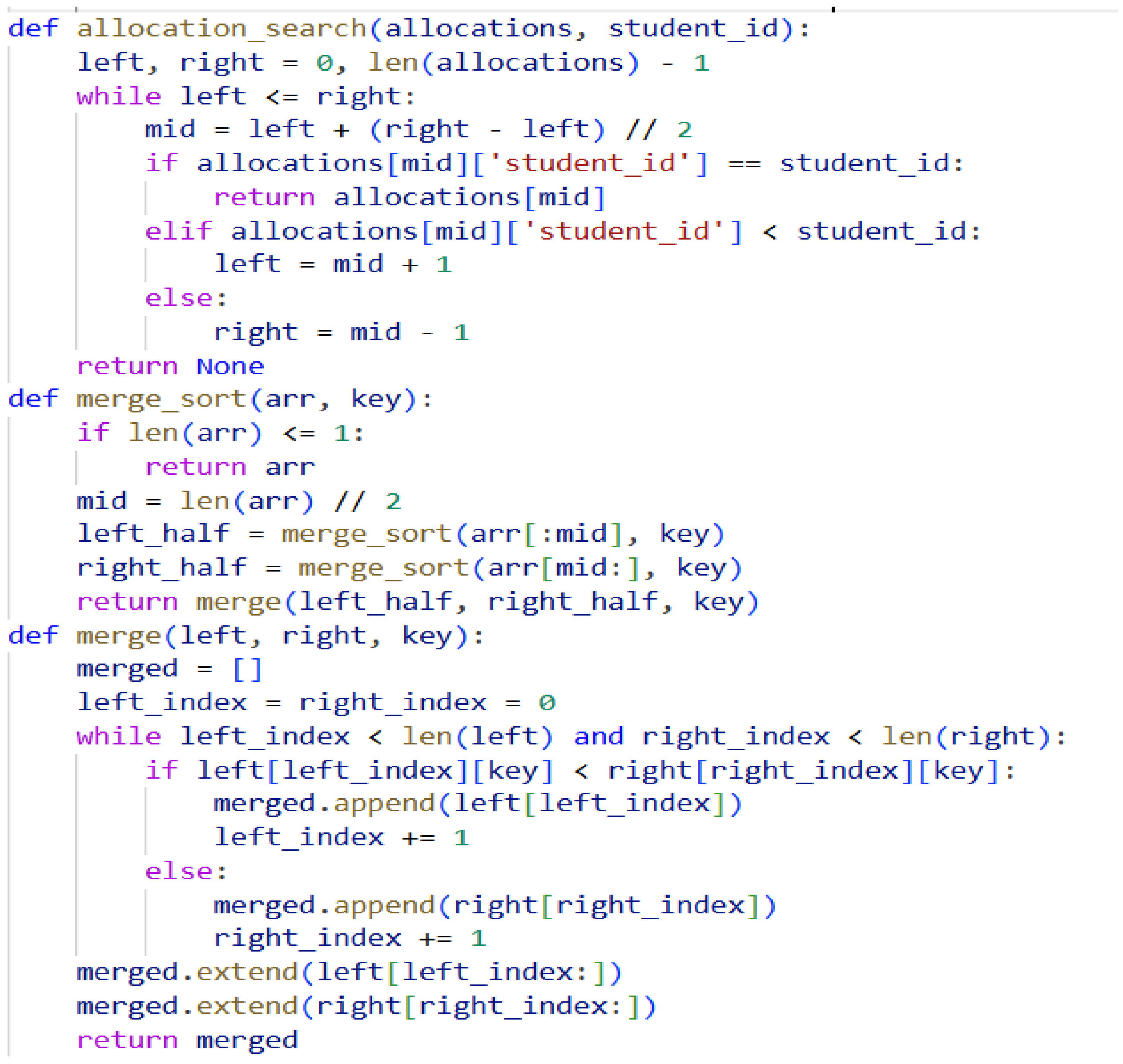
Task 5: Smart Hostel Room Allocation System

Prompt: A hostel management system stores student room allocation detail including student ID, room number, floor, and allocation date. The system needs to:

