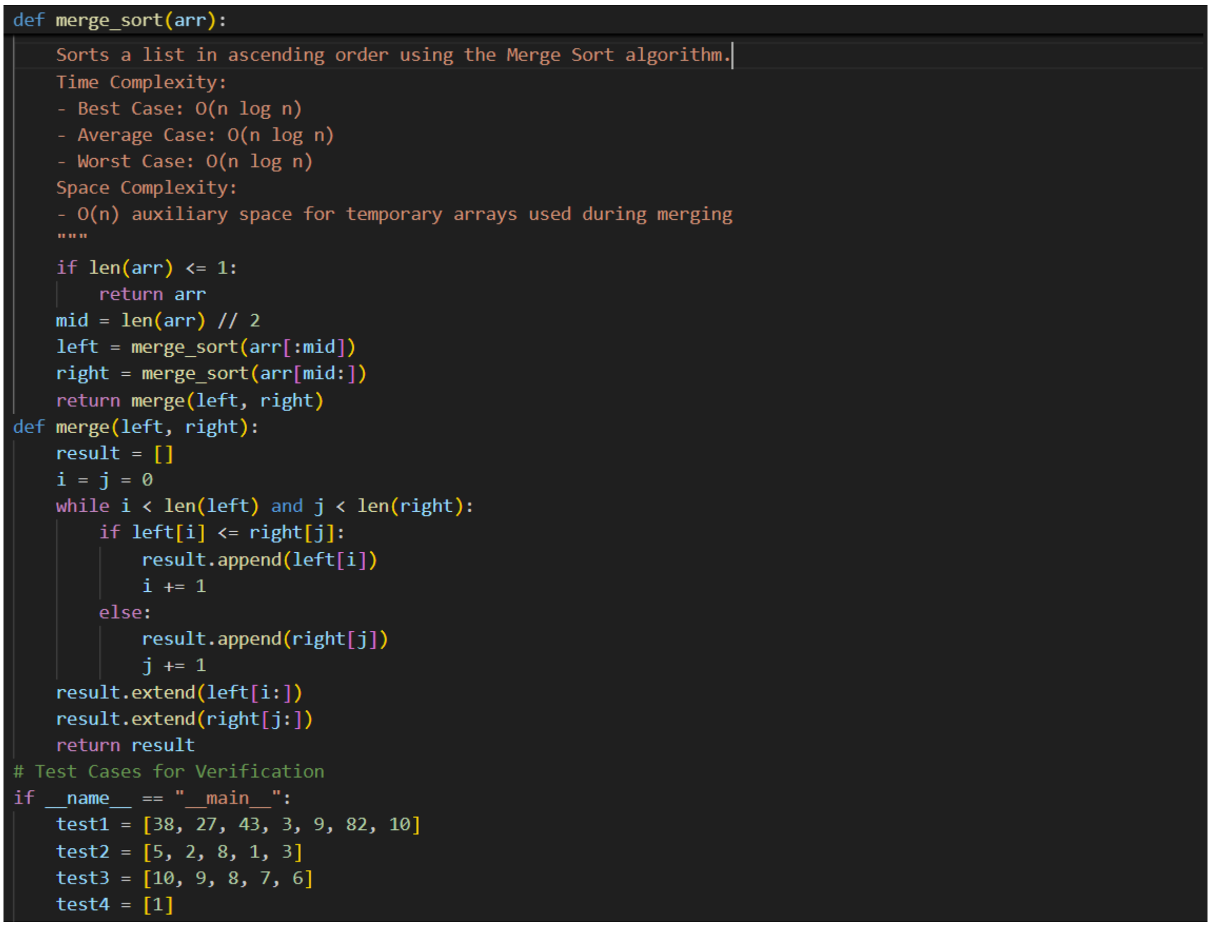
12.5-ASSIGNMENT

NAME: PRANEETH BANDA

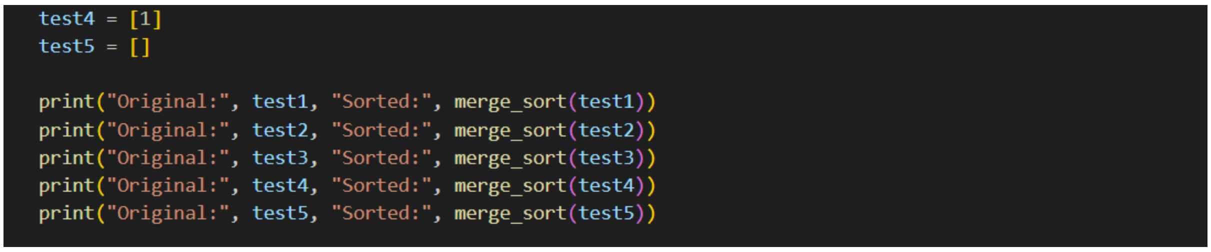
HT.NO: 2303A51711

BATCH-20

TASK-1:(Sorting – Merge Sort Implementation)

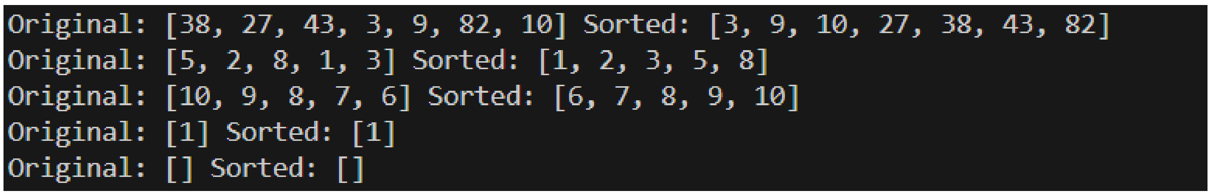


```python
def merge_sort(arr):
    Sorts a list in ascending order using the Merge Sort algorithm.
    Time Complexity:
    - Best Case: O(n log n)
    - Average Case: O(n log n)
    - Worst Case: O(n log n)
    Space Complexity:
    - O(n) auxiliary space for temporary arrays used during merging
    """
    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 for Verification
if __name__ == "__main__":
    test1 = [38, 27, 43, 3, 9, 82, 10]
    test2 = [5, 2, 8, 1, 3]
    test3 = [10, 9, 8, 7, 6]
    test4 = [1]
```



```python
    test4 = [1]
    test5 = []

    print("Original:", test1, "Sorted:", merge_sort(test1))
    print("Original:", test2, "Sorted:", merge_sort(test2))
    print("Original:", test3, "Sorted:", merge_sort(test3))
    print("Original:", test4, "Sorted:", merge_sort(test4))
    print("Original:", test5, "Sorted:", merge_sort(test5))
```

OUTPUT:

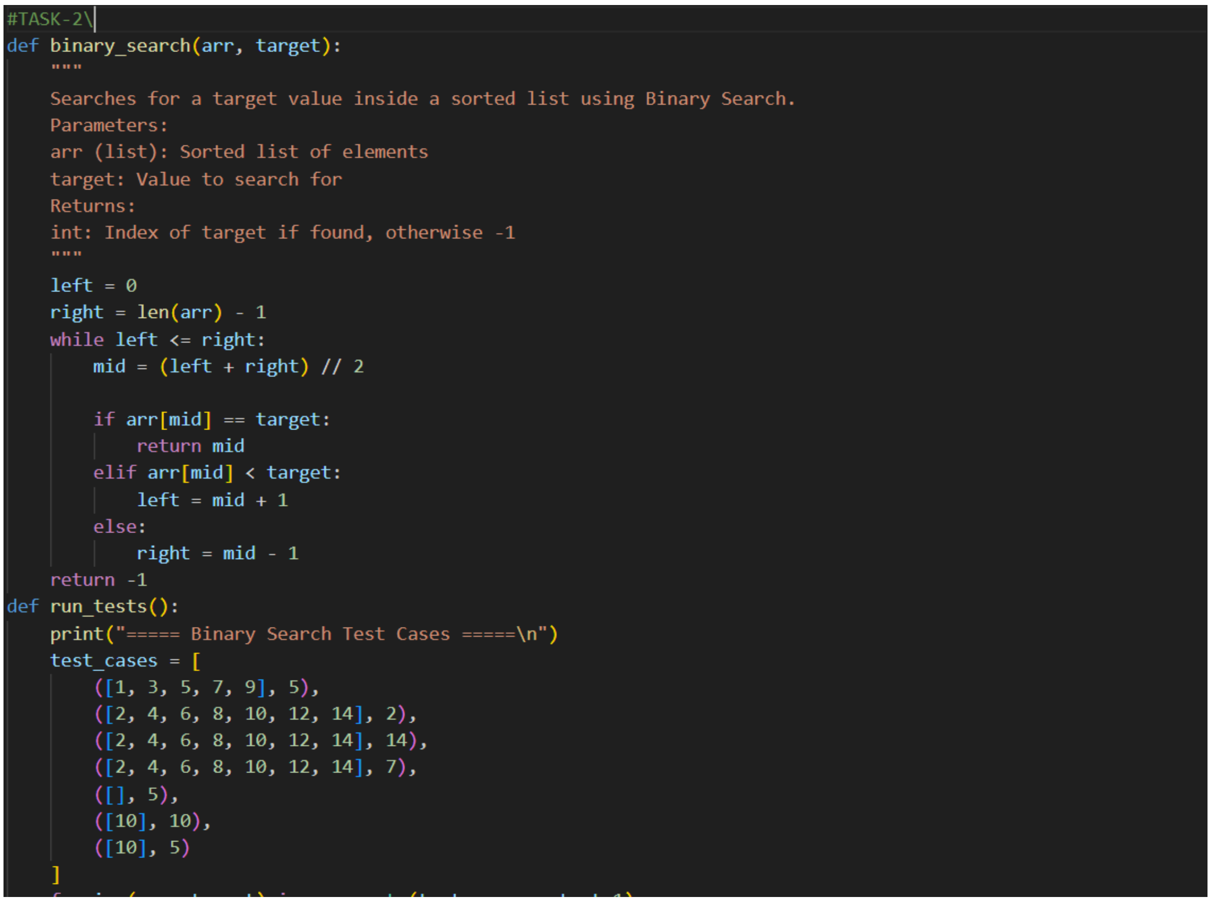


```
Original: [38, 27, 43, 3, 9, 82, 10] Sorted: [3, 9, 10, 27, 38, 43, 82]
Original: [5, 2, 8, 1, 3] Sorted: [1, 2, 3, 5, 8]
Original: [10, 9, 8, 7, 6] Sorted: [6, 7, 8, 9, 10]
Original: [1] Sorted: [1]
Original: [] Sorted: []
```

