ASSIGNMENT_12.5

HT.NO:2303A51717

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.

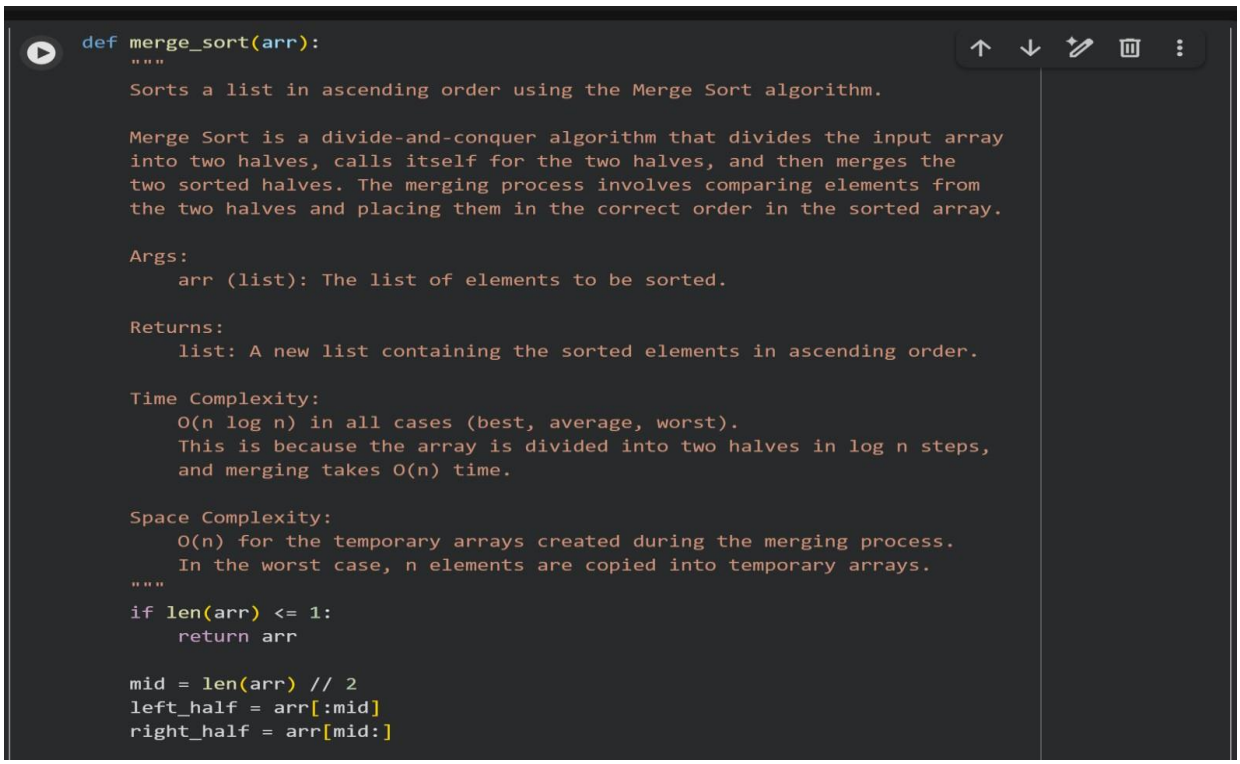
o Verify the generated code with test cases.

• Expected Output:

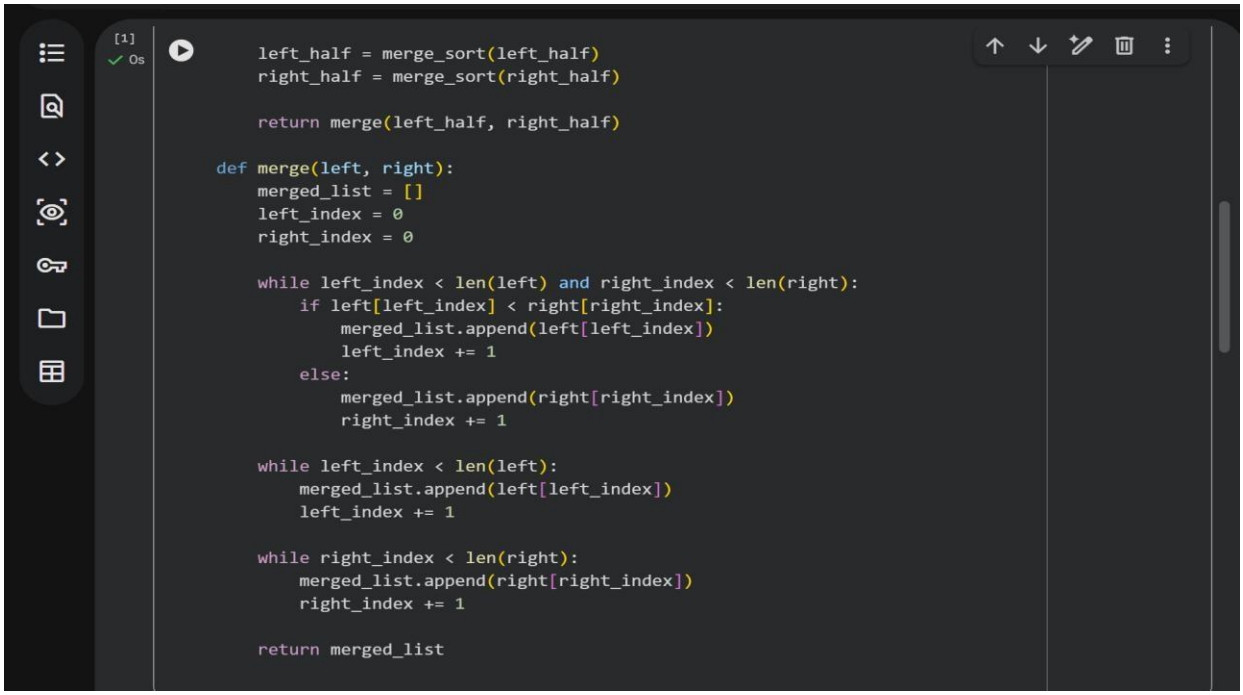
o A functional Python script implementing Merge Sort

with proper documentation.

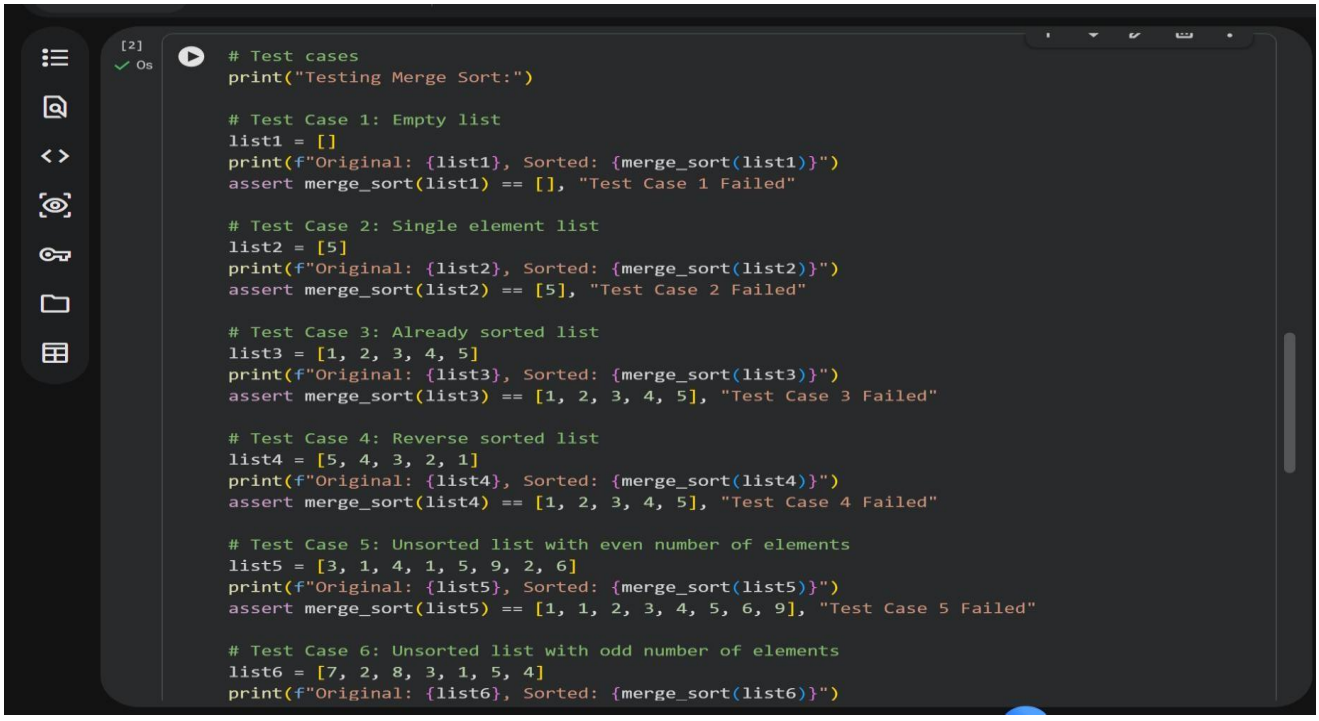
CODE:



```
def merge_sort(arr):
    """
    Sorts a list in ascending order using the Merge Sort algorithm.

    Merge Sort is a divide-and-conquer algorithm that divides the input array
    into two halves, calls itself for the two halves, and then merges the
    two sorted halves. The merging process involves comparing elements from
    the two halves and placing them in the correct order in the sorted array.

    Args:
        arr (list): The list of elements to be sorted.

    Returns:
        list: A new list containing the sorted elements in ascending order.

    Time Complexity:
        O(n log n) in all cases (best, average, worst).
        This is because the array is divided into two halves in log n steps,
        and merging takes O(n) time.

    Space Complexity:
        O(n) for the temporary arrays created during the merging process.
        In the worst case, n elements are copied into temporary arrays.
    """
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left_half = arr[:mid]
    right_half = arr[mid:]
```



```
[1]
✓ 0s

        left_half = merge_sort(left_half)
        right_half = merge_sort(right_half)

        return merge(left_half, right_half)

    def merge(left, right):
        merged_list = []
        left_index = 0
        right_index = 0

        while left_index < len(left) and right_index < len(right):
            if left[left_index] < right[right_index]:
                merged_list.append(left[left_index])
                left_index += 1
            else:
                merged_list.append(right[right_index])
                right_index += 1

        while left_index < len(left):
            merged_list.append(left[left_index])
            left_index += 1

        while right_index < len(right):
            merged_list.append(right[right_index])
            right_index += 1

        return merged_list
```



```
[2]
✓ 0s

# Test cases
print("Testing Merge Sort:")

# Test Case 1: Empty list
list1 = []
print(f"Original: {list1}, Sorted: {merge_sort(list1)}")
assert merge_sort(list1) == [], "Test Case 1 Failed"

# Test Case 2: Single element list
list2 = [5]
print(f"Original: {list2}, Sorted: {merge_sort(list2)}")
assert merge_sort(list2) == [5], "Test Case 2 Failed"

# Test Case 3: Already sorted list
list3 = [1, 2, 3, 4, 5]
print(f"Original: {list3}, Sorted: {merge_sort(list3)}")
assert merge_sort(list3) == [1, 2, 3, 4, 5], "Test Case 3 Failed"

# Test Case 4: Reverse sorted list
list4 = [5, 4, 3, 2, 1]
print(f"Original: {list4}, Sorted: {merge_sort(list4)}")
assert merge_sort(list4) == [1, 2, 3, 4, 5], "Test Case 4 Failed"

# Test Case 5: Unsorted list with even number of elements
list5 = [3, 1, 4, 1, 5, 9, 2, 6]
print(f"Original: {list5}, Sorted: {merge_sort(list5)}")
assert merge_sort(list5) == [1, 1, 2, 3, 4, 5, 6, 9], "Test Case 5 Failed"

# Test Case 6: Unsorted list with odd number of elements
list6 = [7, 2, 8, 3, 1, 5, 4]
print(f"Original: {list6}, Sorted: {merge_sort(list6)}")
```

```
[2]
✓ 0s
# Test Case 6: Unsorted list with odd number of elements
list6 = [7, 2, 8, 3, 1, 5, 4]
print(f"Original: {list6}, Sorted: {merge_sort(list6)}")
assert merge_sort(list6) == [1, 2, 3, 4, 5, 7, 8], "Test Case 6 Failed"

# Test Case 7: List with duplicate elements
list7 = [4, 2, 2, 8, 4, 1, 6]
print(f"Original: {list7}, Sorted: {merge_sort(list7)}")
assert merge_sort(list7) == [1, 2, 2, 4, 4, 6, 8], "Test Case 7 Failed"

# Test Case 8: List with negative numbers
list8 = [-5, -2, -8, 0, -1]
print(f"Original: {list8}, Sorted: {merge_sort(list8)}")
assert merge_sort(list8) == [-8, -5, -2, -1, 0], "Test Case 8 Failed"

print("All test cases passed!")
```