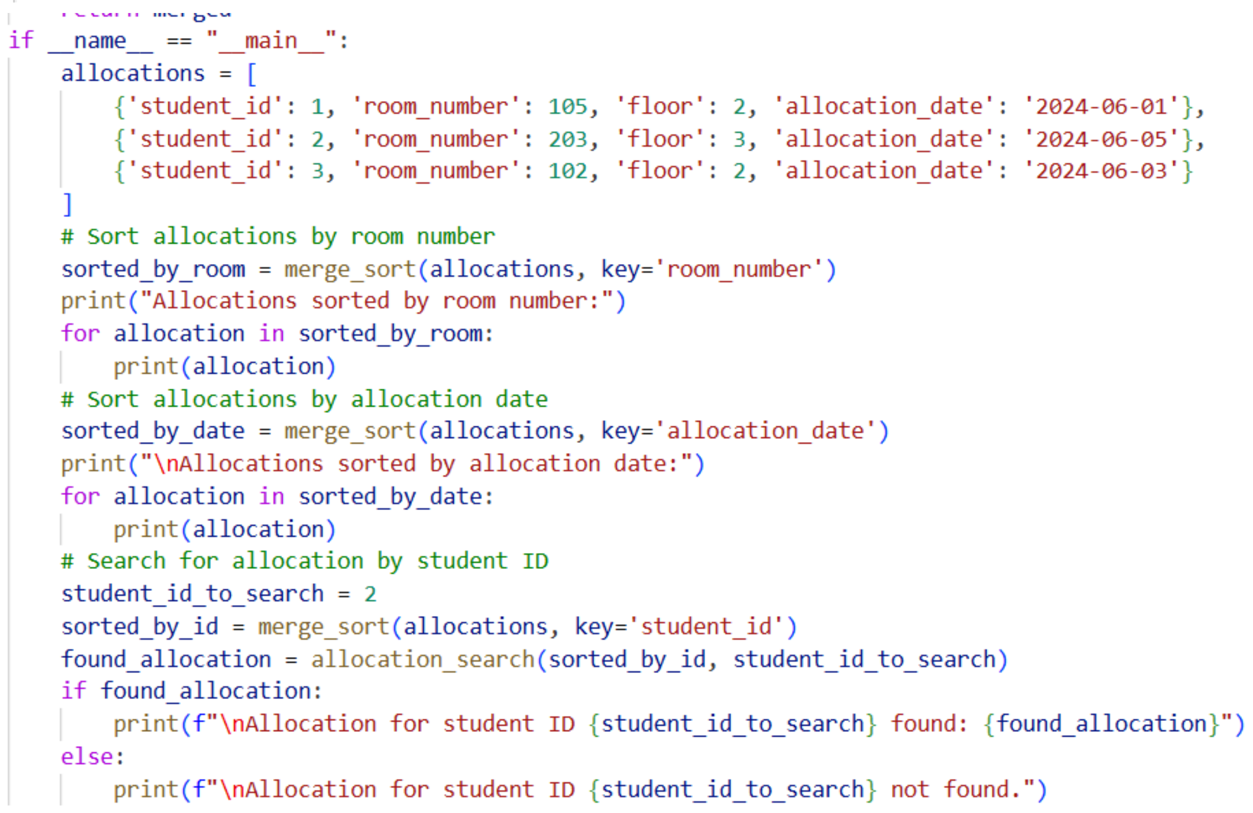
1. Search allocation details using student ID: Use binary search to search allocation details.
2. Sort records based on room number or allocation date: Use merge sort to sort the records.

Generate the code for above scenario.

Code:

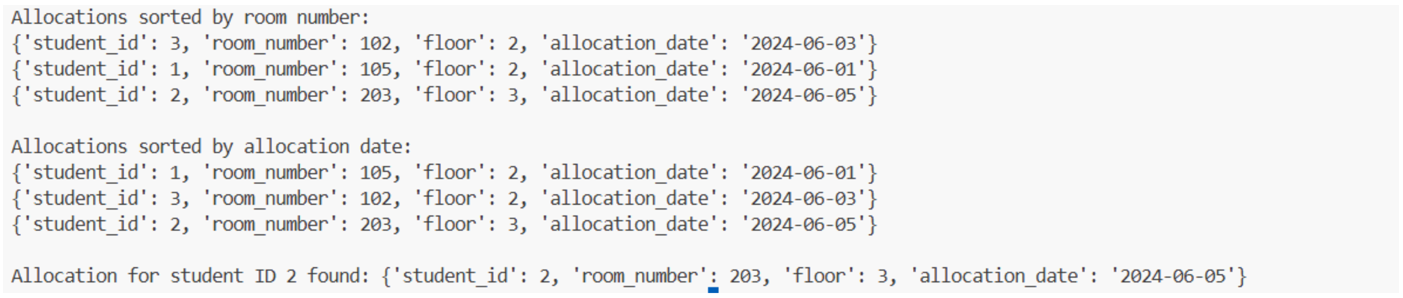


```python
def allocation_search(allocations, student_id):
    left, right = 0, len(allocations) - 1
    while left <= right:
        mid = left + (right - left) // 2
        if allocations[mid]['student_id'] == student_id:
            return allocations[mid]
        elif allocations[mid]['student_id'] < student_id:
            left = mid + 1
        else:
            right = mid - 1
    return None
def merge_sort(arr, key):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left_half = merge_sort(arr[:mid], key)
    right_half = merge_sort(arr[mid:], key)
    return merge(left_half, right_half, key)
def merge(left, right, key):
    merged = []
    left_index = right_index = 0
    while left_index < len(left) and right_index < len(right):
        if left[left_index][key] < right[right_index][key]:
            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
```



```python
if __name__ == "__main__":
    allocations = [
        {'student_id': 1, 'room_number': 105, 'floor': 2, 'allocation_date': '2024-06-01'},
        {'student_id': 2, 'room_number': 203, 'floor': 3, 'allocation_date': '2024-06-05'},
        {'student_id': 3, 'room_number': 102, 'floor': 2, 'allocation_date': '2024-06-03'}
    ]
    # Sort allocations by room number
    sorted_by_room = merge_sort(allocations, key='room_number')
    print("Allocations sorted by room number:")
    for allocation in sorted_by_room:
        print(allocation)
    # Sort allocations by allocation date
    sorted_by_date = merge_sort(allocations, key='allocation_date')
    print("\nAllocations sorted by allocation date:")
    for allocation in sorted_by_date:
        print(allocation)
    # Search for allocation by student ID
    student_id_to_search = 2
    sorted_by_id = merge_sort(allocations, key='student_id')
    found_allocation = allocation_search(sorted_by_id, student_id_to_search)
    if found_allocation:
        print(f"\nAllocation for student ID {student_id_to_search} found: {found_allocation}")
    else:
        print(f"\nAllocation for student ID {student_id_to_search} not found.")
```