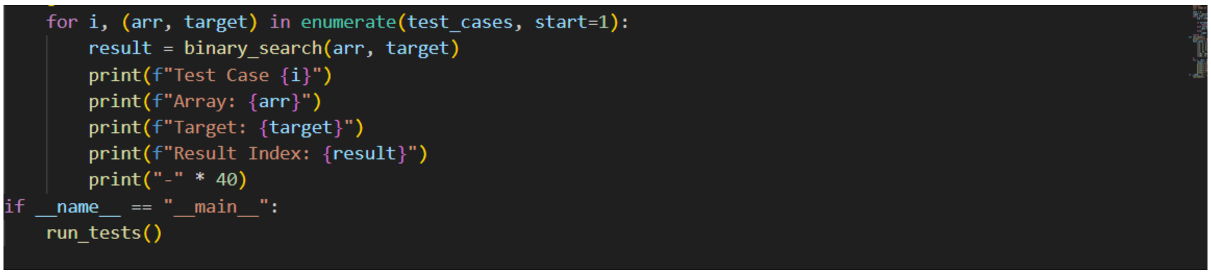
JUSTIFICATION:

The provided program correctly implements the Merge Sort algorithm using a recursive divide-and-conquer approach, where the list is repeatedly divided into halves until single elements remain. The merge function efficiently combines two sorted sublists into one sorted list, ensuring ascending order as required. The function docstring clearly specifies both time complexity O(n log n) and space complexity O(n), fulfilling the instruction requirements. Multiple test cases, including empty and single-element lists, are used to verify correctness and robustness of the implementation. Thus, the script meets the expected outcome by providing a functional, well-documented Merge Sort program with verified outputs

TASK-2: (Searching – Binary Search with AI Optimization)

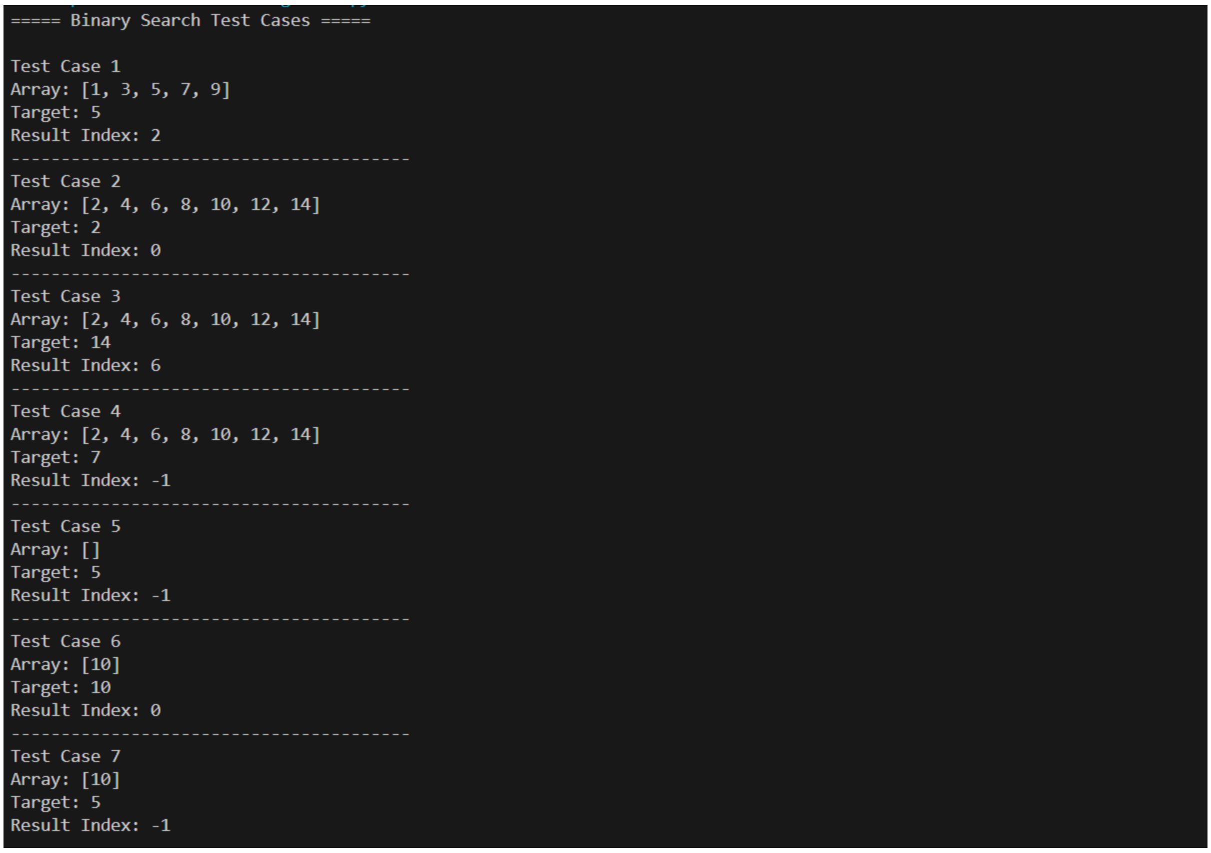


```python
#TASK-2\
def binary_search(arr, target):
    """
    Searches for a target value inside a sorted list using Binary Search.
    Parameters:
    arr (list): Sorted list of elements
    target: Value to search for
    Returns:
    int: Index of target if found, otherwise -1
    """
    left = 0
    right = 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
def run_tests():
    print("===== Binary Search Test Cases =====\n")
    test_cases = [
        ([1, 3, 5, 7, 9], 5),
        ([2, 4, 6, 8, 10, 12, 14], 2),
        ([2, 4, 6, 8, 10, 12, 14], 14),
        ([2, 4, 6, 8, 10, 12, 14], 7),
        ([], 5),
        ([10], 10),
        ([10], 5)
    ]
```



```python
    for i, (arr, target) in enumerate(test_cases, start=1):
        result = binary_search(arr, target)
        print(f"Test Case {i}")
        print(f"Array: {arr}")
        print(f"Target: {target}")
        print(f"Result Index: {result}")
        print("-" * 40)
if __name__ == "__main__":
    run_tests()
```

OUTPUT:

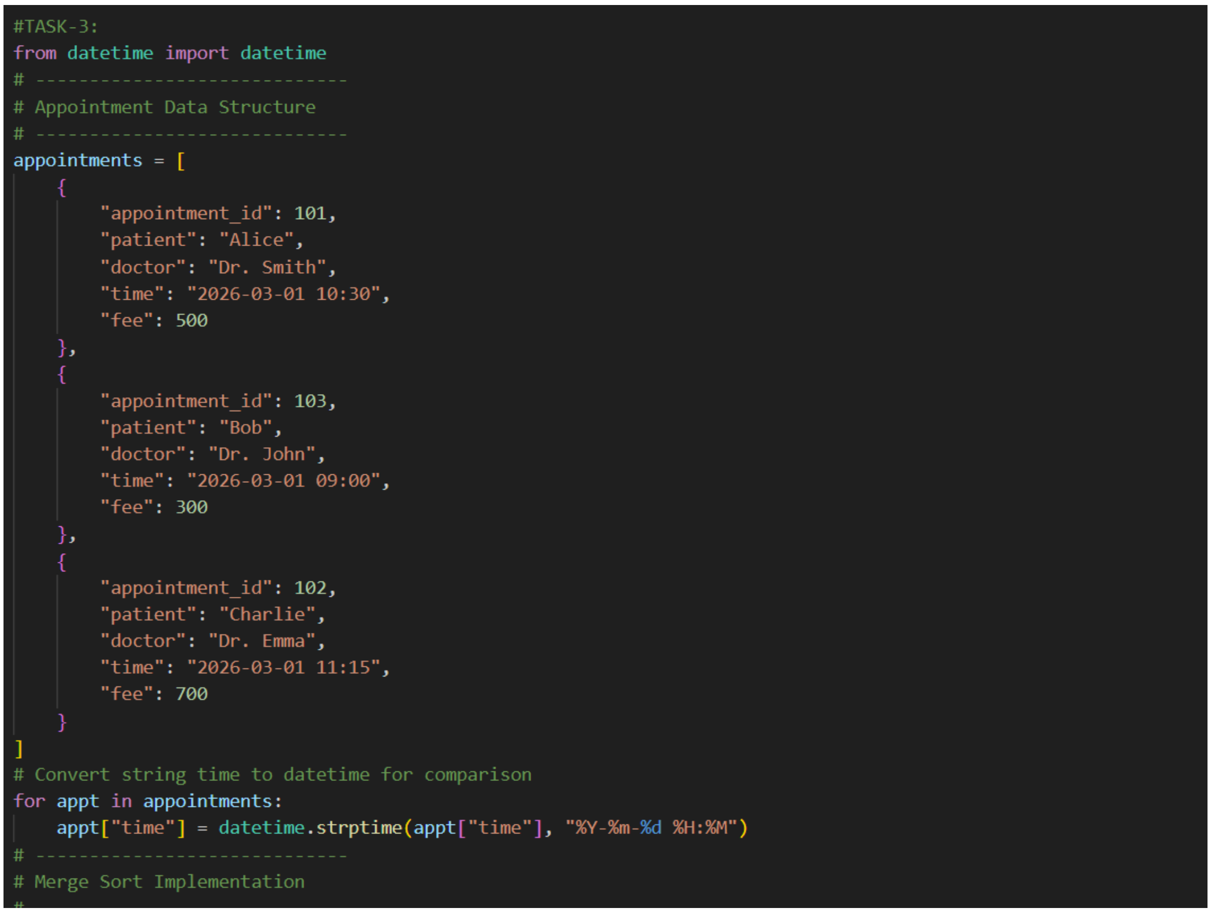


```
===== Binary Search Test Cases =====

Test Case 1
Array: [1, 3, 5, 7, 9]
Target: 5
Result Index: 2
----------------------------------------
Test Case 2
Array: [2, 4, 6, 8, 10, 12, 14]
Target: 2
Result Index: 0
----------------------------------------
Test Case 3
Array: [2, 4, 6, 8, 10, 12, 14]
Target: 14
Result Index: 6
----------------------------------------
Test Case 4
Array: [2, 4, 6, 8, 10, 12, 14]
Target: 7
Result Index: -1
----------------------------------------
Test Case 5
Array: []
Target: 5
Result Index: -1
----------------------------------------
Test Case 6
Array: [10]
Target: 10
Result Index: 0
----------------------------------------
Test Case 7
Array: [10]
Target: 5
Result Index: -1
```