OUTPUT:

```
Testing Merge Sort:
Original: [], Sorted: []
Original: [5], Sorted: [5]
Original: [1, 2, 3, 4, 5], Sorted: [1, 2, 3, 4, 5]
Original: [5, 4, 3, 2, 1], Sorted: [1, 2, 3, 4, 5]
Original: [3, 1, 4, 1, 5, 9, 2, 6], Sorted: [1, 1, 2, 3, 4, 5, 6, 9]
Original: [7, 2, 8, 3, 1, 5, 4], Sorted: [1, 2, 3, 4, 5, 7, 8]
Original: [4, 2, 2, 8, 4, 1, 6], Sorted: [1, 2, 2, 4, 4, 6, 8]
Original: [-5, -2, -8, 0, -1], Sorted: [-8, -5, -2, -1, 0]
All test cases passed!
```

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:

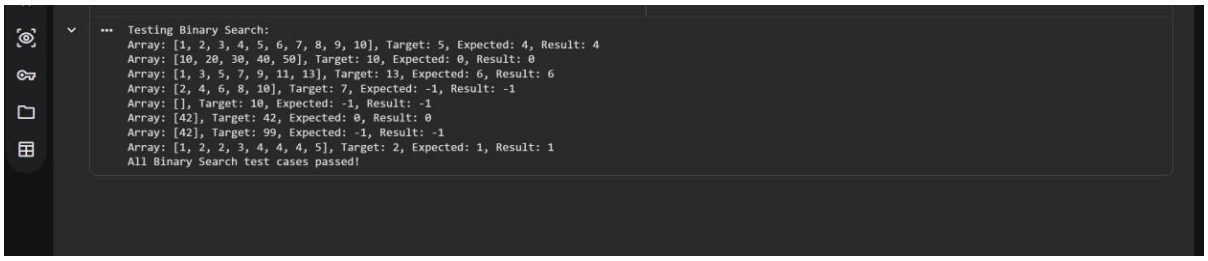
```python
def binary_search(arr, target):
    """
    Searches for a target element in a sorted list using the Binary Search algorithm.

    Binary Search is an efficient algorithm for finding an item from a sorted list
    of items. It works by repeatedly dividing in half the portion of the list that
    could contain the item, until you've narrowed down the possible locations to
    just one.

    Args:
        arr (list): A sorted list of elements (in ascending order) to search within.
        target: The element to search for.

    Returns:
        int: The index of the target element if found, otherwise -1.

    Time Complexity:
        Best Case: O(1) - when the target element is the middle element of the array.
        Average Case: O(log n) - the search space is halved in each step.
        Worst Case: O(log n) - when the target element is at one of the ends or not present.

    Space Complexity:
        O(1) - for iterative implementation, as it uses a constant amount of extra space.
        O(log n) - for recursive implementation, due to recursion stack space.
    """
    low = 0
    high = len(arr) - 1

    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
```

```python
            return mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1

    return -1
```

```python
# Test cases for binary_search
print("Testing Binary Search:")

# Test Case 1: Target in the middle
arr1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
target1 = 5
expected1 = 4
result1 = binary_search(arr1, target1)
print(f"Array: {arr1}, Target: {target1}, Expected: {expected1}, Result: {result1}")
assert result1 == expected1, f"Test Case 1 Failed: Expected {expected1}, Got {result1}"

# Test Case 2: Target at the beginning
arr2 = [10, 20, 30, 40, 50]
target2 = 10
expected2 = 0
result2 = binary_search(arr2, target2)
print(f"Array: {arr2}, Target: {target2}, Expected: {expected2}, Result: {result2}")
assert result2 == expected2, f"Test Case 2 Failed: Expected {expected2}, Got {result2}"

# Test Case 3: Target at the end
arr3 = [1, 3, 5, 7, 9, 11, 13]
target3 = 13
```

```python
assert result4 == expected4, f"Test Case 4 Failed: Expected {expected4}, Got {result4}"

# Test Case 5: Empty array
arr5 = []
target5 = 10
expected5 = -1
result5 = binary_search(arr5, target5)
print(f"Array: {arr5}, Target: {target5}, Expected: {expected5}, Result: {result5}")
assert result5 == expected5, f"Test Case 5 Failed: Expected {expected5}, Got {result5}"

# Test Case 6: Single element array - target found
arr6 = [42]
target6 = 42
expected6 = 0
result6 = binary_search(arr6, target6)
print(f"Array: {arr6}, Target: {target6}, Expected: {expected6}, Result: {result6}")
assert result6 == expected6, f"Test Case 6 Failed: Expected {expected6}, Got {result6}"

# Test Case 7: Single element array - target not found
arr7 = [42]
target7 = 99
expected7 = -1
result7 = binary_search(arr7, target7)
print(f"Array: {arr7}, Target: {target7}, Expected: {expected7}, Result: {result7}")
assert result7 == expected7, f"Test Case 7 Failed: Expected {expected7}, Got {result7}"

# Test Case 8: Array with duplicate elements (first occurrence)
arr8 = [1, 2, 2, 3, 4, 4, 4, 5]
target8 = 2
# Binary search typically finds *an* index, not necessarily the first or last.
# Here, it should find index 1 or 2 depending on implementation; our iterative
# implementation finds the first valid match.
```

OUTPUT:



Task Description #3: Smart Healthcare Appointment Scheduling

System

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.

2. Sort appointments based on time or consultation fee.

Student Task

• Use AI to recommend suitable searching and sorting

algorithms.

• Justify the selected algorithms.

• Implement the algorithms in Python.

CODE:

Recommended Algorithms and Justifications

1. Searching Appointments by ID

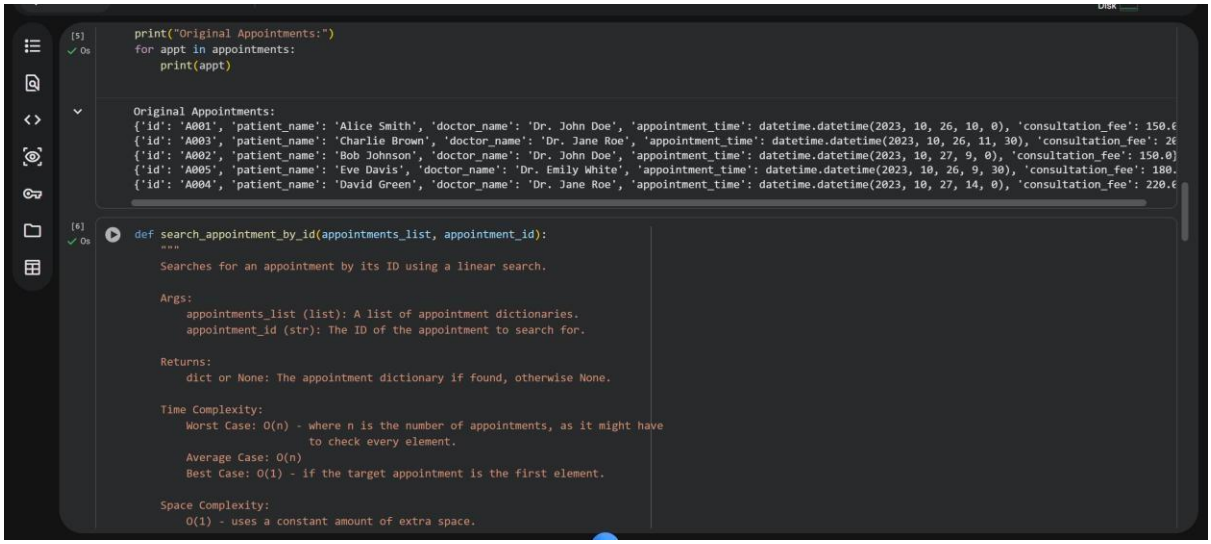
- **Algorithm:** Linear Search
- **Justification:** For a list of appointment records that are not necessarily sorted by appointment_id, a linear search is the most straightforward approach. It iterates through each appointment until the matching ID is found. While its time complexity is O(n) in the worst case (where 'n' is the number of appointments), it's simple to implement and sufficient for moderately sized lists or when the data structure doesn't guarantee sorting by ID. For a real-world system requiring very fast lookups on large datasets, a hash map (Python dictionary) that maps appointment_id to the appointment object would provide an average time complexity of O(1). However, since the task implies working with a list, linear search is chosen for direct list iteration.