Output:



```
Allocations sorted by room number:
{'student_id': 3, 'room_number': 102, 'floor': 2, 'allocation_date': '2024-06-03'}
{'student_id': 1, 'room_number': 105, 'floor': 2, 'allocation_date': '2024-06-01'}
{'student_id': 2, 'room_number': 203, 'floor': 3, 'allocation_date': '2024-06-05'}

Allocations sorted by allocation date:
{'student_id': 1, 'room_number': 105, 'floor': 2, 'allocation_date': '2024-06-01'}
{'student_id': 3, 'room_number': 102, 'floor': 2, 'allocation_date': '2024-06-03'}
{'student_id': 2, 'room_number': 203, 'floor': 3, 'allocation_date': '2024-06-05'}

Allocation for student ID 2 found: {'student_id': 2, 'room_number': 203, 'floor': 3, 'allocation_date': '2024-06-05'}
```

Explanation:

- Binary Search is chosen because it reduces search time significantly compared to linear search.
- Merge Sort is chosen because it guarantees O (n log n) time even in worst case.
- Both are scalable for large hostel databases.

Task 6: Online Movie Streaming Platform

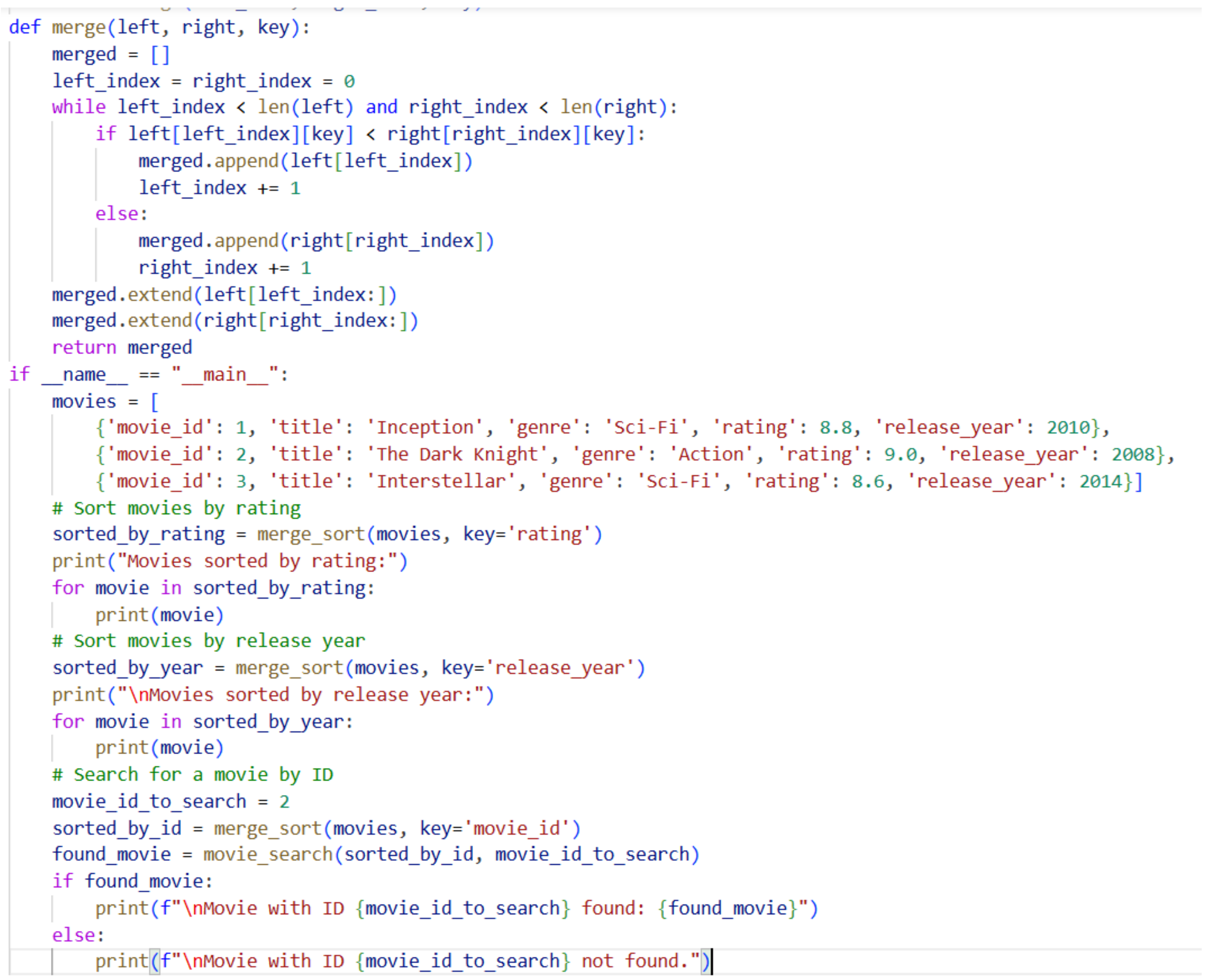
Prompt: A streaming service maintains movie records with movie ID, title, genre, rating, and release year. The platform needs to:

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.