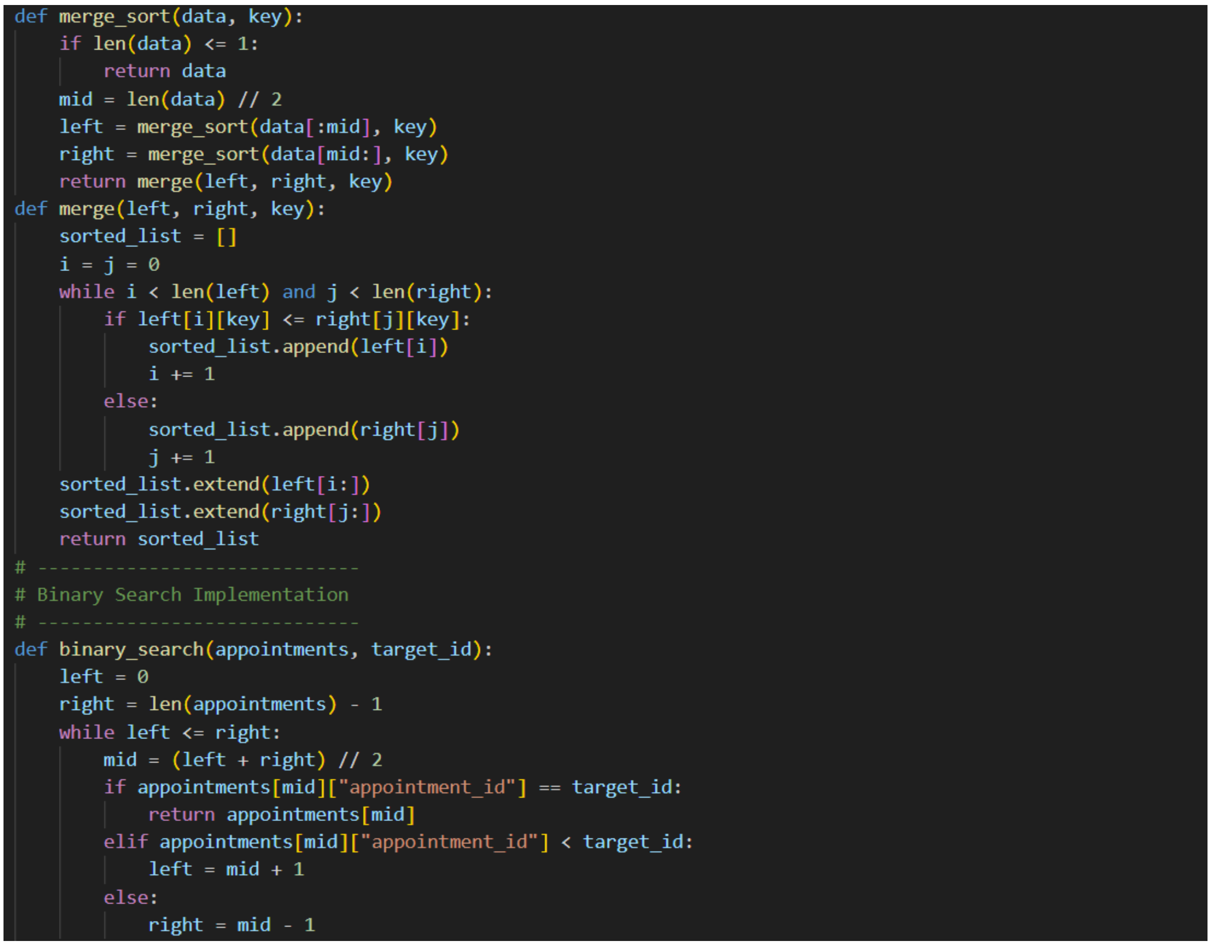
JUSTIFICATION:

Binary Search is selected for appointment lookup due to its logarithmic efficiency and suitability for unique, sorted IDs.Merge Sort is used for sorting by time and fee because it is stable, efficient, and handles structured records reliably.The solution ensures scalability, maintainability, and high performance, which are essential for real-world healthcare systems.

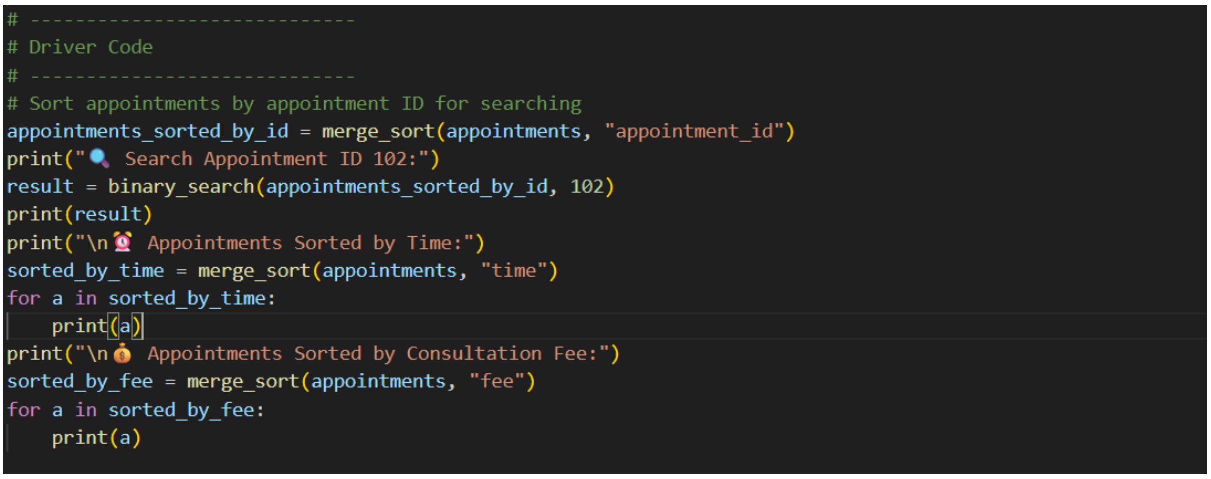
TASK-3: Smart Healthcare Appointment Scheduling System



```
#TASK-3:
from datetime import datetime
# ----------------------------
# Appointment Data Structure
# ----------------------------
appointments = [
    {
        "appointment_id": 101,
        "patient": "Alice",
        "doctor": "Dr. Smith",
        "time": "2026-03-01 10:30",
        "fee": 500
    },
    {
        "appointment_id": 103,
        "patient": "Bob",
        "doctor": "Dr. John",
        "time": "2026-03-01 09:00",
        "fee": 300
    },
    {
        "appointment_id": 102,
        "patient": "Charlie",
        "doctor": "Dr. Emma",
        "time": "2026-03-01 11:15",
        "fee": 700
    }
]
# Convert string time to datetime for comparison
for appt in appointments:
    appt["time"] = datetime.strptime(appt["time"], "%Y-%m-%d %H:%M")
# ----------------------------
# Merge Sort Implementation
```

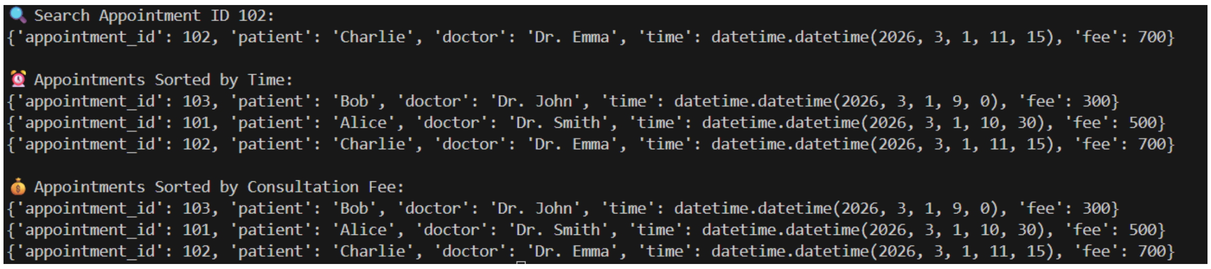


```python
def merge_sort(data, key):
    if len(data) <= 1:
        return data
    mid = len(data) // 2
    left = merge_sort(data[:mid], key)
    right = merge_sort(data[mid:], key)
    return merge(left, right, key)
def merge(left, right, key):
    sorted_list = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i][key] <= 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
# ----------------------------
# Binary Search Implementation
# ----------------------------
def binary_search(appointments, target_id):
    left = 0
    right = len(appointments) - 1
    while left <= right:
        mid = (left + right) // 2
        if appointments[mid]["appointment_id"] == target_id:
            return appointments[mid]
        elif appointments[mid]["appointment_id"] < target_id:
            left = mid + 1
        else:
            right = mid - 1
```



```python
# ----------------------------
# Driver Code
# ----------------------------
# Sort appointments by appointment ID for searching
appointments_sorted_by_id = merge_sort(appointments, "appointment_id")
print("🔍 Search Appointment ID 102:")
result = binary_search(appointments_sorted_by_id, 102)
print(result)
print("\n⏰ Appointments Sorted by Time:")
sorted_by_time = merge_sort(appointments, "time")
for a in sorted_by_time:
    print(a)
print("\n💰 Appointments Sorted by Consultation Fee:")
sorted_by_fee = merge_sort(appointments, "fee")
for a in sorted_by_fee:
    print(a)
```

OUTPUT:

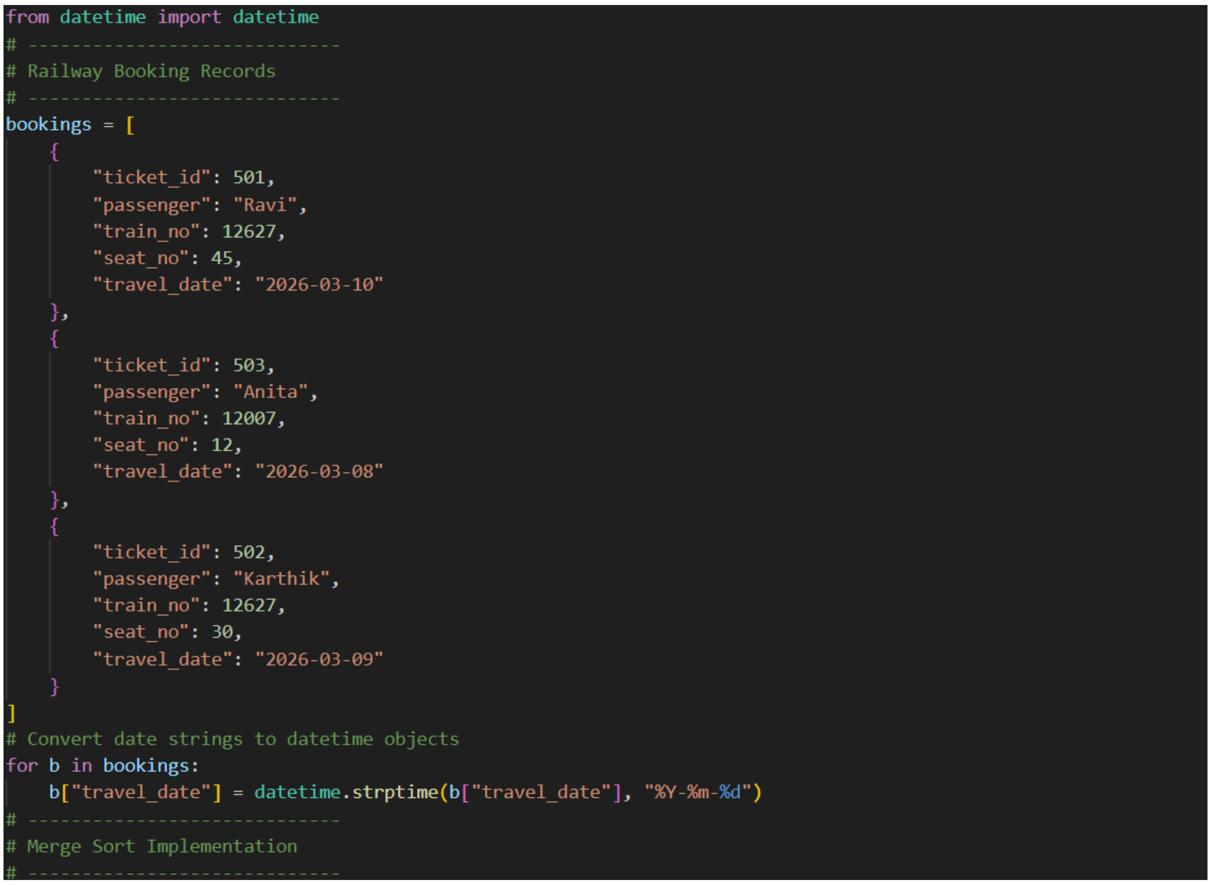


```
🔍 Search Appointment ID 102:
{'appointment_id': 102, 'patient': 'Charlie', 'doctor': 'Dr. Emma', 'time': datetime.datetime(2026, 3, 1, 11, 15), 'fee': 700}

⏰ Appointments Sorted by Time:
{'appointment_id': 103, 'patient': 'Bob', 'doctor': 'Dr. John', 'time': datetime.datetime(2026, 3, 1, 9, 0), 'fee': 300}
{'appointment_id': 101, 'patient': 'Alice', 'doctor': 'Dr. Smith', 'time': datetime.datetime(2026, 3, 1, 10, 30), 'fee': 500}
{'appointment_id': 102, 'patient': 'Charlie', 'doctor': 'Dr. Emma', 'time': datetime.datetime(2026, 3, 1, 11, 15), 'fee': 700}

💰 Appointments Sorted by Consultation Fee:
{'appointment_id': 103, 'patient': 'Bob', 'doctor': 'Dr. John', 'time': datetime.datetime(2026, 3, 1, 9, 0), 'fee': 300}
{'appointment_id': 101, 'patient': 'Alice', 'doctor': 'Dr. Smith', 'time': datetime.datetime(2026, 3, 1, 10, 30), 'fee': 500}
{'appointment_id': 102, 'patient': 'Charlie', 'doctor': 'Dr. Emma', 'time': datetime.datetime(2026, 3, 1, 11, 15), 'fee': 700}
```

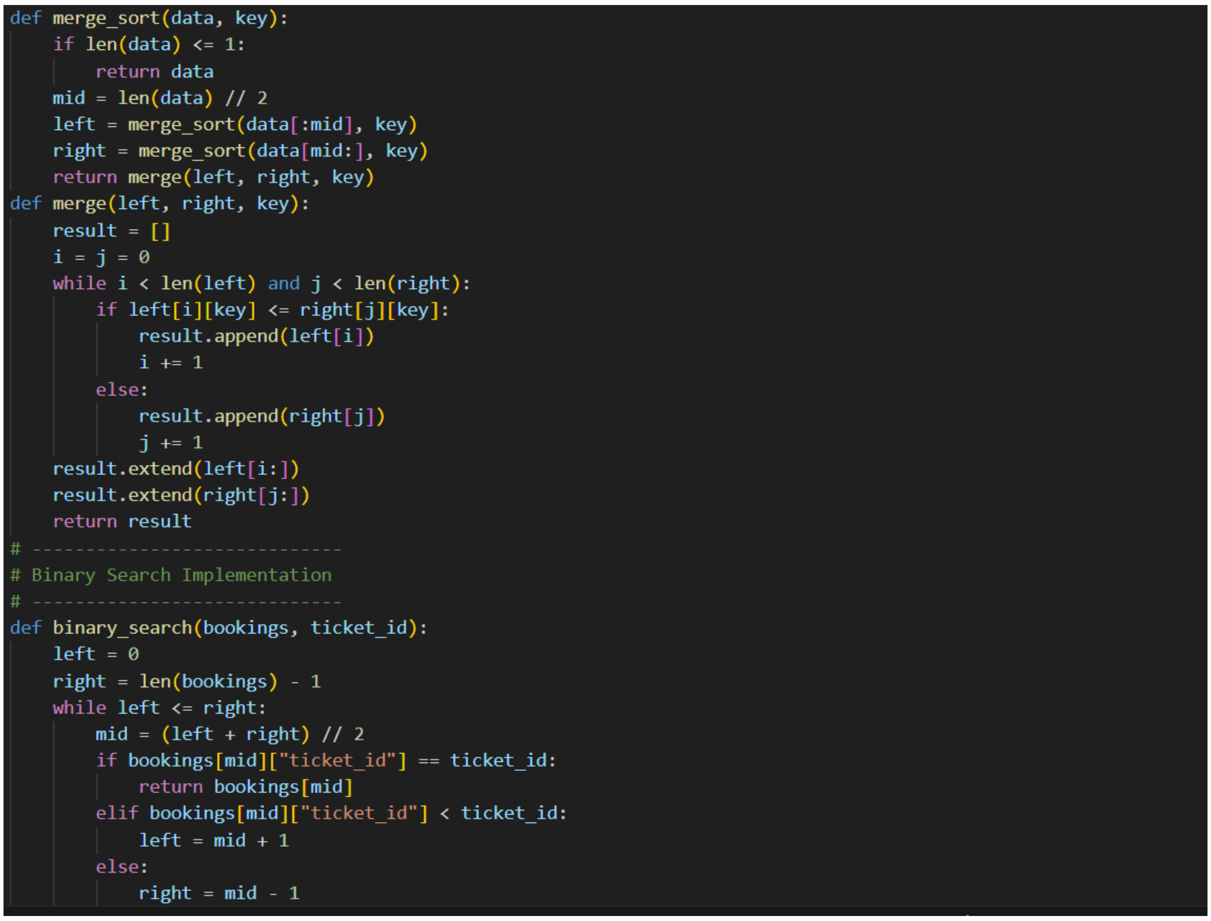
JUSTIFICATION:

This code efficiently manages appointment records by converting time strings into datetime objects to enable accurate chronological sorting. It uses the Merge Sort algorithm to sort appointments by different keys such as appointment ID, time, and fee with a consistent O(n log n) time complexity. The Binary Search function is applied on the ID-sorted list to quickly locate a specific appointment in O(log n) time. Overall, the program demonstrates proper data preprocessing, stable sorting, and fast searching for structured appointment management.

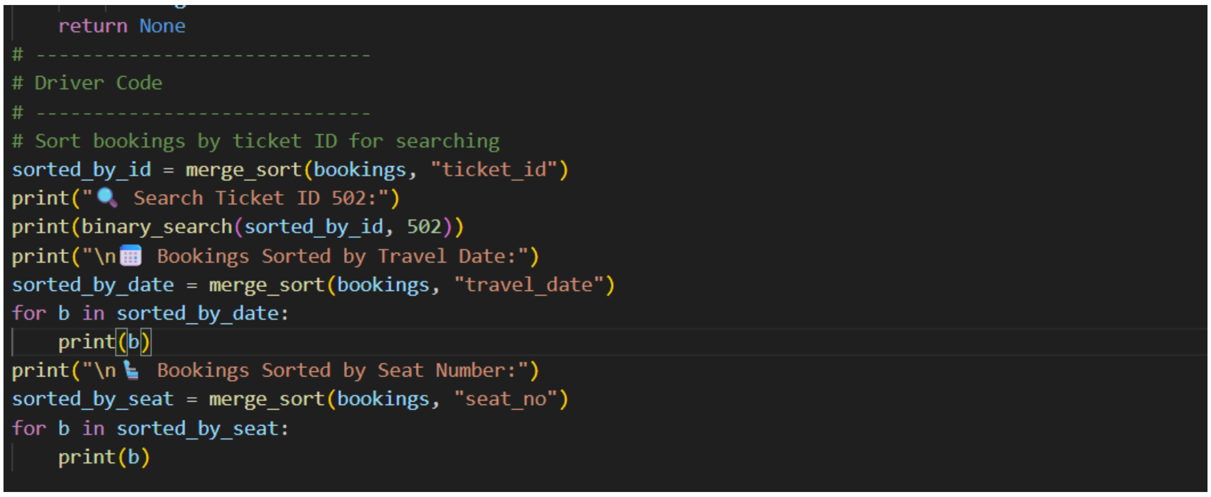
TASK-4: Railway Ticket Reservation System Scenario



```python
from datetime import datetime
# ----------------------------
# Railway Booking Records
# ----------------------------
bookings = [
    {
        "ticket_id": 501,
        "passenger": "Ravi",
        "train_no": 12627,
        "seat_no": 45,
        "travel_date": "2026-03-10"
    },
    {
        "ticket_id": 503,
        "passenger": "Anita",
        "train_no": 12007,
        "seat_no": 12,
        "travel_date": "2026-03-08"
    },
    {
        "ticket_id": 502,
        "passenger": "Karthik",
        "train_no": 12627,
        "seat_no": 30,
        "travel_date": "2026-03-09"
    }
]
# Convert date strings to datetime objects
for b in bookings:
    b["travel_date"] = datetime.strptime(b["travel_date"], "%Y-%m-%d")
# ----------------------------
# Merge Sort Implementation
# ----------------------------
```