2. Sorting Appointments by Time or Consultation Fee

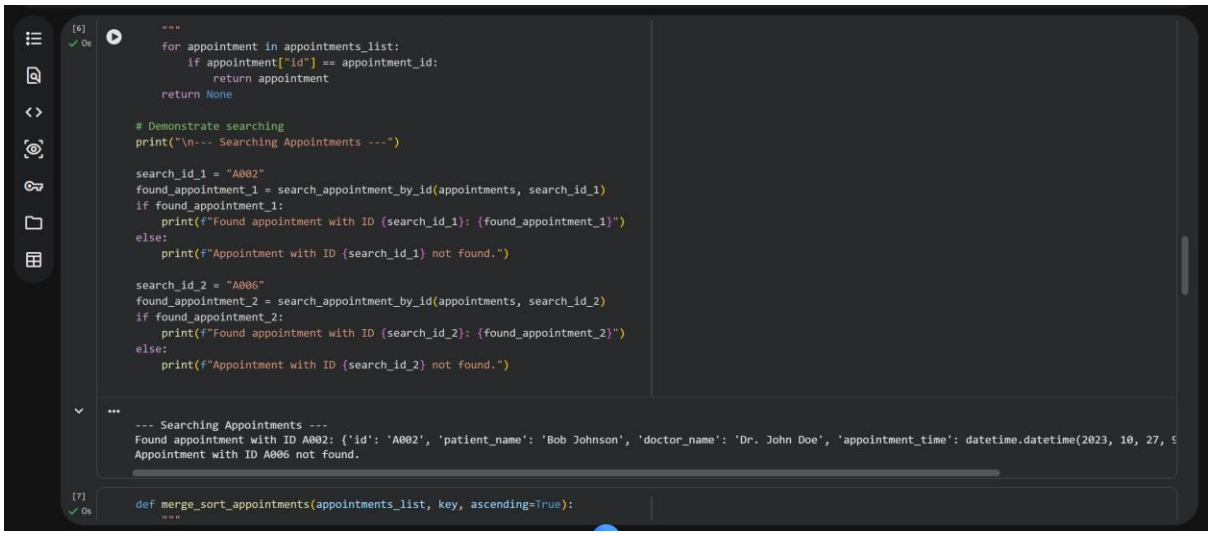
- **Algorithm:** Merge Sort

- **Justification:** Merge Sort is an excellent choice for sorting appointments based on either time or consultation fee for several reasons:
    - **Time Complexity:** It consistently offers O(n log n) time complexity in all cases (best, average, worst). This makes it highly efficient for larger datasets, unlike Quick Sort which can degrade to O(n^2) in the worst case.
    - **Stability:** Merge Sort is a stable sorting algorithm, meaning that if two appointments have the same time or consultation fee, their relative order in the original list is preserved in the sorted list. This can be important for certain business logic (e.g., first-come, first-served for appointments with the same time).
    - **Predictable Performance:** Its consistent O(n log n) performance makes it reliable for critical system functions where predictable execution time is valued.

```
[5]
# Sample Appointment Data
appointments = [
    {
        "id": "A001",
        "patient_name": "Alice Smith",
        "doctor_name": "Dr. John Doe",
        "appointment_time": datetime(2023, 10, 26, 10, 0),
        "consultation_fee": 150.00,
    },
    {
        "id": "A003",
        "patient_name": "Charlie Brown",
        "doctor_name": "Dr. Jane Roe",
        "appointment_time": datetime(2023, 10, 26, 11, 30),
        "consultation_fee": 200.00,
    },
    {
        "id": "A002",
        "patient_name": "Bob Johnson",
        "doctor_name": "Dr. John Doe",
        "appointment_time": datetime(2023, 10, 27, 9, 0),
        "consultation_fee": 150.00,
    },
    {
        "id": "A005",
        "patient_name": "Eve Davis",
        "doctor_name": "Dr. Emily White",
        "appointment_time": datetime(2023, 10, 26, 9, 30),
        "consultation_fee": 180.00,
    },
```



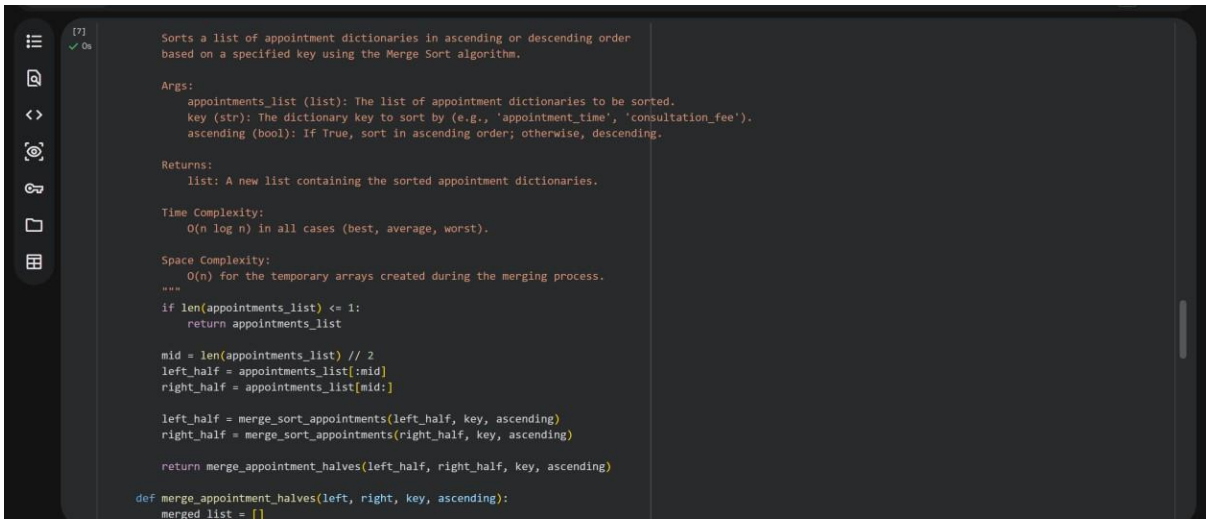
```
[5]
print("Original Appointments:")
for appt in appointments:
    print(appt)

Original Appointments:
{'id': 'A001', 'patient_name': 'Alice Smith', 'doctor_name': 'Dr. John Doe', 'appointment_time': datetime.datetime(2023, 10, 26, 10, 0), 'consultation_fee': 150.0
{'id': 'A003', 'patient_name': 'Charlie Brown', 'doctor_name': 'Dr. Jane Roe', 'appointment_time': datetime.datetime(2023, 10, 26, 11, 30), 'consultation_fee': 20
{'id': 'A002', 'patient_name': 'Bob Johnson', 'doctor_name': 'Dr. John Doe', 'appointment_time': datetime.datetime(2023, 10, 27, 9, 0), 'consultation_fee': 150.0}
{'id': 'A005', 'patient_name': 'Eve Davis', 'doctor_name': 'Dr. Emily White', 'appointment_time': datetime.datetime(2023, 10, 26, 9, 30), 'consultation_fee': 180.
{'id': 'A004', 'patient_name': 'David Green', 'doctor_name': 'Dr. Jane Roe', 'appointment_time': datetime.datetime(2023, 10, 27, 14, 0), 'consultation_fee': 220.0

[6]
def search_appointment_by_id(appointments_list, appointment_id):
    """
    Searches for an appointment by its ID using a linear search.

    Args:
        appointments_list (list): A list of appointment dictionaries.
        appointment_id (str): The ID of the appointment to search for.

    Returns:
        dict or None: The appointment dictionary if found, otherwise None.

    Time Complexity:
        Worst Case: O(n) - where n is the number of appointments, as it might have
                           to check every element.
        Average Case: O(n)
        Best Case: O(1) - if the target appointment is the first element.

    Space Complexity:
        O(1) - uses a constant amount of extra space.
```