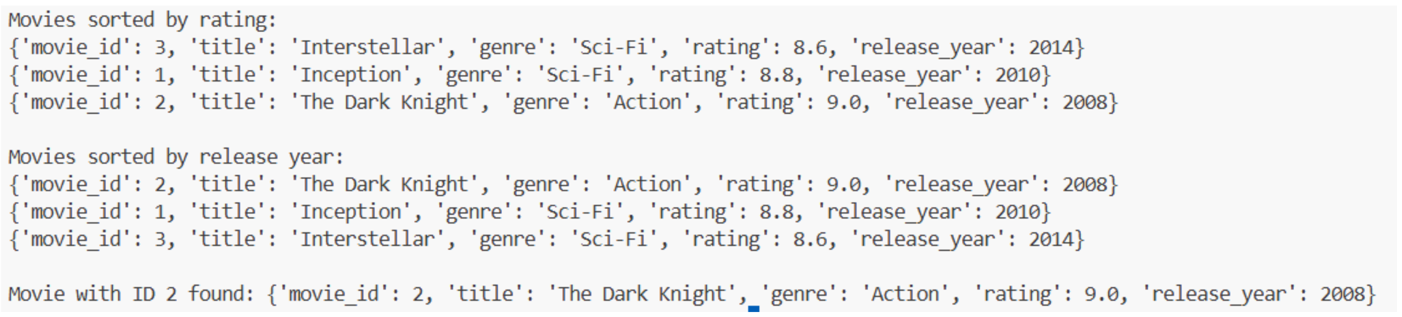
Code:

```python
def movie_search(movies, movie_id):
    left, right = 0, len(movies) - 1
    while left <= right:
        mid = left + (right - left) // 2
        if movies[mid]['movie_id'] == movie_id:
            return movies[mid]
        elif movies[mid]['movie_id'] < movie_id:
            left = mid + 1
        else:
            right = mid - 1
    return None
def merge_sort(arr, key):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left_half = merge_sort(arr[:mid], key)
    right_half = merge_sort(arr[mid:], key)
    return merge(left_half, right_half, key)
```



```python
def merge(left, right, key):
    merged = []
    left_index = right_index = 0
    while left_index < len(left) and right_index < len(right):
        if left[left_index][key] < right[right_index][key]:
            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
if __name__ == "__main__":
    movies = [
        {'movie_id': 1, 'title': 'Inception', 'genre': 'Sci-Fi', 'rating': 8.8, 'release_year': 2010},
        {'movie_id': 2, 'title': 'The Dark Knight', 'genre': 'Action', 'rating': 9.0, 'release_year': 2008},
        {'movie_id': 3, 'title': 'Interstellar', 'genre': 'Sci-Fi', 'rating': 8.6, 'release_year': 2014}]
    # Sort movies by rating
    sorted_by_rating = merge_sort(movies, key='rating')
    print("Movies sorted by rating:")
    for movie in sorted_by_rating:
        print(movie)
    # Sort movies by release year
    sorted_by_year = merge_sort(movies, key='release_year')
    print("\nMovies sorted by release year:")
    for movie in sorted_by_year:
        print(movie)
    # Search for a movie by ID
    movie_id_to_search = 2
    sorted_by_id = merge_sort(movies, key='movie_id')
    found_movie = movie_search(sorted_by_id, movie_id_to_search)
    if found_movie:
        print(f"\nMovie with ID {movie_id_to_search} found: {found_movie}")
    else:
        print(f"\nMovie with ID {movie_id_to_search} not found.")
```

Output:



```
Movies sorted by rating:
{'movie_id': 3, 'title': 'Interstellar', 'genre': 'Sci-Fi', 'rating': 8.6, 'release_year': 2014}
{'movie_id': 1, 'title': 'Inception', 'genre': 'Sci-Fi', 'rating': 8.8, 'release_year': 2010}
{'movie_id': 2, 'title': 'The Dark Knight', 'genre': 'Action', 'rating': 9.0, 'release_year': 2008}

Movies sorted by release year:
{'movie_id': 2, 'title': 'The Dark Knight', 'genre': 'Action', 'rating': 9.0, 'release_year': 2008}
{'movie_id': 1, 'title': 'Inception', 'genre': 'Sci-Fi', 'rating': 8.8, 'release_year': 2010}
{'movie_id': 3, 'title': 'Interstellar', 'genre': 'Sci-Fi', 'rating': 8.6, 'release_year': 2014}

Movie with ID 2 found: {'movie_id': 2, 'title': 'The Dark Knight', 'genre': 'Action', 'rating': 9.0, 'release_year': 2008}
```

Explanation:

Binary Search is used for searching movie records because it provides O (log n) time complexity and is efficient for large datasets. Merge Sort is used for sorting movies based on rating or release year because it guarantees O (n log n) performance and maintains stability.