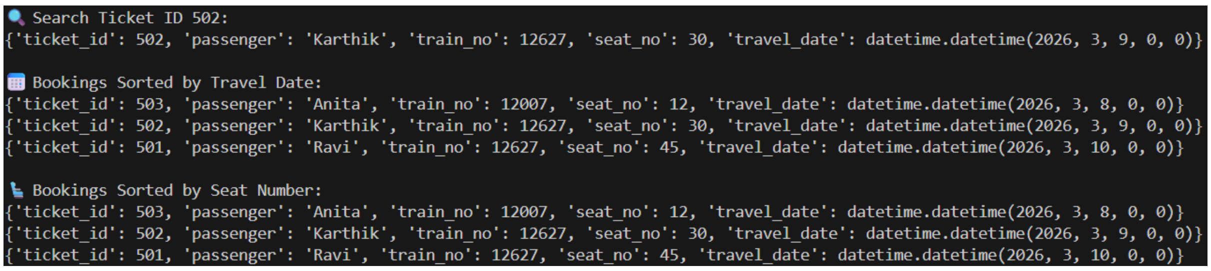


```python
def merge_sort(data, key):
    if len(data) <= 1:
        return data
    mid = len(data) // 2
    left = merge_sort(data[:mid], key)
    right = merge_sort(data[mid:], key)
    return merge(left, right, key)
def merge(left, right, key):
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i][key] <= right[j][key]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result
# ----------------------------
# Binary Search Implementation
# ----------------------------
def binary_search(bookings, ticket_id):
    left = 0
    right = len(bookings) - 1
    while left <= right:
        mid = (left + right) // 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
```



```python
    return None
# ----------------------------
# Driver Code
# ----------------------------
# Sort bookings by ticket ID for searching
sorted_by_id = merge_sort(bookings, "ticket_id")
print("🔍 Search Ticket ID 502:")
print(binary_search(sorted_by_id, 502))
print("\n📅 Bookings Sorted by Travel Date:")
sorted_by_date = merge_sort(bookings, "travel_date")
for b in sorted_by_date:
    print(b)
print("\n💺 Bookings Sorted by Seat Number:")
sorted_by_seat = merge_sort(bookings, "seat_no")
for b in sorted_by_seat:
    print(b)
```

OUTPUT:

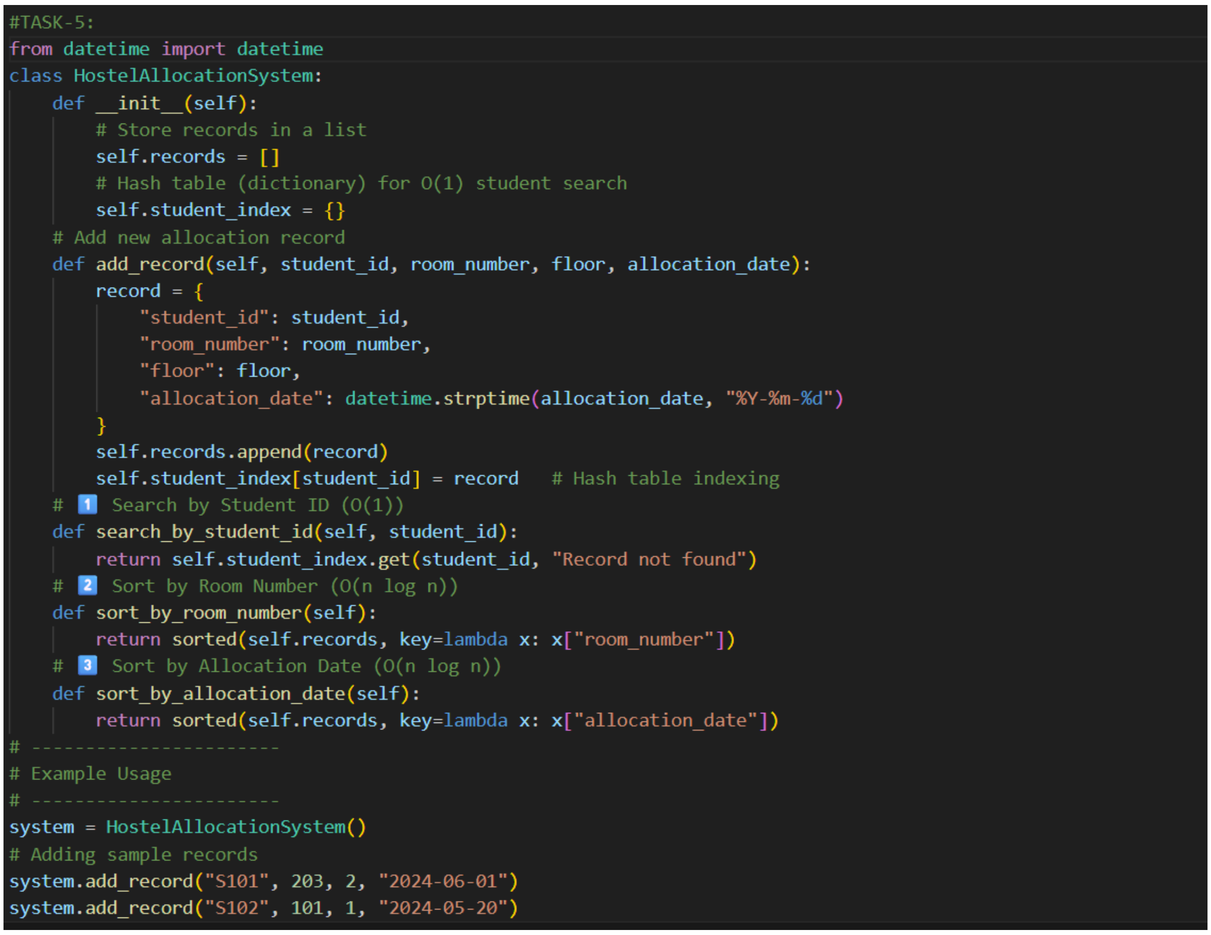


```
🔍 Search Ticket ID 502:
{'ticket_id': 502, 'passenger': 'Karthik', 'train_no': 12627, 'seat_no': 30, 'travel_date': datetime.datetime(2026, 3, 9, 0, 0)}

📅 Bookings Sorted by Travel Date:
{'ticket_id': 503, 'passenger': 'Anita', 'train_no': 12007, 'seat_no': 12, 'travel_date': datetime.datetime(2026, 3, 8, 0, 0)}
{'ticket_id': 502, 'passenger': 'Karthik', 'train_no': 12627, 'seat_no': 30, 'travel_date': datetime.datetime(2026, 3, 9, 0, 0)}
{'ticket_id': 501, 'passenger': 'Ravi', 'train_no': 12627, 'seat_no': 45, 'travel_date': datetime.datetime(2026, 3, 10, 0, 0)}

💺 Bookings Sorted by Seat Number:
{'ticket_id': 503, 'passenger': 'Anita', 'train_no': 12007, 'seat_no': 12, 'travel_date': datetime.datetime(2026, 3, 8, 0, 0)}
{'ticket_id': 502, 'passenger': 'Karthik', 'train_no': 12627, 'seat_no': 30, 'travel_date': datetime.datetime(2026, 3, 9, 0, 0)}
{'ticket_id': 501, 'passenger': 'Ravi', 'train_no': 12627, 'seat_no': 45, 'travel_date': datetime.datetime(2026, 3, 10, 0, 0)}
```

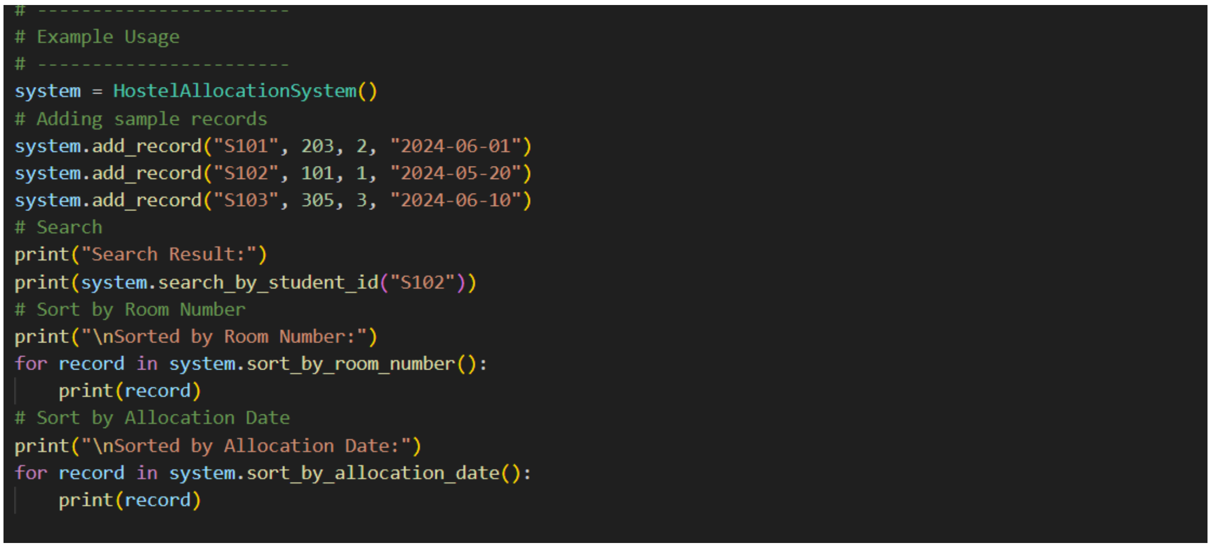
JUSTIFICATION:

This code manages railway booking records by first converting travel dates into datetime objects for accurate comparison and sorting. It uses Merge Sort to sort bookings based on different keys like ticket ID, travel date, and seat number in a stable and efficient manner with O(n log n) time complexity. The Binary Search function is then applied on the ticket ID–sorted list to quickly locate a specific booking in O(log n) time. Overall, the program demonstrates efficient data organization and fast searching by combining sorting and searching algorithms on structured booking data.