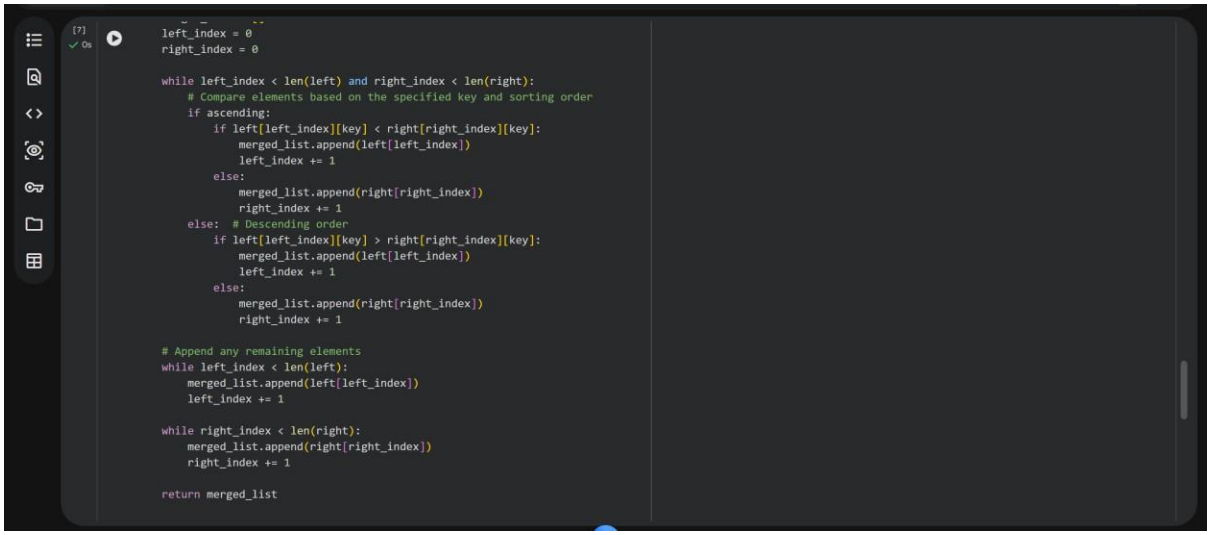
```
[6]
    """
    for appointment in appointments_list:
        if appointment["id"] == appointment_id:
            return appointment
    return None

# Demonstrate searching
print("\n--- Searching Appointments ---")

search_id_1 = "A002"
found_appointment_1 = search_appointment_by_id(appointments, search_id_1)
if found_appointment_1:
    print(f"Found appointment with ID {search_id_1}: {found_appointment_1}")
else:
    print(f"Appointment with ID {search_id_1} not found.")

search_id_2 = "A006"
found_appointment_2 = search_appointment_by_id(appointments, search_id_2)
if found_appointment_2:
    print(f"Found appointment with ID {search_id_2}: {found_appointment_2}")
else:
    print(f"Appointment with ID {search_id_2} not found.")
```

```
--- Searching Appointments ---
Found appointment with ID A002: {'id': 'A002', 'patient_name': 'Bob Johnson', 'doctor_name': 'Dr. John Doe', 'appointment_time': datetime.datetime(2023, 10, 27, 9
Appointment with ID A006 not found.
```

```
[7]
def merge_sort_appointments(appointments_list, key, ascending=True):
    """
```



```
    Sorts a list of appointment dictionaries in ascending or descending order
    based on a specified key using the Merge Sort algorithm.

    Args:
        appointments_list (list): The list of appointment dictionaries to be sorted.
        key (str): The dictionary key to sort by (e.g., 'appointment_time', 'consultation_fee').
        ascending (bool): If True, sort in ascending order; otherwise, descending.

    Returns:
        list: A new list containing the sorted appointment dictionaries.

    Time Complexity:
        O(n log n) in all cases (best, average, worst).

    Space Complexity:
        O(n) for the temporary arrays created during the merging process.
    """
    if len(appointments_list) <= 1:
        return appointments_list

    mid = len(appointments_list) // 2
    left_half = appointments_list[:mid]
    right_half = appointments_list[mid:]

    left_half = merge_sort_appointments(left_half, key, ascending)
    right_half = merge_sort_appointments(right_half, key, ascending)

    return merge_appointment_halves(left_half, right_half, key, ascending)

def merge_appointment_halves(left, right, key, ascending):
    merged_list = []
```



```
[7]
    left_index = 0
    right_index = 0

    while left_index < len(left) and right_index < len(right):
        # Compare elements based on the specified key and sorting order
        if ascending:
            if left[left_index][key] < right[right_index][key]:
                merged_list.append(left[left_index])
                left_index += 1
            else:
                merged_list.append(right[right_index])
                right_index += 1
        else:  # Descending order
            if left[left_index][key] > right[right_index][key]:
                merged_list.append(left[left_index])
                left_index += 1
            else:
                merged_list.append(right[right_index])
                right_index += 1

    # Append any remaining elements
    while left_index < len(left):
        merged_list.append(left[left_index])
        left_index += 1

    while right_index < len(right):
        merged_list.append(right[right_index])
        right_index += 1

    return merged_list
```

```
# Demonstrate sorting by time
print("\n--- Appointments Sorted by Time (Ascending) ---")
sorted_by_time = merge_sort_appointments(appointments, "appointment_time")
for appt in sorted_by_time:
    print(appt)

# Demonstrate sorting by consultation fee (Descending)
print("\n--- Appointments Sorted by Consultation Fee (Descending) ---")
sorted_by_fee_desc = merge_sort_appointments(appointments, "consultation_fee", ascending=False)
for appt in sorted_by_fee_desc:
    print(appt)
```

OUTPUT:

```
--- Appointments Sorted by Time (Ascending) ---
{'id': 'A005', 'patient_name': 'Eve Davis', 'doctor_name': 'Dr. Emily White', 'appointment_time': datetime.datetime(2023, 10, 26, 9, 30), 'consultation_fee': 180.
{'id': 'A001', 'patient_name': 'Alice Smith', 'doctor_name': 'Dr. John Doe', 'appointment_time': datetime.datetime(2023, 10, 26, 10, 0), 'consultation_fee': 150.0
{'id': 'A003', 'patient_name': 'Charlie Brown', 'doctor_name': 'Dr. Jane Roe', 'appointment_time': datetime.datetime(2023, 10, 26, 11, 30), 'consultation_fee': 20
{'id': 'A002', 'patient_name': 'Bob Johnson', 'doctor_name': 'Dr. John Doe', 'appointment_time': datetime.datetime(2023, 10, 27, 9, 0), 'consultation_fee': 150.0}
{'id': 'A004', 'patient_name': 'David Green', 'doctor_name': 'Dr. Jane Roe', 'appointment_time': datetime.datetime(2023, 10, 27, 14, 0), 'consultation_fee': 220.0

--- Appointments Sorted by Consultation Fee (Descending) ---
{'id': 'A004', 'patient_name': 'David Green', 'doctor_name': 'Dr. Jane Roe', 'appointment_time': datetime.datetime(2023, 10, 27, 14, 0), 'consultation_fee': 220.0
{'id': 'A003', 'patient_name': 'Charlie Brown', 'doctor_name': 'Dr. Jane Roe', 'appointment_time': datetime.datetime(2023, 10, 26, 11, 30), 'consultation_fee': 20
{'id': 'A005', 'patient_name': 'Eve Davis', 'doctor_name': 'Dr. Emily White', 'appointment_time': datetime.datetime(2023, 10, 26, 9, 30), 'consultation_fee': 180.
{'id': 'A002', 'patient_name': 'Bob Johnson', 'doctor_name': 'Dr. John Doe', 'appointment_time': datetime.datetime(2023, 10, 27, 9, 0), 'consultation_fee': 150.0}
{'id': 'A001', 'patient_name': 'Alice Smith', 'doctor_name': 'Dr. John Doe', 'appointment_time': datetime.datetime(2023, 10, 26, 10, 0), 'consultation_fee': 150.0
```

Task Description #4: Railway Ticket Reservation System

Scenario

A railway reservation system stores booking details such as ticket

ID, passenger name, train number, seat number, and travel date. The

system must:

1. Search tickets using ticket ID.

2. Sort bookings based on travel date or seat number.

Student Task

• Identify efficient algorithms using AI assistance.

• Justify the algorithm choices.

• Implement searching and sorting in Python.

CODE:

Recommended Algorithms and Justifications for Railway Ticket Reservation System

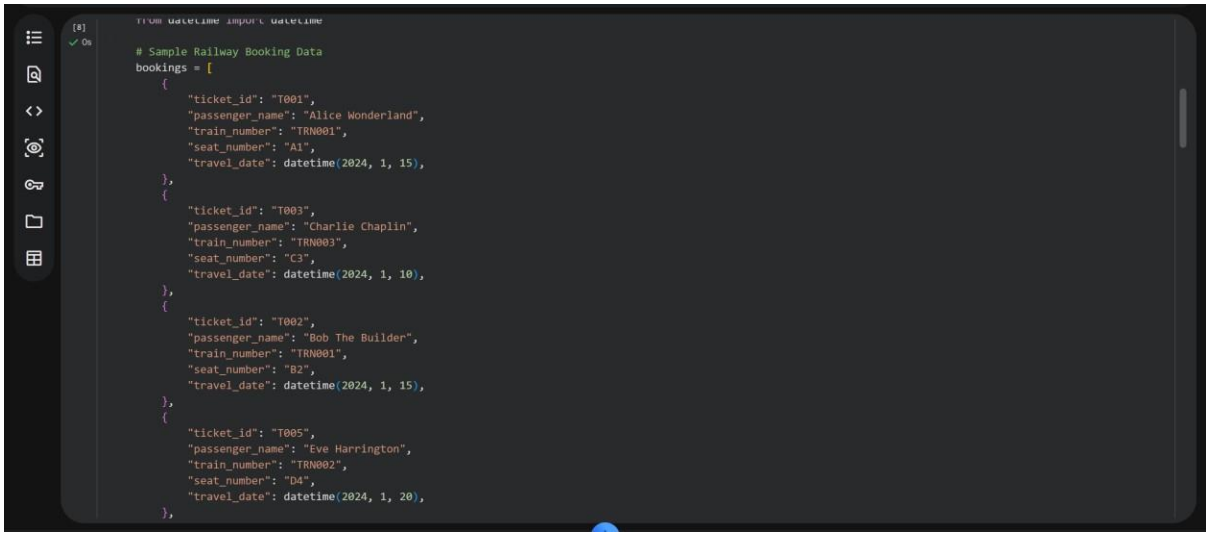
1. Searching Tickets by Ticket ID

- **Algorithm:** Linear Search
- **Justification:** When dealing with a list of booking records where ticket_ids are unique but not necessarily stored in any particular order (e.g., insertion order), a linear search is a practical and easy-to-implement solution. It involves iterating through each booking record until the matching ticket_id is found. Its time complexity is O(n) in the worst case, which is acceptable for moderately sized lists. For extremely large

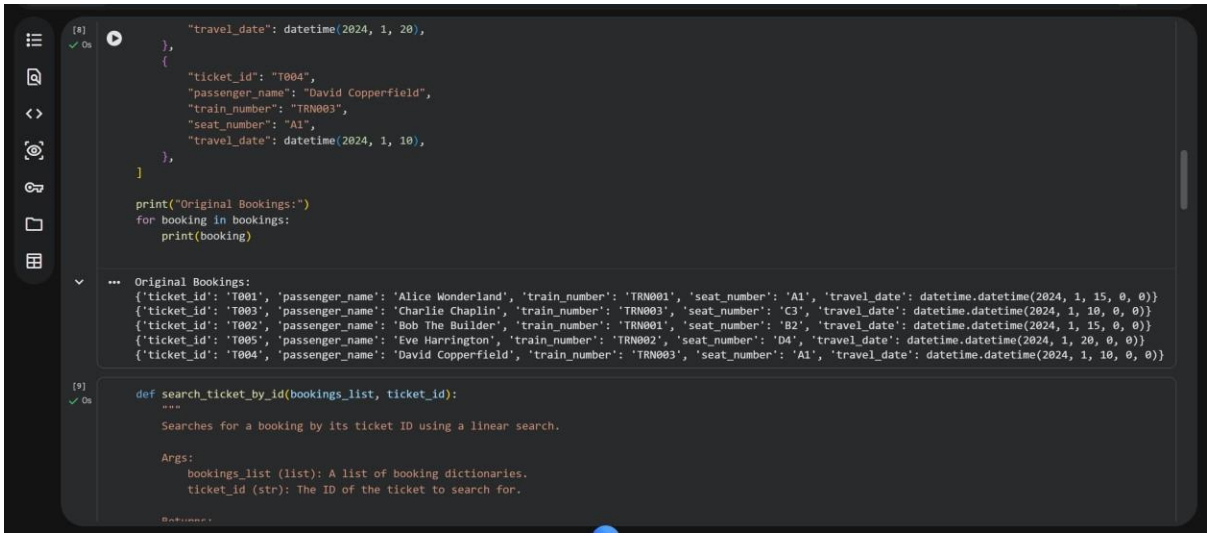
datasets, a hash map (Python dictionary) could offer O(1) average-case lookup, but for a list-based scenario, linear search is a direct fit.