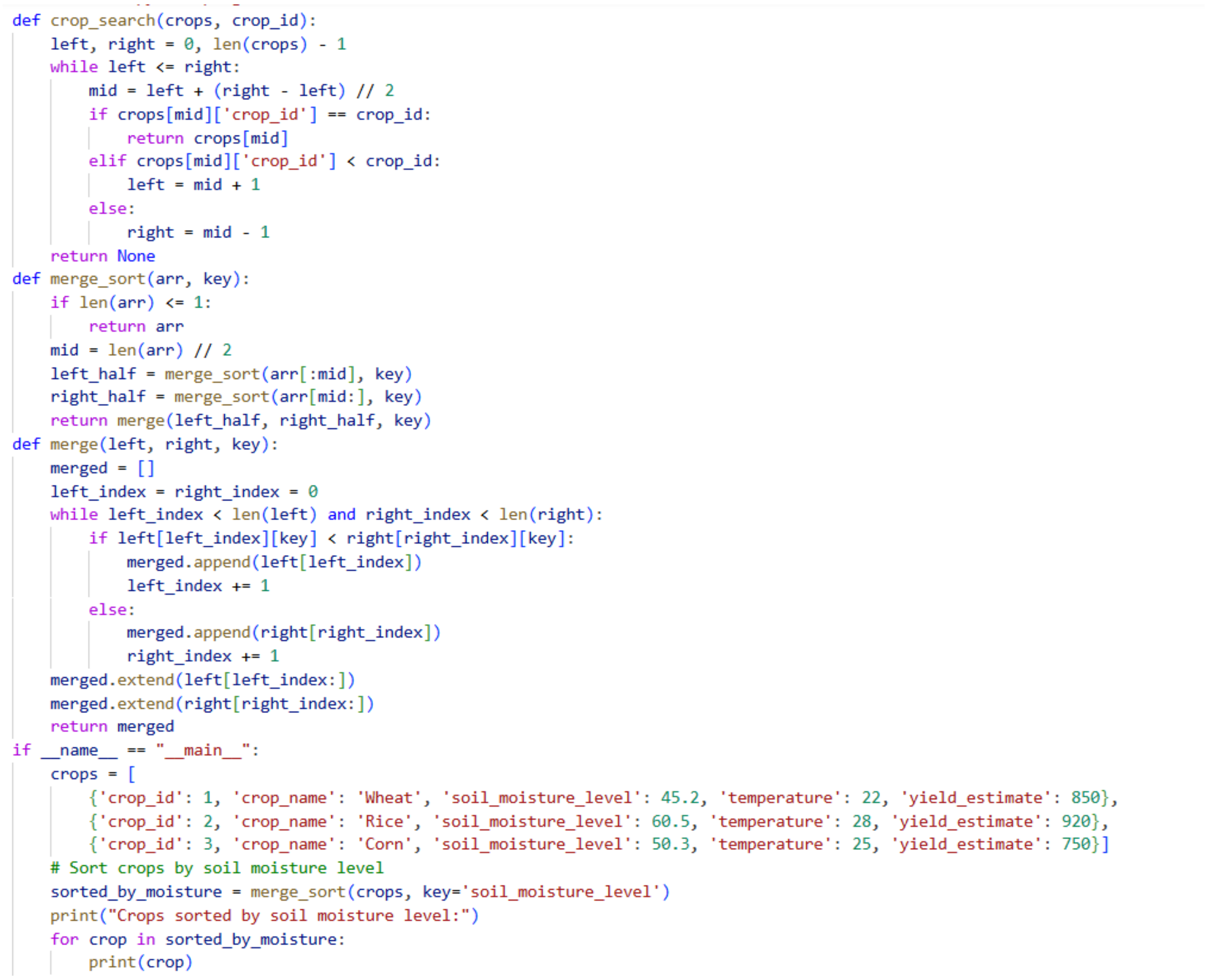
Task 7: Smart Agriculture Crop Monitoring System

Prompt: An agriculture monitoring system stores crop data with crop ID, crop name, soil moisture level, temperature, and yield estimate. Farmers need to:

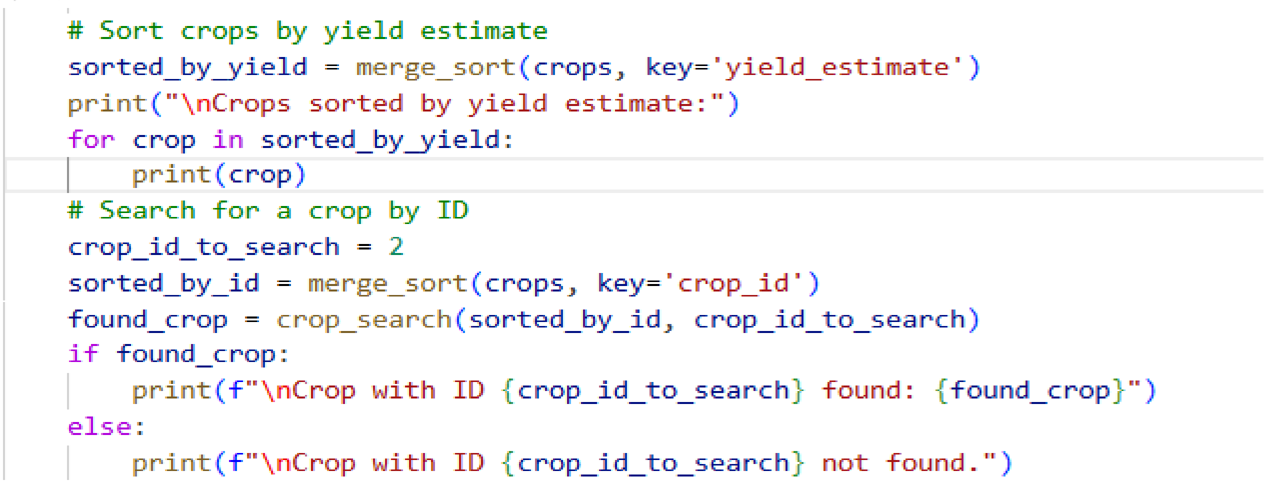
1. Search crop details using crop ID: Use binary search to search the crop details.
2. Sort crops based on moisture level or yield estimate: Use merge sort to sort crops.

Generate the python program for above scenario.