TASK-5: Smart Hostel Room Allocation System

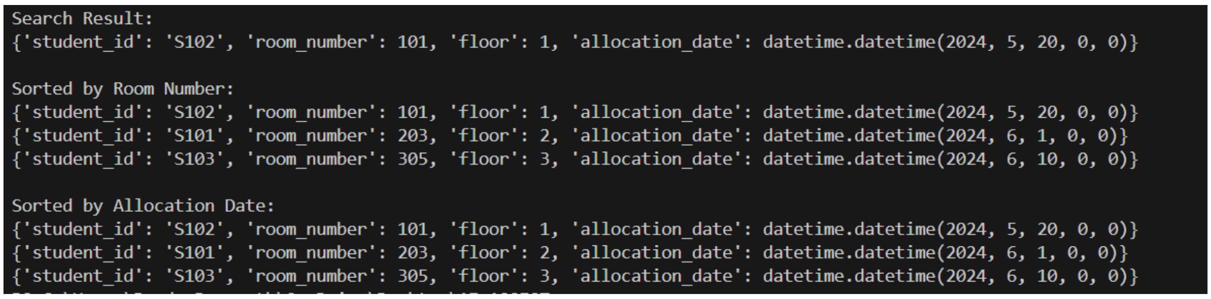


```
#TASK-5:
from datetime import datetime
class HostelAllocationSystem:
    def __init__(self):
        # Store records in a list
        self.records = []
        # Hash table (dictionary) for O(1) student search
        self.student_index = {}
    # Add new allocation record
    def add_record(self, student_id, room_number, floor, allocation_date):
        record = {
            "student_id": student_id,
            "room_number": room_number,
            "floor": floor,
            "allocation_date": datetime.strptime(allocation_date, "%Y-%m-%d")
        }
        self.records.append(record)
        self.student_index[student_id] = record   # Hash table indexing
    # 1️⃣ Search by Student ID (O(1))
    def search_by_student_id(self, student_id):
        return self.student_index.get(student_id, "Record not found")
    # 2️⃣ Sort by Room Number (O(n log n))
    def sort_by_room_number(self):
        return sorted(self.records, key=lambda x: x["room_number"])
    # 3️⃣ Sort by Allocation Date (O(n log n))
    def sort_by_allocation_date(self):
        return sorted(self.records, key=lambda x: x["allocation_date"])
# -----------------------
# Example Usage
# -----------------------
system = HostelAllocationSystem()
# Adding sample records
system.add_record("S101", 203, 2, "2024-06-01")
system.add_record("S102", 101, 1, "2024-05-20")
```



```python
# -----------------------
# Example Usage
# -----------------------
system = HostelAllocationSystem()
# Adding sample records
system.add_record("S101", 203, 2, "2024-06-01")
system.add_record("S102", 101, 1, "2024-05-20")
system.add_record("S103", 305, 3, "2024-06-10")
# Search
print("Search Result:")
print(system.search_by_student_id("S102"))
# Sort by Room Number
print("\nSorted by Room Number:")
for record in system.sort_by_room_number():
    print(record)
# Sort by Allocation Date
print("\nSorted by Allocation Date:")
for record in system.sort_by_allocation_date():
    print(record)
```

OUTPUT:

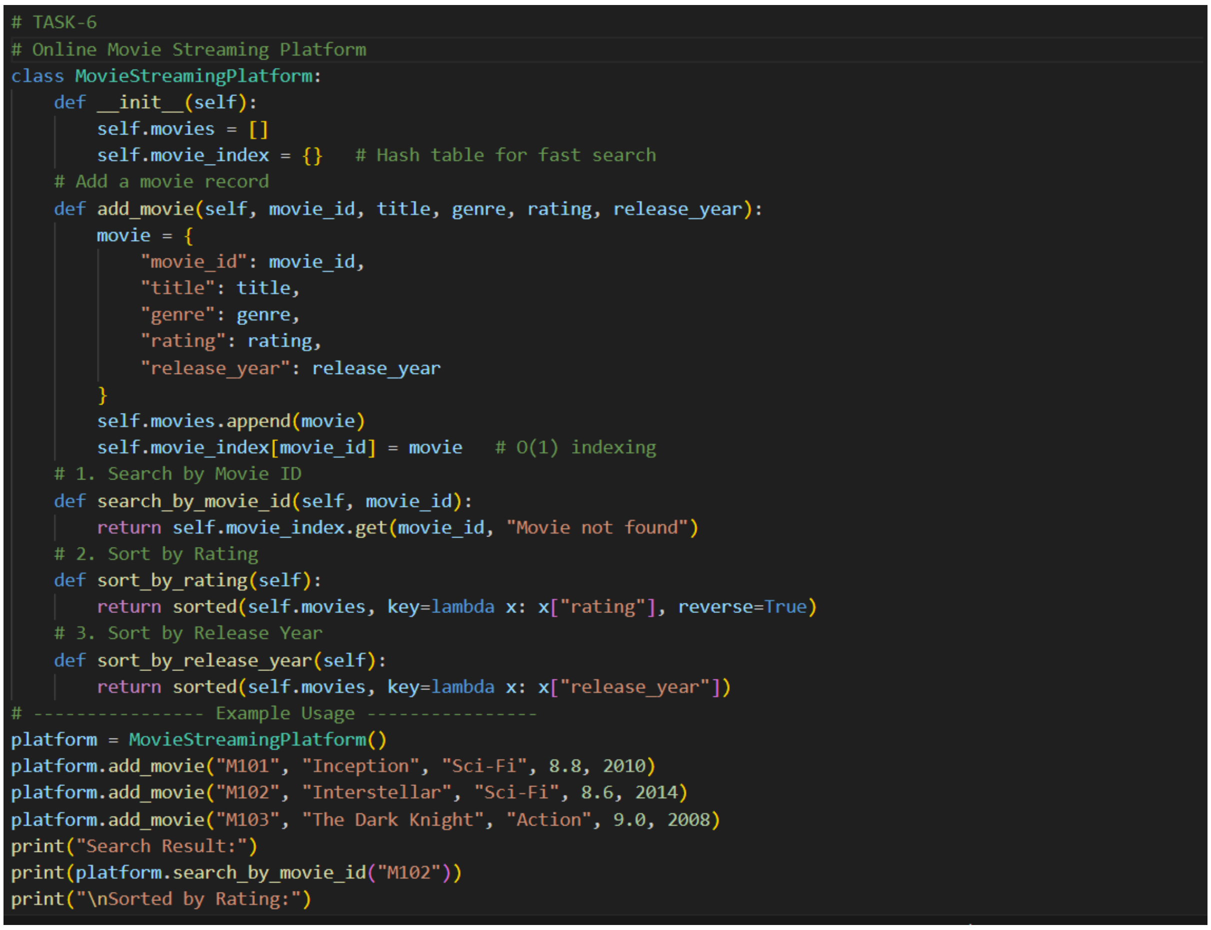


```
Search Result:
{'student_id': 'S102', 'room_number': 101, 'floor': 1, 'allocation_date': datetime.datetime(2024, 5, 20, 0, 0)}

Sorted by Room Number:
{'student_id': 'S102', 'room_number': 101, 'floor': 1, 'allocation_date': datetime.datetime(2024, 5, 20, 0, 0)}
{'student_id': 'S101', 'room_number': 203, 'floor': 2, 'allocation_date': datetime.datetime(2024, 6, 1, 0, 0)}
{'student_id': 'S103', 'room_number': 305, 'floor': 3, 'allocation_date': datetime.datetime(2024, 6, 10, 0, 0)}

Sorted by Allocation Date:
{'student_id': 'S102', 'room_number': 101, 'floor': 1, 'allocation_date': datetime.datetime(2024, 5, 20, 0, 0)}
{'student_id': 'S101', 'room_number': 203, 'floor': 2, 'allocation_date': datetime.datetime(2024, 6, 1, 0, 0)}
{'student_id': 'S103', 'room_number': 305, 'floor': 3, 'allocation_date': datetime.datetime(2024, 6, 10, 0, 0)}
```

JUSTIFICATION:

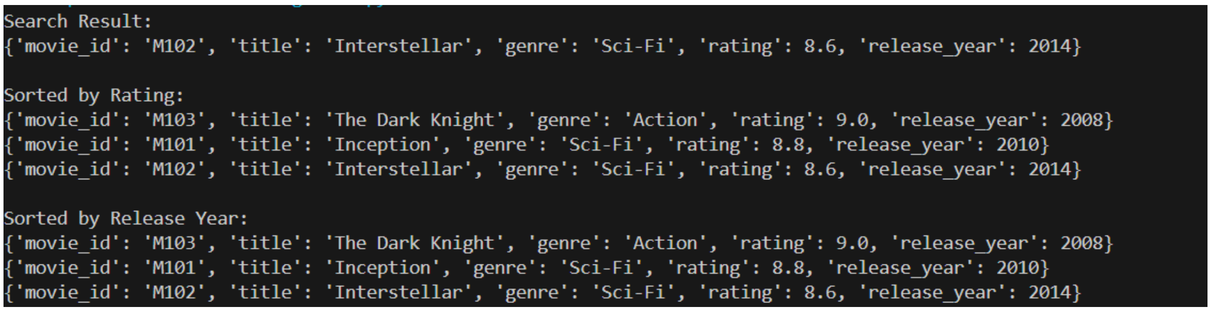
A hash table (dictionary) is used to search student allocation details by student ID becauSe it provides O(1) average time complexity for fast lookups.Timsort (Python’s built-in sorting algorithm) is used to sort records by room number or allocation date with O(n log n) time complexity.These choices ensure the system is efficient, scalable, and suitable for large hostel datasets.