2. Sorting Bookings by Travel Date or Seat Number

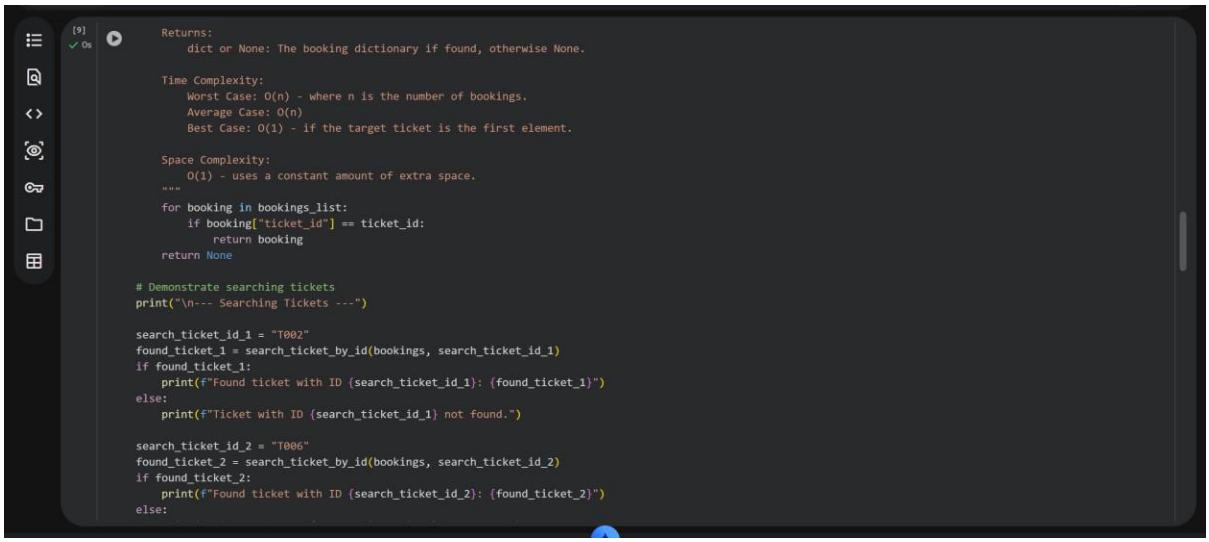
- **Algorithm:** Merge Sort
- **Justification:** Merge Sort is an excellent choice for sorting booking records based on travel_date or seat_number due to its consistent and efficient performance:
    - **Time Complexity:** It guarantees an O(n log n) time complexity in all scenarios (best, average, worst case). This makes it highly scalable and reliable for various sizes of booking data.
    - **Stability:** Merge Sort is a stable sorting algorithm. This property is beneficial here because if two booking records have the same travel_date or seat_number, their original relative order will be preserved in the sorted output. For example, if multiple passengers are on the same date, their original booking order would be maintained, which can be useful for certain system requirements.



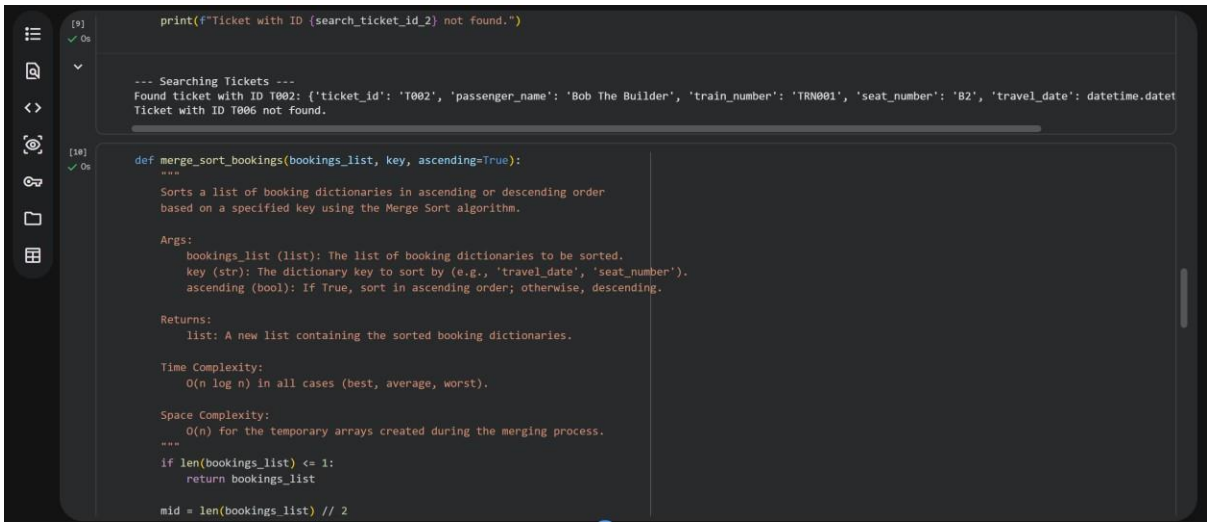
```
[8]
✓ 0s
from datetime import datetime

# Sample Railway Booking Data
bookings = [
    {
        "ticket_id": "T001",
        "passenger_name": "Alice Wonderland",
        "train_number": "TRN001",
        "seat_number": "A1",
        "travel_date": datetime(2024, 1, 15),
    },
    {
        "ticket_id": "T003",
        "passenger_name": "Charlie Chaplin",
        "train_number": "TRN003",
        "seat_number": "C3",
        "travel_date": datetime(2024, 1, 10),
    },
    {
        "ticket_id": "T002",
        "passenger_name": "Bob The Builder",
        "train_number": "TRN001",
        "seat_number": "B2",
        "travel_date": datetime(2024, 1, 15),
    },
    {
        "ticket_id": "T005",
        "passenger_name": "Eve Harrington",
        "train_number": "TRN002",
        "seat_number": "D4",
        "travel_date": datetime(2024, 1, 20),
    },
```