Code:

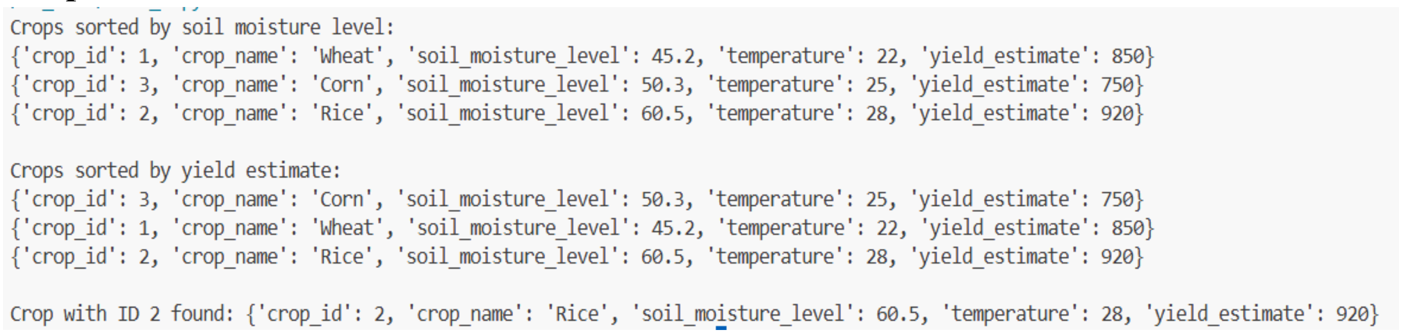


```python
def crop_search(crops, crop_id):
    left, right = 0, len(crops) - 1
    while left <= right:
        mid = left + (right - left) // 2
        if crops[mid]['crop_id'] == crop_id:
            return crops[mid]
        elif crops[mid]['crop_id'] < crop_id:
            left = mid + 1
        else:
            right = mid - 1
    return None
def merge_sort(arr, key):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left_half = merge_sort(arr[:mid], key)
    right_half = merge_sort(arr[mid:], key)
    return merge(left_half, right_half, key)
def merge(left, right, key):
    merged = []
    left_index = right_index = 0
    while left_index < len(left) and right_index < len(right):
        if left[left_index][key] < right[right_index][key]:
            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
if __name__ == "__main__":
    crops = [
        {'crop_id': 1, 'crop_name': 'Wheat', 'soil_moisture_level': 45.2, 'temperature': 22, 'yield_estimate': 850},
        {'crop_id': 2, 'crop_name': 'Rice', 'soil_moisture_level': 60.5, 'temperature': 28, 'yield_estimate': 920},
        {'crop_id': 3, 'crop_name': 'Corn', 'soil_moisture_level': 50.3, 'temperature': 25, 'yield_estimate': 750}]
    # Sort crops by soil moisture level
    sorted_by_moisture = merge_sort(crops, key='soil_moisture_level')
    print("Crops sorted by soil moisture level:")
    for crop in sorted_by_moisture:
        print(crop)
```



```python
    # Sort crops by yield estimate
    sorted_by_yield = merge_sort(crops, key='yield_estimate')
    print("\nCrops sorted by yield estimate:")
    for crop in sorted_by_yield:
        print(crop)
    # Search for a crop by ID
    crop_id_to_search = 2
    sorted_by_id = merge_sort(crops, key='crop_id')
    found_crop = crop_search(sorted_by_id, crop_id_to_search)
    if found_crop:
        print(f"\nCrop with ID {crop_id_to_search} found: {found_crop}")
    else:
        print(f"\nCrop with ID {crop_id_to_search} not found.")
```

Output:



```
Crops sorted by soil moisture level:
{'crop_id': 1, 'crop_name': 'Wheat', 'soil_moisture_level': 45.2, 'temperature': 22, 'yield_estimate': 850}
{'crop_id': 3, 'crop_name': 'Corn', 'soil_moisture_level': 50.3, 'temperature': 25, 'yield_estimate': 750}
{'crop_id': 2, 'crop_name': 'Rice', 'soil_moisture_level': 60.5, 'temperature': 28, 'yield_estimate': 920}

Crops sorted by yield estimate:
{'crop_id': 3, 'crop_name': 'Corn', 'soil_moisture_level': 50.3, 'temperature': 25, 'yield_estimate': 750}
{'crop_id': 1, 'crop_name': 'Wheat', 'soil_moisture_level': 45.2, 'temperature': 22, 'yield_estimate': 850}
{'crop_id': 2, 'crop_name': 'Rice', 'soil_moisture_level': 60.5, 'temperature': 28, 'yield_estimate': 920}

Crop with ID 2 found: {'crop_id': 2, 'crop_name': 'Rice', 'soil_moisture_level': 60.5, 'temperature': 28, 'yield_estimate': 920}
```