TASK-6: Online Movie Streaming Platform



```python
# TASK-6
# Online Movie Streaming Platform
class MovieStreamingPlatform:
    def __init__(self):
        self.movies = []
        self.movie_index = {}   # Hash table for fast search
    # Add a movie record
    def add_movie(self, movie_id, title, genre, rating, release_year):
        movie = {
            "movie_id": movie_id,
            "title": title,
            "genre": genre,
            "rating": rating,
            "release_year": release_year
        }
        self.movies.append(movie)
        self.movie_index[movie_id] = movie   # O(1) indexing
    # 1. Search by Movie ID
    def search_by_movie_id(self, movie_id):
        return self.movie_index.get(movie_id, "Movie not found")
    # 2. Sort by Rating
    def sort_by_rating(self):
        return sorted(self.movies, key=lambda x: x["rating"], reverse=True)
    # 3. Sort by Release Year
    def sort_by_release_year(self):
        return sorted(self.movies, key=lambda x: x["release_year"])
# ---------------- Example Usage ----------------
platform = MovieStreamingPlatform()
platform.add_movie("M101", "Inception", "Sci-Fi", 8.8, 2010)
platform.add_movie("M102", "Interstellar", "Sci-Fi", 8.6, 2014)
platform.add_movie("M103", "The Dark Knight", "Action", 9.0, 2008)
print("Search Result:")
print(platform.search_by_movie_id("M102"))
print("\nSorted by Rating:")
```

OUTPUT:

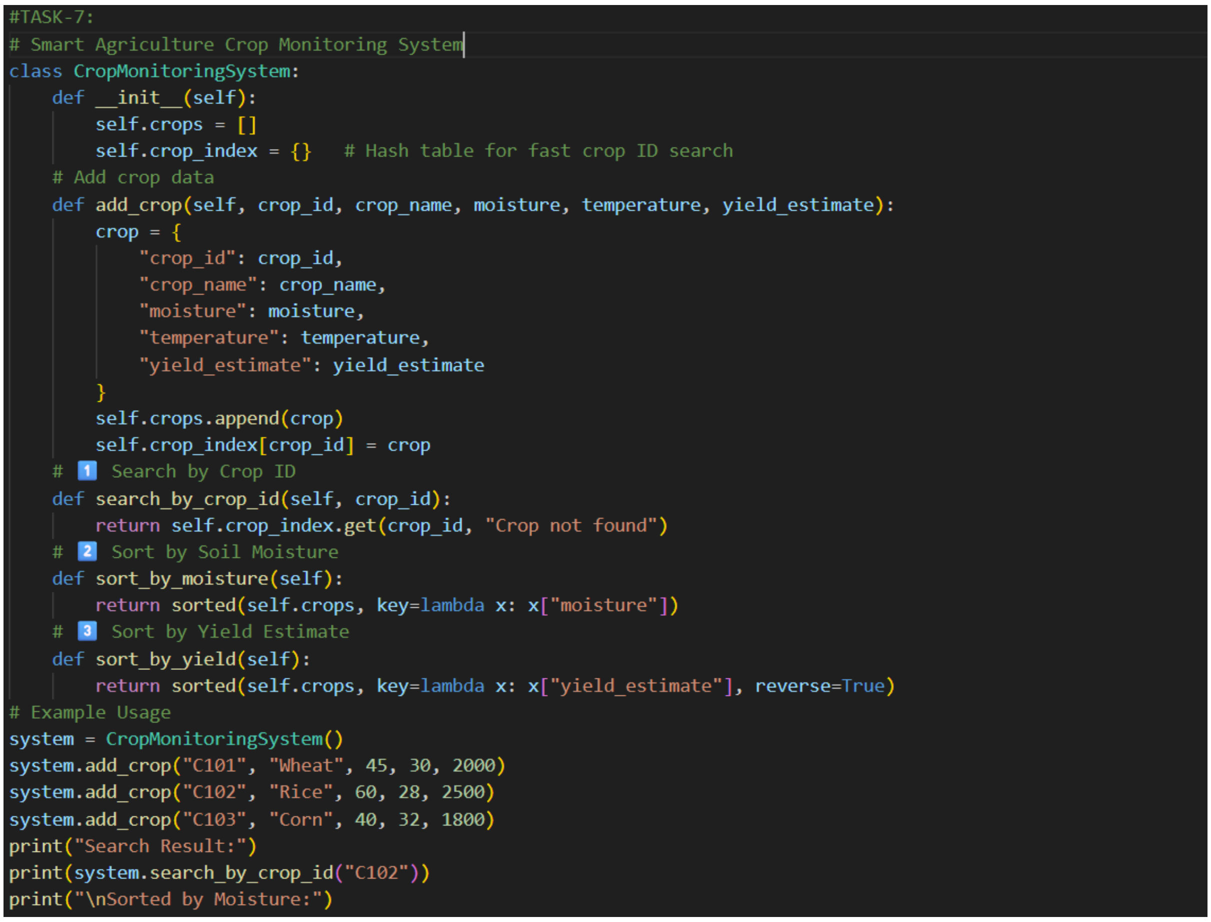


```
Search Result:
{'movie_id': 'M102', 'title': 'Interstellar', 'genre': 'Sci-Fi', 'rating': 8.6, 'release_year': 2014}

Sorted by Rating:
{'movie_id': 'M103', 'title': 'The Dark Knight', 'genre': 'Action', 'rating': 9.0, 'release_year': 2008}
{'movie_id': 'M101', 'title': 'Inception', 'genre': 'Sci-Fi', 'rating': 8.8, 'release_year': 2010}
{'movie_id': 'M102', 'title': 'Interstellar', 'genre': 'Sci-Fi', 'rating': 8.6, 'release_year': 2014}

Sorted by Release Year:
{'movie_id': 'M103', 'title': 'The Dark Knight', 'genre': 'Action', 'rating': 9.0, 'release_year': 2008}
{'movie_id': 'M101', 'title': 'Inception', 'genre': 'Sci-Fi', 'rating': 8.8, 'release_year': 2010}
{'movie_id': 'M102', 'title': 'Interstellar', 'genre': 'Sci-Fi', 'rating': 8.6, 'release_year': 2014}
```

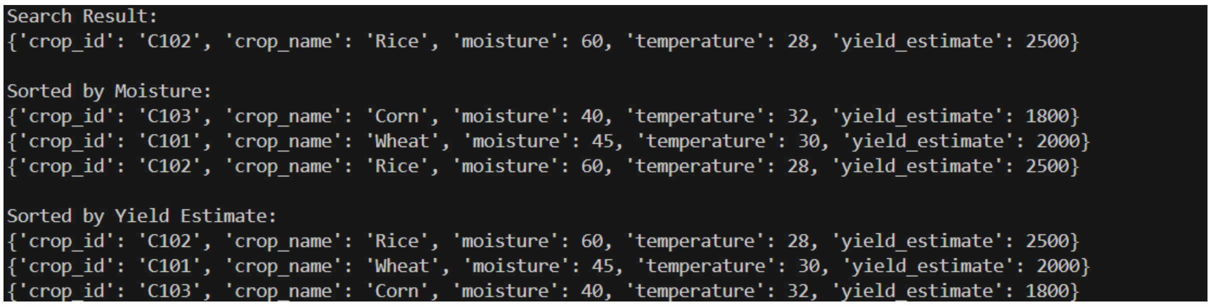
JUSTIFICATION:

This code models an online movie streaming platform using a class-based structure to manage movie records efficiently. It uses a hash table (movie_index) to enable fast O(1) search by movie ID, improving performance compared to linear search. The built-in sorted() function is used to organize movies by rating and release year, ensuring clear and efficient data sorting. Overall, the program demonstrates effective data storage, quick retrieval, and flexible sorting, making it suitable for real-world streaming platform management.

TASK-7: Smart Agriculture Crop Monitoring System



```python
#TASK-7:
# Smart Agriculture Crop Monitoring System
class CropMonitoringSystem:
    def __init__(self):
        self.crops = []
        self.crop_index = {}   # Hash table for fast crop ID search
    # Add crop data
    def add_crop(self, crop_id, crop_name, moisture, temperature, yield_estimate):
        crop = {
            "crop_id": crop_id,
            "crop_name": crop_name,
            "moisture": moisture,
            "temperature": temperature,
            "yield_estimate": yield_estimate
        }
        self.crops.append(crop)
        self.crop_index[crop_id] = crop
    # 1️⃣ Search by Crop ID
    def search_by_crop_id(self, crop_id):
        return self.crop_index.get(crop_id, "Crop not found")
    # 2️⃣ Sort by Soil Moisture
    def sort_by_moisture(self):
        return sorted(self.crops, key=lambda x: x["moisture"])
    # 3️⃣ Sort by Yield Estimate
    def sort_by_yield(self):
        return sorted(self.crops, key=lambda x: x["yield_estimate"], reverse=True)
# Example Usage
system = CropMonitoringSystem()
system.add_crop("C101", "Wheat", 45, 30, 2000)
system.add_crop("C102", "Rice", 60, 28, 2500)
system.add_crop("C103", "Corn", 40, 32, 1800)
print("Search Result:")
print(system.search_by_crop_id("C102"))
print("\nSorted by Moisture:")
```