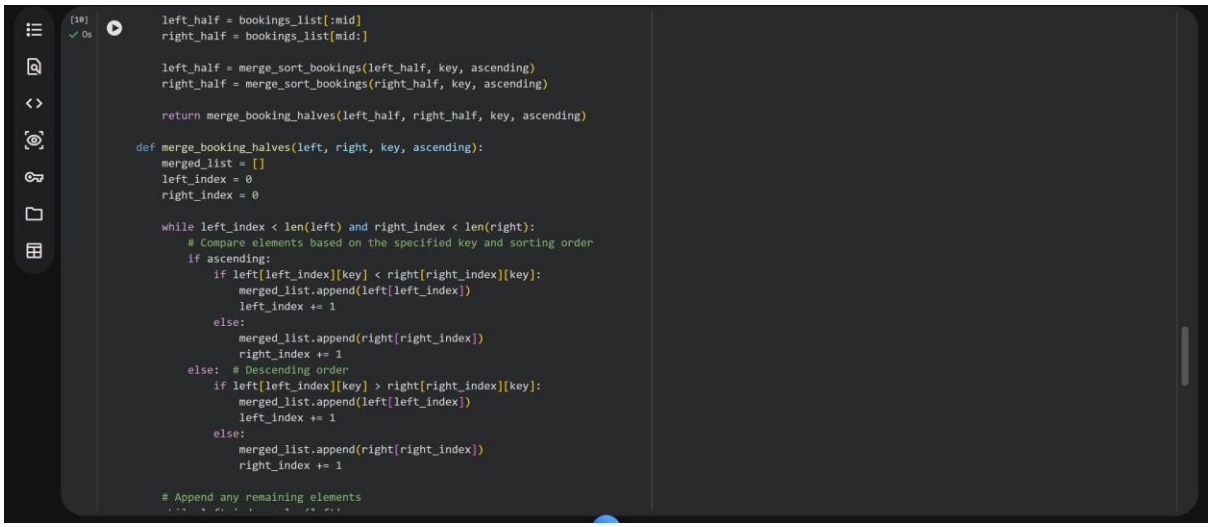
```
[8]
✓ 0s
        "travel_date": datetime(2024, 1, 20),
    },
    {
        "ticket_id": "T004",
        "passenger_name": "David Copperfield",
        "train_number": "TRN003",
        "seat_number": "A1",
        "travel_date": datetime(2024, 1, 10),
    },
]

print("Original Bookings:")
for booking in bookings:
    print(booking)

Original Bookings:
{'ticket_id': 'T001', 'passenger_name': 'Alice Wonderland', 'train_number': 'TRN001', 'seat_number': 'A1', 'travel_date': datetime.datetime(2024, 1, 15, 0, 0)}
{'ticket_id': 'T003', 'passenger_name': 'Charlie Chaplin', 'train_number': 'TRN003', 'seat_number': 'C3', 'travel_date': datetime.datetime(2024, 1, 10, 0, 0)}
{'ticket_id': 'T002', 'passenger_name': 'Bob The Builder', 'train_number': 'TRN001', 'seat_number': 'B2', 'travel_date': datetime.datetime(2024, 1, 15, 0, 0)}
{'ticket_id': 'T005', 'passenger_name': 'Eve Harrington', 'train_number': 'TRN002', 'seat_number': 'D4', 'travel_date': datetime.datetime(2024, 1, 20, 0, 0)}
{'ticket_id': 'T004', 'passenger_name': 'David Copperfield', 'train_number': 'TRN003', 'seat_number': 'A1', 'travel_date': datetime.datetime(2024, 1, 10, 0, 0)}

[9]
✓ 0s
def search_ticket_by_id(bookings_list, ticket_id):
    """
    Searches for a booking by its ticket ID using a linear search.

    Args:
        bookings_list (list): A list of booking dictionaries.
        ticket_id (str): The ID of the ticket to search for.

    Returns:
```



```
[9]
✓ 0s
        Returns:
            dict or None: The booking dictionary if found, otherwise None.

        Time Complexity:
            Worst Case: O(n) - where n is the number of bookings.
            Average Case: O(n)
            Best Case: O(1) - if the target ticket is the first element.

        Space Complexity:
            O(1) - uses a constant amount of extra space.
        """
        for booking in bookings_list:
            if booking["ticket_id"] == ticket_id:
                return booking
        return None

# Demonstrate searching tickets
print("\n--- Searching Tickets ---")

search_ticket_id_1 = "T002"
found_ticket_1 = search_ticket_by_id(bookings, search_ticket_id_1)
if found_ticket_1:
    print(f"Found ticket with ID {search_ticket_id_1}: {found_ticket_1}")
else:
    print(f"Ticket with ID {search_ticket_id_1} not found.")

search_ticket_id_2 = "T006"
found_ticket_2 = search_ticket_by_id(bookings, search_ticket_id_2)
if found_ticket_2:
    print(f"Found ticket with ID {search_ticket_id_2}: {found_ticket_2}")
else:
```



```
[9]
✓ 0s
    print(f"Ticket with ID {search_ticket_id_2} not found.")


--- Searching Tickets ---
Found ticket with ID T002: {'ticket_id': 'T002', 'passenger_name': 'Bob The Builder', 'train_number': 'TRN001', 'seat_number': 'B2', 'travel_date': datetime.datet
Ticket with ID T006 not found.

[10]
✓ 0s
def merge_sort_bookings(bookings_list, key, ascending=True):
    """
    Sorts a list of booking dictionaries in ascending or descending order
    based on a specified key using the Merge Sort algorithm.

    Args:
        bookings_list (list): The list of booking dictionaries to be sorted.
        key (str): The dictionary key to sort by (e.g., 'travel_date', 'seat_number').
        ascending (bool): If True, sort in ascending order; otherwise, descending.

    Returns:
        list: A new list containing the sorted booking dictionaries.

    Time Complexity:
        O(n log n) in all cases (best, average, worst).

    Space Complexity:
        O(n) for the temporary arrays created during the merging process.
    """
    if len(bookings_list) <= 1:
        return bookings_list

    mid = len(bookings_list) // 2
```



```
[10]
✓ 0s
    left_half = bookings_list[:mid]
    right_half = bookings_list[mid:]

    left_half = merge_sort_bookings(left_half, key, ascending)
    right_half = merge_sort_bookings(right_half, key, ascending)

    return merge_booking_halves(left_half, right_half, key, ascending)

def merge_booking_halves(left, right, key, ascending):
    merged_list = []
    left_index = 0
    right_index = 0

    while left_index < len(left) and right_index < len(right):
        # Compare elements based on the specified key and sorting order
        if ascending:
            if left[left_index][key] < right[right_index][key]:
                merged_list.append(left[left_index])
                left_index += 1
            else:
                merged_list.append(right[right_index])
                right_index += 1
        else:  # Descending order
            if left[left_index][key] > right[right_index][key]:
                merged_list.append(left[left_index])
                left_index += 1
            else:
                merged_list.append(right[right_index])
                right_index += 1

    # Append any remaining elements
```

```
        while left_index < len(left):
            merged_list.append(left[left_index])
            left_index += 1

        while right_index < len(right):
            merged_list.append(right[right_index])
            right_index += 1

        return merged_list

    # Demonstrate sorting by travel date (Ascending)
    print("\n--- Bookings Sorted by Travel Date (Ascending) ---")
    sorted_by_date = merge_sort_bookings(bookings, "travel_date")
    for booking in sorted_by_date:
        print(booking)

    # Demonstrate sorting by seat number (Ascending)
    # Note: For seat numbers like 'A1', 'B2', 'C3', sorting as strings works fine.
    # If seat numbers could be 'A1', 'A10', 'A2', a custom sort key might be needed.
    print("\n--- Bookings Sorted by Seat Number (Ascending) ---")
    sorted_by_seat = merge_sort_bookings(bookings, "seat_number", ascending=True)
    for booking in sorted_by_seat:
        print(booking)
```

OUTPUT:

```
--- Bookings Sorted by Travel Date (Ascending) ---
{'ticket_id': 'T004', 'passenger_name': 'David Copperfield', 'train_number': 'TRN003', 'seat_number': 'A1', 'travel_date': datetime.datetime(2024, 1, 10, 0, 0)}
{'ticket_id': 'T003', 'passenger_name': 'Charlie Chaplin', 'train_number': 'TRN003', 'seat_number': 'C3', 'travel_date': datetime.datetime(2024, 1, 10, 0, 0)}
{'ticket_id': 'T002', 'passenger_name': 'Bob The Builder', 'train_number': 'TRN001', 'seat_number': 'B2', 'travel_date': datetime.datetime(2024, 1, 15, 0, 0)}
{'ticket_id': 'T001', 'passenger_name': 'Alice Wonderland', 'train_number': 'TRN001', 'seat_number': 'A1', 'travel_date': datetime.datetime(2024, 1, 15, 0, 0)}
{'ticket_id': 'T005', 'passenger_name': 'Eve Harrington', 'train_number': 'TRN002', 'seat_number': 'D4', 'travel_date': datetime.datetime(2024, 1, 20, 0, 0)}

--- Bookings Sorted by Seat Number (Ascending) ---
{'ticket_id': 'T004', 'passenger_name': 'David Copperfield', 'train_number': 'TRN003', 'seat_number': 'A1', 'travel_date': datetime.datetime(2024, 1, 10, 0, 0)}
{'ticket_id': 'T001', 'passenger_name': 'Alice Wonderland', 'train_number': 'TRN001', 'seat_number': 'A1', 'travel_date': datetime.datetime(2024, 1, 15, 0, 0)}
{'ticket_id': 'T002', 'passenger_name': 'Bob The Builder', 'train_number': 'TRN001', 'seat_number': 'B2', 'travel_date': datetime.datetime(2024, 1, 15, 0, 0)}
{'ticket_id': 'T003', 'passenger_name': 'Charlie Chaplin', 'train_number': 'TRN003', 'seat_number': 'C3', 'travel_date': datetime.datetime(2024, 1, 10, 0, 0)}
{'ticket_id': 'T005', 'passenger_name': 'Eve Harrington', 'train_number': 'TRN002', 'seat_number': 'D4', 'travel_date': datetime.datetime(2024, 1, 20, 0, 0)}
```

Task Description #5: Smart Hostel Room Allocation System

A hostel management system stores student room allocation details

including student ID, room number, floor, and allocation date. The

system needs to:

1. Search allocation details using student ID.

2. Sort records based on room number or allocation date.

Student Task

• Use AI to suggest optimized algorithms.

• Justify the selections.

• Implement the solution in Python.

CODE:

Recommended Algorithms and Justifications for Smart Hostel Room Allocation System

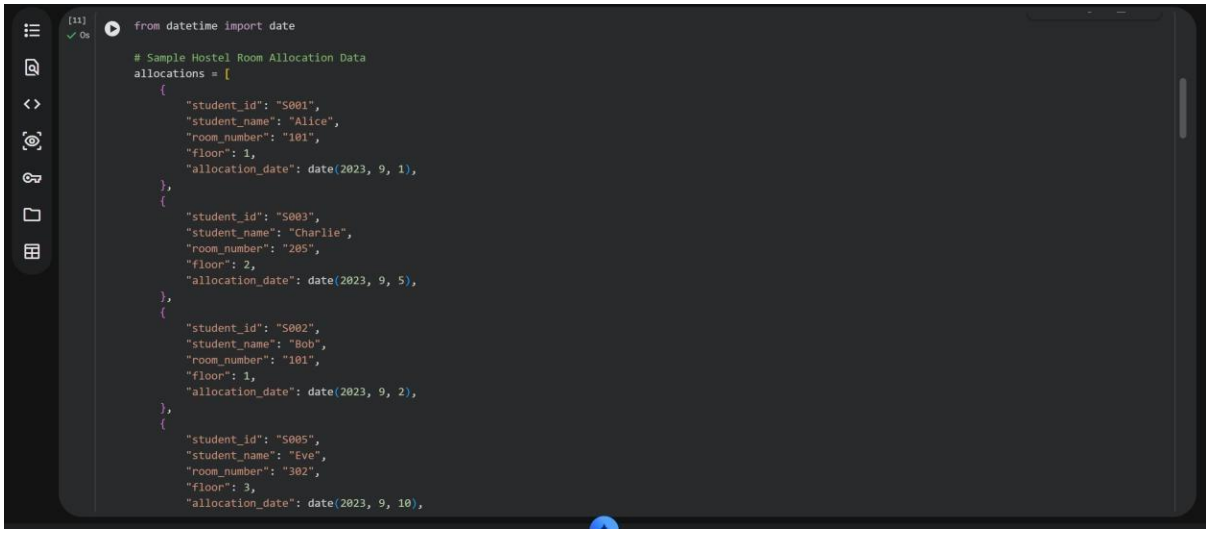
1. Searching Allocation Details by Student ID

- **Algorithm:** Linear Search
- **Justification:** For a list of room allocation records that are typically not kept in sorted order by student_id, a linear search is the most direct and simplest approach. It involves iterating through each allocation record until the student_id matches. While its worst-case time complexity is O(n), it's efficient enough for lists of moderate size

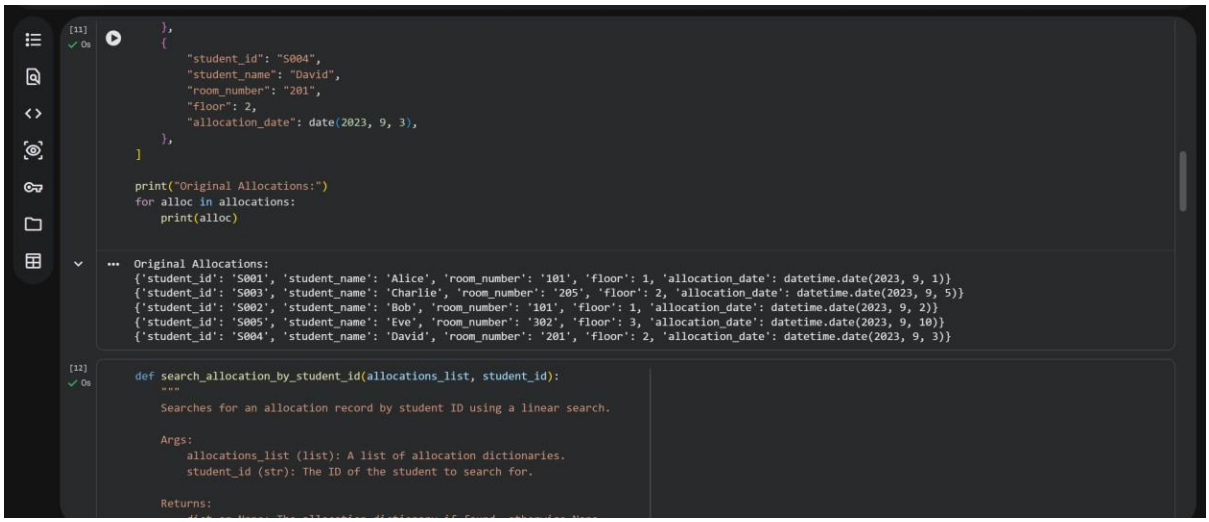
or when the data structure does not support faster indexed lookups. If real-time, very fast lookups on a massive scale were critical, a hash map (Python dictionary) keyed by student_id would be preferred for its average O(1) time complexity.