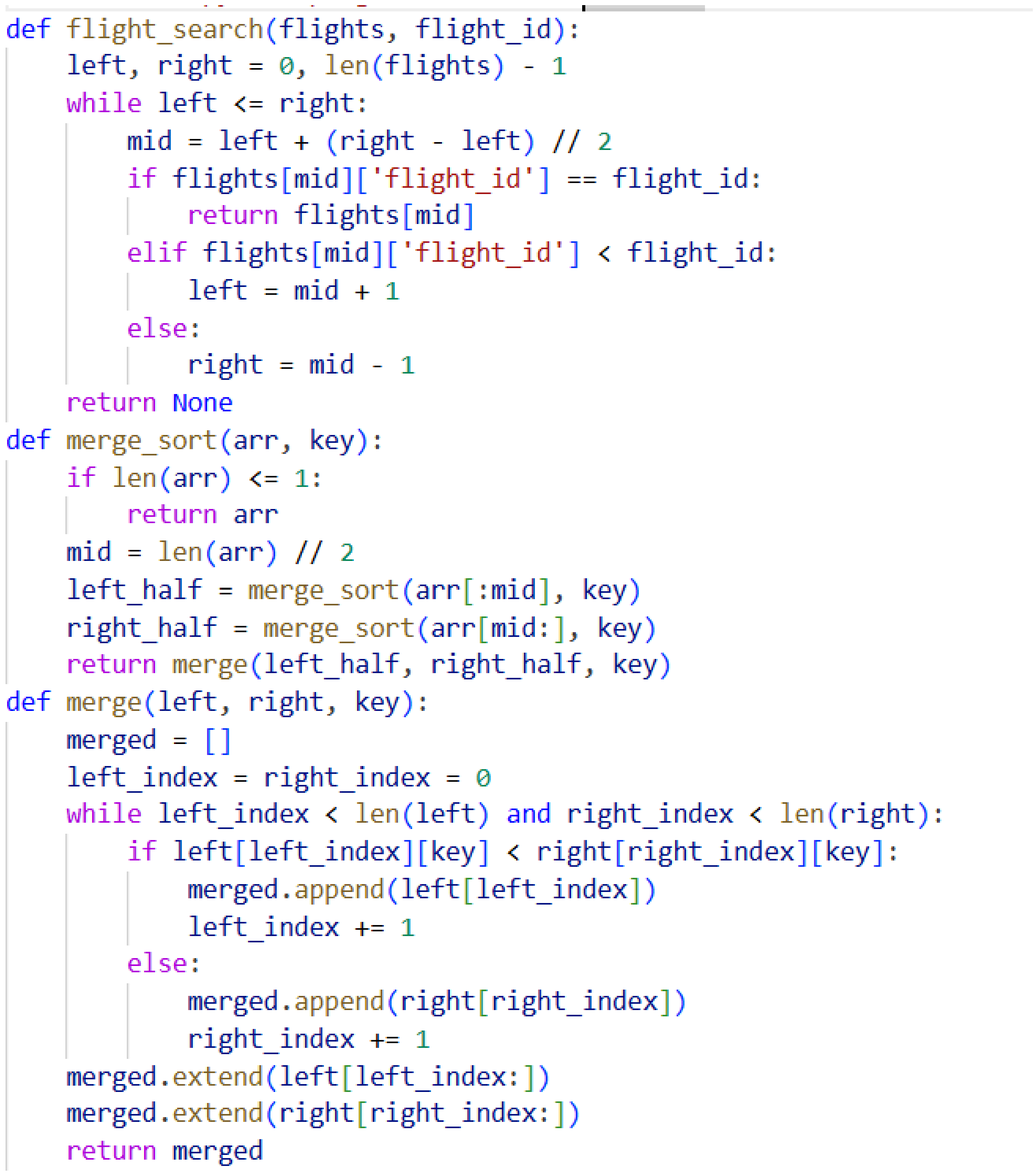
Explanation:

- **Binary Search** is used because it provides fast searching with **O (log n)** time complexity by repeatedly dividing the sorted data into halves.
- **Merge Sort** is chosen because it guarantees efficient and stable sorting with **O (n log n)** time complexity in all cases, making it suitable for large datasets.

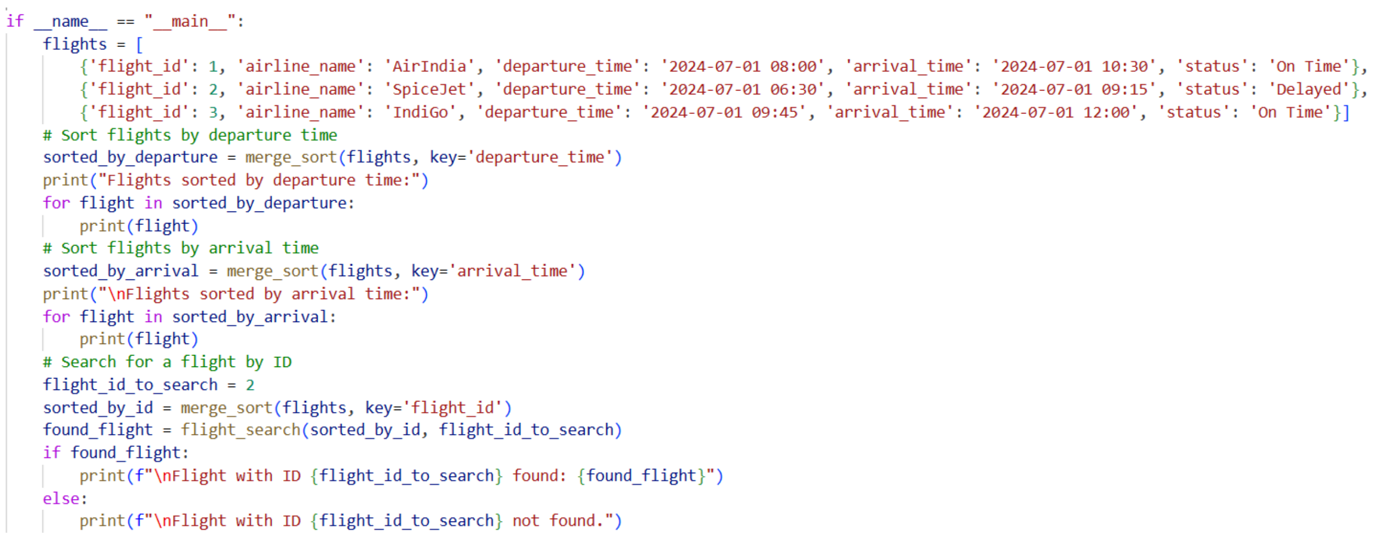
Task 8: Airport Flight Management System

Prompt: An airport system stores flight information including flight ID, airline name, departure time, arrival time, and status. The system must:
1. Search flight details using flight ID: Use binary search to search flight details.
2. Sort flights based on departure time or arrival time: Use merge sort to sort the flights.
Generate the python program for above scenario.