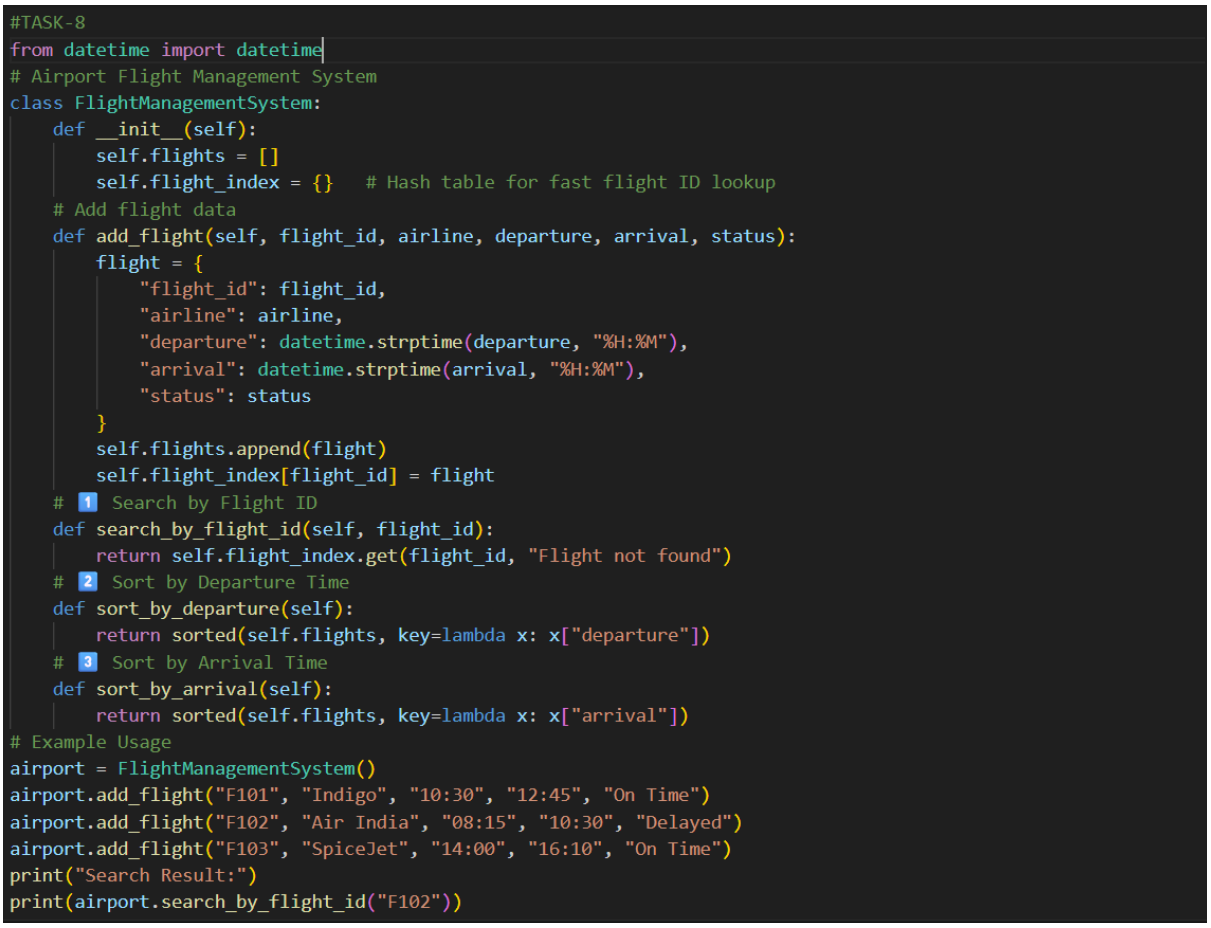


```
Search Result:
{'crop_id': 'C102', 'crop_name': 'Rice', 'moisture': 60, 'temperature': 28, 'yield_estimate': 2500}

Sorted by Moisture:
{'crop_id': 'C103', 'crop_name': 'Corn', 'moisture': 40, 'temperature': 32, 'yield_estimate': 1800}
{'crop_id': 'C101', 'crop_name': 'Wheat', 'moisture': 45, 'temperature': 30, 'yield_estimate': 2000}
{'crop_id': 'C102', 'crop_name': 'Rice', 'moisture': 60, 'temperature': 28, 'yield_estimate': 2500}

Sorted by Yield Estimate:
{'crop_id': 'C102', 'crop_name': 'Rice', 'moisture': 60, 'temperature': 28, 'yield_estimate': 2500}
{'crop_id': 'C101', 'crop_name': 'Wheat', 'moisture': 45, 'temperature': 30, 'yield_estimate': 2000}
{'crop_id': 'C103', 'crop_name': 'Corn', 'moisture': 40, 'temperature': 32, 'yield_estimate': 1800}
```

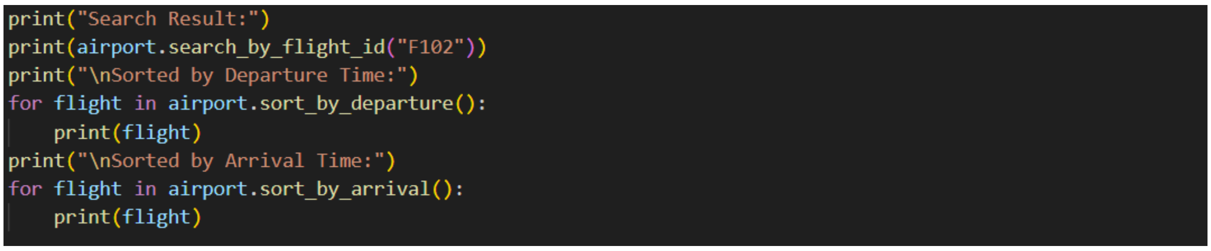
JUSTIFICATION:

A hash table is used because crop IDs are unique and require fast retrieval, giving O(1) average search time.Timsort is used for sorting moisture and yield values because it is stable and performs efficiently with O(n log n) complexity.This ensures real-time monitoring and efficient analysis of agricultural data.

TASK-8: Airport Flight Management System

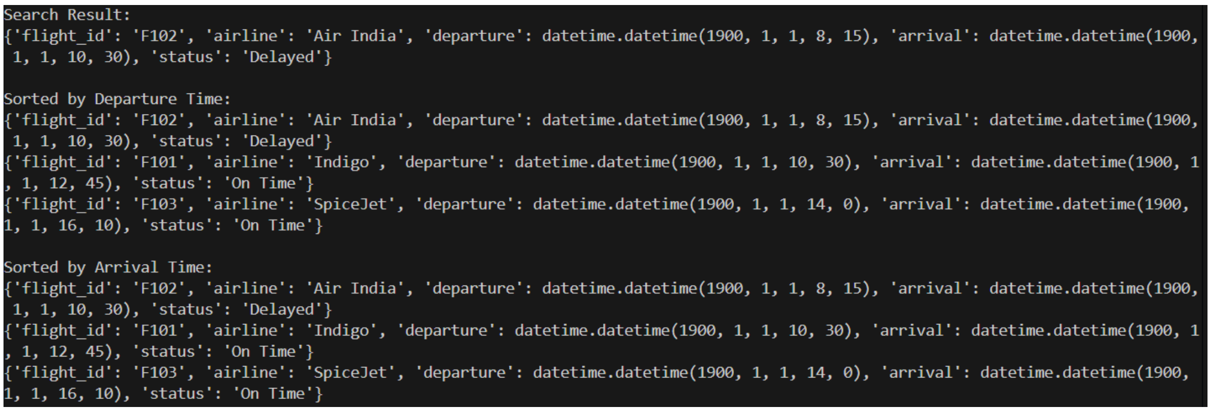


```python
#TASK-8
from datetime import datetime
# Airport Flight Management System
class FlightManagementSystem:
    def __init__(self):
        self.flights = []
        self.flight_index = {}   # Hash table for fast flight ID lookup
    # Add flight data
    def add_flight(self, flight_id, airline, departure, arrival, status):
        flight = {
            "flight_id": flight_id,
            "airline": airline,
            "departure": datetime.strptime(departure, "%H:%M"),
            "arrival": datetime.strptime(arrival, "%H:%M"),
            "status": status
        }
        self.flights.append(flight)
        self.flight_index[flight_id] = flight
    # 1️⃣ Search by Flight ID
    def search_by_flight_id(self, flight_id):
        return self.flight_index.get(flight_id, "Flight not found")
    # 2️⃣ Sort by Departure Time
    def sort_by_departure(self):
        return sorted(self.flights, key=lambda x: x["departure"])
    # 3️⃣ Sort by Arrival Time
    def sort_by_arrival(self):
        return sorted(self.flights, key=lambda x: x["arrival"])
# Example Usage
airport = FlightManagementSystem()
airport.add_flight("F101", "Indigo", "10:30", "12:45", "On Time")
airport.add_flight("F102", "Air India", "08:15", "10:30", "Delayed")
airport.add_flight("F103", "SpiceJet", "14:00", "16:10", "On Time")
print("Search Result:")
print(airport.search_by_flight_id("F102"))
```



```python
print("Search Result:")
print(airport.search_by_flight_id("F102"))
print("\nSorted by Departure Time:")
for flight in airport.sort_by_departure():
    print(flight)
print("\nSorted by Arrival Time:")
for flight in airport.sort_by_arrival():
    print(flight)
```

OUTPUT:



```
Search Result:
{'flight_id': 'F102', 'airline': 'Air India', 'departure': datetime.datetime(1900, 1, 1, 8, 15), 'arrival': datetime.datetime(1900,
 1, 1, 10, 30), 'status': 'Delayed'}

Sorted by Departure Time:
{'flight_id': 'F102', 'airline': 'Air India', 'departure': datetime.datetime(1900, 1, 1, 8, 15), 'arrival': datetime.datetime(1900,
 1, 1, 10, 30), 'status': 'Delayed'}
{'flight_id': 'F101', 'airline': 'Indigo', 'departure': datetime.datetime(1900, 1, 1, 10, 30), 'arrival': datetime.datetime(1900, 1
, 1, 12, 45), 'status': 'On Time'}
{'flight_id': 'F103', 'airline': 'SpiceJet', 'departure': datetime.datetime(1900, 1, 1, 14, 0), 'arrival': datetime.datetime(1900,
1, 1, 16, 10), 'status': 'On Time'}

Sorted by Arrival Time:
{'flight_id': 'F102', 'airline': 'Air India', 'departure': datetime.datetime(1900, 1, 1, 8, 15), 'arrival': datetime.datetime(1900,
 1, 1, 10, 30), 'status': 'Delayed'}
{'flight_id': 'F101', 'airline': 'Indigo', 'departure': datetime.datetime(1900, 1, 1, 10, 30), 'arrival': datetime.datetime(1900, 1
, 1, 12, 45), 'status': 'On Time'}
{'flight_id': 'F103', 'airline': 'SpiceJet', 'departure': datetime.datetime(1900, 1, 1, 14, 0), 'arrival': datetime.datetime(1900,
1, 1, 16, 10), 'status': 'On Time'}
```

JUSTIFICATION:

A dictionary-based indexing approach is selected to enable instant lookup of flight records using unique flight IDs.Sorting is implemented using Python's optimized Timsort, which guarantees efficient time-based ordering with O(n log n) complexity.Datetime conversion ensures accurate chronological comparison rather than simple string-based sorting, improving correctness in real-world scheduling systems.