2. Sorting Records by Room Number or Allocation Date

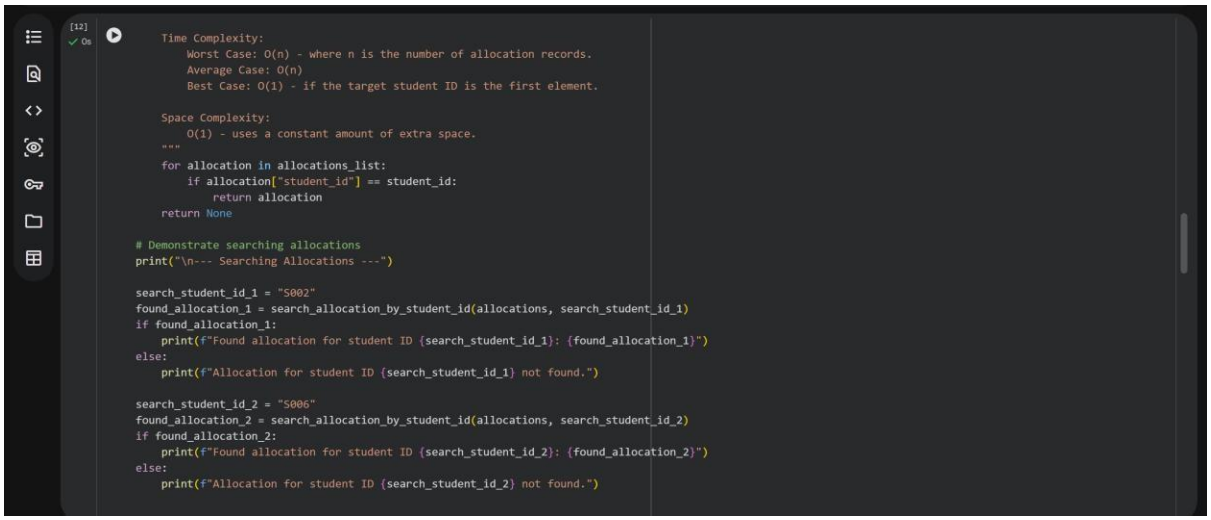
- **Algorithm:** Merge Sort
- **Justification:** Merge Sort is an excellent algorithm for sorting room allocation records based on either room_number or allocation_date for the following reasons:
    - **Time Complexity:** It consistently provides an optimal O(n log n) time complexity across all cases (best, average, and worst). This makes it very efficient for handling larger datasets and ensures predictable performance.
    - **Stability:** Merge Sort is a stable sorting algorithm. This means that if two allocation records have the same room_number or allocation_date, their relative order from the original list will be preserved in the sorted list. This can be important, for instance, if allocations for the same room number need to maintain an original entry order.



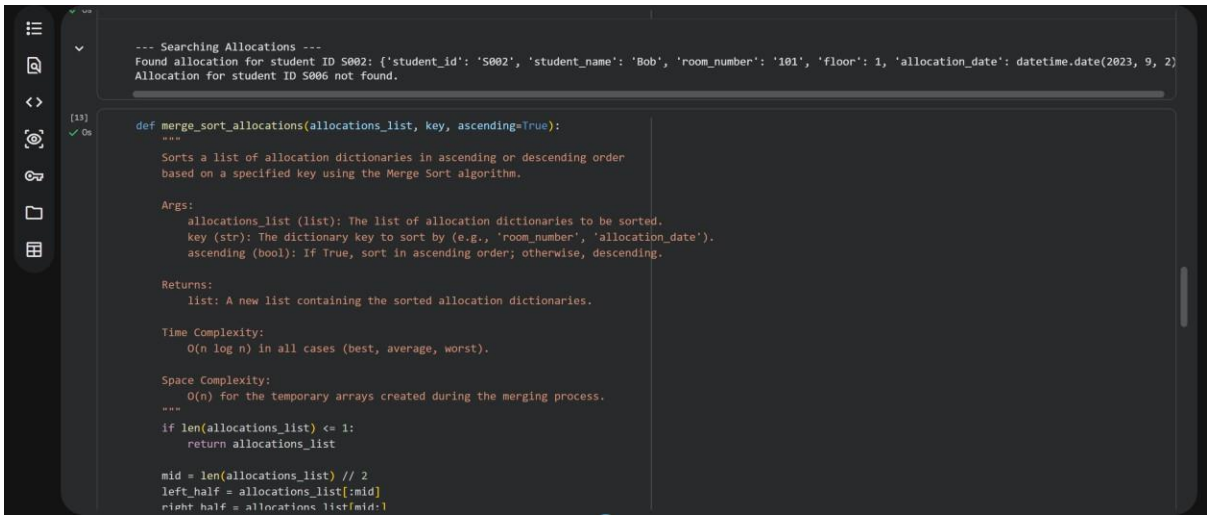
```
[11] ✓ 0s
from datetime import date

# Sample Hostel Room Allocation Data
allocations = [
    {
        "student_id": "S001",
        "student_name": "Alice",
        "room_number": "101",
        "floor": 1,
        "allocation_date": date(2023, 9, 1),
    },
    {
        "student_id": "S003",
        "student_name": "Charlie",
        "room_number": "205",
        "floor": 2,
        "allocation_date": date(2023, 9, 5),
    },
    {
        "student_id": "S002",
        "student_name": "Bob",
        "room_number": "101",
        "floor": 1,
        "allocation_date": date(2023, 9, 2),
    },
    {
        "student_id": "S005",
        "student_name": "Eve",
        "room_number": "302",
        "floor": 3,
        "allocation_date": date(2023, 9, 10),
```



```
    },
    {
        "student_id": "S004",
        "student_name": "David",
        "room_number": "201",
        "floor": 2,
        "allocation_date": date(2023, 9, 3),
    },
]

print("Original Allocations:")
for alloc in allocations:
    print(alloc)

Original Allocations:
{'student_id': 'S001', 'student_name': 'Alice', 'room_number': '101', 'floor': 1, 'allocation_date': datetime.date(2023, 9, 1)}
{'student_id': 'S003', 'student_name': 'Charlie', 'room_number': '205', 'floor': 2, 'allocation_date': datetime.date(2023, 9, 5)}
{'student_id': 'S002', 'student_name': 'Bob', 'room_number': '101', 'floor': 1, 'allocation_date': datetime.date(2023, 9, 2)}
{'student_id': 'S005', 'student_name': 'Eve', 'room_number': '302', 'floor': 3, 'allocation_date': datetime.date(2023, 9, 10)}
{'student_id': 'S004', 'student_name': 'David', 'room_number': '201', 'floor': 2, 'allocation_date': datetime.date(2023, 9, 3)}

[12] ✓ 0s
def search_allocation_by_student_id(allocations_list, student_id):
    """
    Searches for an allocation record by student ID using a linear search.

    Args:
        allocations_list (list): A list of allocation dictionaries.
        student_id (str): The ID of the student to search for.

    Returns:
```



```
[12]
✓ 0s

    Time Complexity:
        Worst Case: O(n) - where n is the number of allocation records.
        Average Case: O(n)
        Best Case: O(1) - if the target student ID is the first element.

    Space Complexity:
        O(1) - uses a constant amount of extra space.
    """
    for allocation in allocations_list:
        if allocation["student_id"] == student_id:
            return allocation
    return None

# Demonstrate searching allocations
print("\n--- Searching Allocations ---")

search_student_id_1 = "S002"
found_allocation_1 = search_allocation_by_student_id(allocations, search_student_id_1)
if found_allocation_1:
    print(f"Found allocation for student ID {search_student_id_1}: {found_allocation_1}")
else:
    print(f"Allocation for student ID {search_student_id_1} not found.")

search_student_id_2 = "S006"
found_allocation_2 = search_allocation_by_student_id(allocations, search_student_id_2)
if found_allocation_2:
    print(f"Found allocation for student ID {search_student_id_2}: {found_allocation_2}")
else:
    print(f"Allocation for student ID {search_student_id_2} not found.")
```



```
--- Searching Allocations ---
Found allocation for student ID S002: {'student_id': 'S002', 'student_name': 'Bob', 'room_number': '101', 'floor': 1, 'allocation_date': datetime.date(2023, 9, 2)
Allocation for student ID S006 not found.
```

```
[13]
✓ 0s

def merge_sort_allocations(allocations_list, key, ascending=True):
    """
    Sorts a list of allocation dictionaries in ascending or descending order
    based on a specified key using the Merge Sort algorithm.

    Args:
        allocations_list (list): The list of allocation dictionaries to be sorted.
        key (str): The dictionary key to sort by (e.g., 'room_number', 'allocation_date').
        ascending (bool): If True, sort in ascending order; otherwise, descending.

    Returns:
        list: A new list containing the sorted allocation dictionaries.

    Time Complexity:
        O(n log n) in all cases (best, average, worst).

    Space Complexity:
        O(n) for the temporary arrays created during the merging process.
    """
    if len(allocations_list) <= 1:
        return allocations_list

    mid = len(allocations_list) // 2
    left_half = allocations_list[:mid]
    right_half = allocations_list[mid:]
```

```
[13]
✓ 0s

    left_half = merge_sort_allocations(left_half, key, ascending)
    right_half = merge_sort_allocations(right_half, key, ascending)

    return merge_allocation_halves(left_half, right_half, key, ascending)

def merge_allocation_halves(left, right, key, ascending):
    merged_list = []
    left_index = 0
    right_index = 0

    while left_index < len(left) and right_index < len(right):
        # Compare elements based on the specified key and sorting order
        if ascending:
            if left[left_index][key] < right[right_index][key]:
                merged_list.append(left[left_index])
                left_index += 1
            else:
                merged_list.append(right[right_index])
                right_index += 1
        else:  # Descending order
            if left[left_index][key] > right[right_index][key]:
                merged_list.append(left[left_index])
                left_index += 1
            else:
                merged_list.append(right[right_index])
                right_index += 1

    # Append any remaining elements
    while left_index < len(left):
        merged_list.append(left[left_index])
        left_index += 1
```

```
[13]
✓ 0s
        while right_index < len(right):
            merged_list.append(right[right_index])
            right_index += 1

        return merged_list

    # Demonstrate sorting by room number (Ascending)
    print("\n--- Allocations Sorted by Room Number (Ascending) ---")
    sorted_by_room = merge_sort_allocations(allocations, "room_number")
    for alloc in sorted_by_room:
        print(alloc)

    # Demonstrate sorting by allocation date (Descending)
    print("\n--- Allocations Sorted by Allocation Date (Descending) ---")
    sorted_by_date_desc = merge_sort_allocations(allocations, "allocation_date", ascending=False)
    for alloc in sorted_by_date_desc:
        print(alloc)
```