Code:

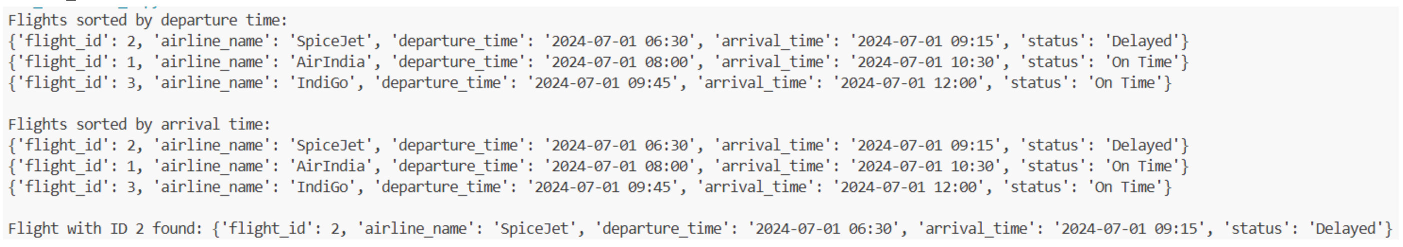


```python
def flight_search(flights, flight_id):
    left, right = 0, len(flights) - 1
    while left <= right:
        mid = left + (right - left) // 2
        if flights[mid]['flight_id'] == flight_id:
            return flights[mid]
        elif flights[mid]['flight_id'] < flight_id:
            left = mid + 1
        else:
            right = mid - 1
    return None
def merge_sort(arr, key):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left_half = merge_sort(arr[:mid], key)
    right_half = merge_sort(arr[mid:], key)
    return merge(left_half, right_half, key)
def merge(left, right, key):
    merged = []
    left_index = right_index = 0
    while left_index < len(left) and right_index < len(right):
        if left[left_index][key] < right[right_index][key]:
            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
```



```python
if __name__ == "__main__":
    flights = [
        {'flight_id': 1, 'airline_name': 'AirIndia', 'departure_time': '2024-07-01 08:00', 'arrival_time': '2024-07-01 10:30', 'status': 'On Time'},
        {'flight_id': 2, 'airline_name': 'SpiceJet', 'departure_time': '2024-07-01 06:30', 'arrival_time': '2024-07-01 09:15', 'status': 'Delayed'},
        {'flight_id': 3, 'airline_name': 'IndiGo', 'departure_time': '2024-07-01 09:45', 'arrival_time': '2024-07-01 12:00', 'status': 'On Time'}]
    # Sort flights by departure time
    sorted_by_departure = merge_sort(flights, key='departure_time')
    print("Flights sorted by departure time:")
    for flight in sorted_by_departure:
        print(flight)
    # Sort flights by arrival time
    sorted_by_arrival = merge_sort(flights, key='arrival_time')
    print("\nFlights sorted by arrival time:")
    for flight in sorted_by_arrival:
        print(flight)
    # Search for a flight by ID
    flight_id_to_search = 2
    sorted_by_id = merge_sort(flights, key='flight_id')
    found_flight = flight_search(sorted_by_id, flight_id_to_search)
    if found_flight:
        print(f"\nFlight with ID {flight_id_to_search} found: {found_flight}")
    else:
        print(f"\nFlight with ID {flight_id_to_search} not found.")
```

Output:



```
Flights sorted by departure time:
{'flight_id': 2, 'airline_name': 'SpiceJet', 'departure_time': '2024-07-01 06:30', 'arrival_time': '2024-07-01 09:15', 'status': 'Delayed'}
{'flight_id': 1, 'airline_name': 'AirIndia', 'departure_time': '2024-07-01 08:00', 'arrival_time': '2024-07-01 10:30', 'status': 'On Time'}
{'flight_id': 3, 'airline_name': 'IndiGo', 'departure_time': '2024-07-01 09:45', 'arrival_time': '2024-07-01 12:00', 'status': 'On Time'}

Flights sorted by arrival time:
{'flight_id': 2, 'airline_name': 'SpiceJet', 'departure_time': '2024-07-01 06:30', 'arrival_time': '2024-07-01 09:15', 'status': 'Delayed'}
{'flight_id': 1, 'airline_name': 'AirIndia', 'departure_time': '2024-07-01 08:00', 'arrival_time': '2024-07-01 10:30', 'status': 'On Time'}
{'flight_id': 3, 'airline_name': 'IndiGo', 'departure_time': '2024-07-01 09:45', 'arrival_time': '2024-07-01 12:00', 'status': 'On Time'}

Flight with ID 2 found: {'flight_id': 2, 'airline_name': 'SpiceJet', 'departure_time': '2024-07-01 06:30', 'arrival_time': '2024-07-01 09:15', 'status': 'Delayed'}
```

Explanation:

- Binary Search is selected because flight ID is unique and searching becomes faster with O (log n) time complexity compared to linear search O(n). It efficiently reduces the search space by half in each step.
- Merge Sort is selected because it guarantees O (n log n) time complexity in all cases. It is stable and efficient for sorting large flight datasets based on departure or arrival time.