OUTPUT:

```
--- Allocations Sorted by Room Number (Ascending) ---
{'student_id': 'S002', 'student_name': 'Bob', 'room_number': '101', 'floor': 1, 'allocation_date': datetime.date(2023, 9, 2)}
{'student_id': 'S001', 'student_name': 'Alice', 'room_number': '101', 'floor': 1, 'allocation_date': datetime.date(2023, 9, 1)}
{'student_id': 'S004', 'student_name': 'David', 'room_number': '201', 'floor': 2, 'allocation_date': datetime.date(2023, 9, 3)}
{'student_id': 'S003', 'student_name': 'Charlie', 'room_number': '205', 'floor': 2, 'allocation_date': datetime.date(2023, 9, 5)}
{'student_id': 'S005', 'student_name': 'Eve', 'room_number': '302', 'floor': 3, 'allocation_date': datetime.date(2023, 9, 10)}

--- Allocations Sorted by Allocation Date (Descending) ---
{'student_id': 'S005', 'student_name': 'Eve', 'room_number': '302', 'floor': 3, 'allocation_date': datetime.date(2023, 9, 10)}
{'student_id': 'S003', 'student_name': 'Charlie', 'room_number': '205', 'floor': 2, 'allocation_date': datetime.date(2023, 9, 5)}
{'student_id': 'S004', 'student_name': 'David', 'room_number': '201', 'floor': 2, 'allocation_date': datetime.date(2023, 9, 3)}
{'student_id': 'S002', 'student_name': 'Bob', 'room_number': '101', 'floor': 1, 'allocation_date': datetime.date(2023, 9, 2)}
{'student_id': 'S001', 'student_name': 'Alice', 'room_number': '101', 'floor': 1, 'allocation_date': datetime.date(2023, 9, 1)}
```

Task Description #6: Online Movie Streaming Platform

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.

2. Sort movies based on rating or release year.

Student Task

• Recommend searching and sorting algorithms using AI.

• Justify the chosen algorithms.

• Implement Python functions.

CODE:

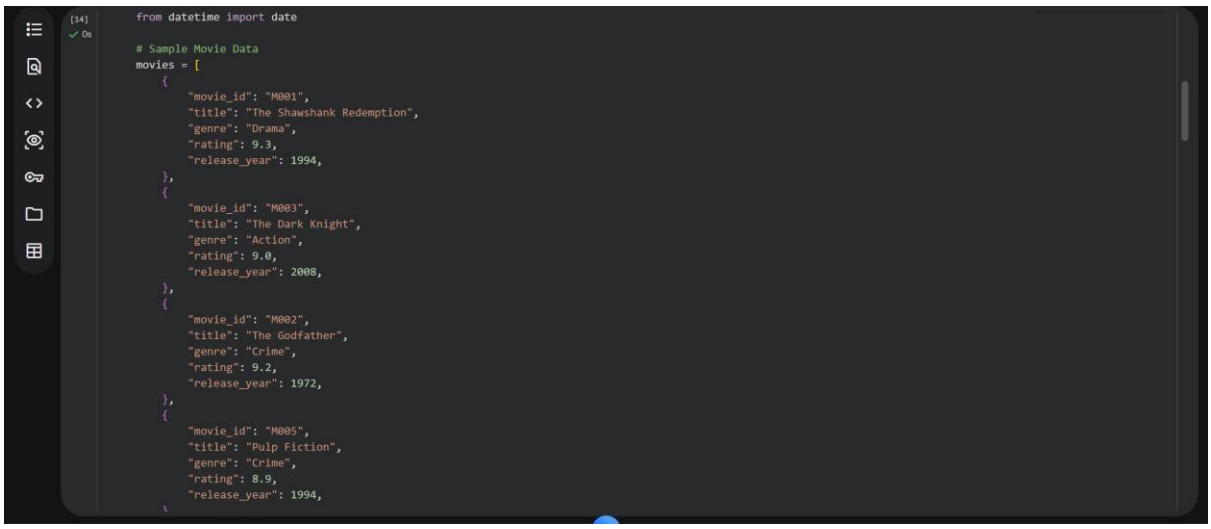
Recommended Algorithms and Justifications for Online Movie Streaming Platform

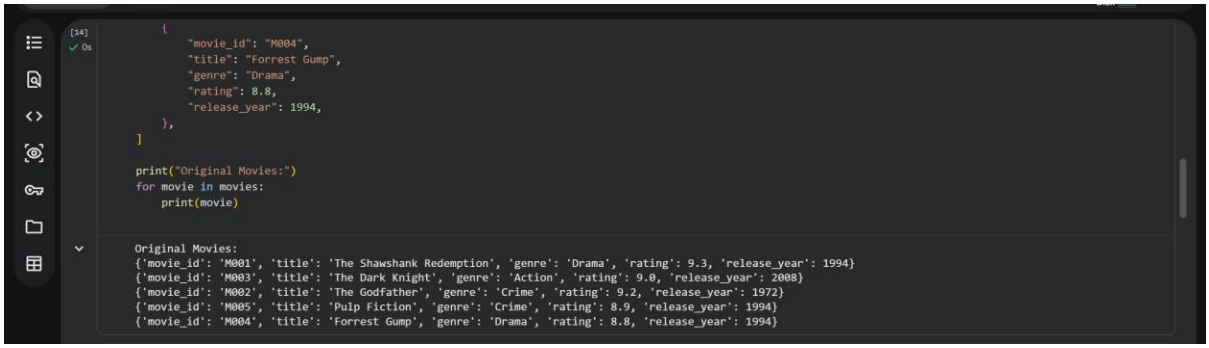
1. Searching Movies by Movie ID

- **Algorithm:** Linear Search
- **Justification:** For a list of movie records where movie_ids are unique but not necessarily stored in any specific order, a linear search is a practical and easy-to-implement solution. It involves iterating through each movie record until the matching movie_id is found. Its time complexity is O(n) in the worst case, which is generally acceptable for moderately sized lists. For platforms with extremely large movie catalogs and frequent ID lookups, a hash map (Python dictionary) could provide average O(1) time complexity by mapping movie IDs to their respective movie objects.

2. Sorting Movies by Rating or Release Year

- **Algorithm:** Merge Sort
- **Justification:** Merge Sort is an excellent choice for sorting movie records based on rating or release_year due to its consistent and efficient performance:
    - **Time Complexity:** It guarantees an optimal O(n log n) time complexity across all cases (best, average, and worst). This makes it highly scalable and reliable for various sizes of movie data, ensuring predictable performance.
    - **Stability:** Merge Sort is a stable sorting algorithm. This property is particularly useful here because if multiple movies have the same rating or release year, their original relative order will be preserved in the sorted output. For example, if two movies have the same high rating, their order might be preserved based on their original entry into the system, which can be a desirable feature.





```
[15]
✓ 0s
def search_movie_by_id(movies_list, movie_id):
    """
    Searches for a movie record by its ID using a linear search.

    Args:
        movies_list (list): A list of movie dictionaries.
        movie_id (str): The ID of the movie to search for.

    Returns:
        dict or None: The movie dictionary if found, otherwise None.

    Time Complexity:
        Worst Case: O(n) - where n is the number of movie records.
        Average Case: O(n)
        Best Case: O(1) - if the target movie is the first element.

    Space Complexity:
        O(1) - uses a constant amount of extra space.
    """
    for movie in movies_list:
        if movie["movie_id"] == movie_id:
            return movie
    return None

# Demonstrate searching movies
print("\n--- Searching Movies ---")

search_movie_id_1 = "M002"
found_movie_1 = search_movie_by_id(movies, search_movie_id_1)
if found_movie_1:
```

```
[15]
✓ 0s
    print(f"Found movie with ID {search_movie_id_1}: {found_movie_1}")
else:
    print(f"Movie with ID {search_movie_id_1} not found.")

search_movie_id_2 = "M006"
found_movie_2 = search_movie_by_id(movies, search_movie_id_2)
if found_movie_2:
    print(f"Found movie with ID {search_movie_id_2}: {found_movie_2}")
else:
    print(f"Movie with ID {search_movie_id_2} not found.")


--- Searching Movies ---
Found movie with ID M002: {'movie_id': 'M002', 'title': 'The Godfather', 'genre': 'Crime', 'rating': 9.2, 'release_year': 1972}
Movie with ID M006 not found.

[16]
✓ 0s
def merge_sort_movies(movies_list, key, ascending=True):
    """
    Sorts a list of movie dictionaries in ascending or descending order
    based on a specified key using the Merge Sort algorithm.

    Args:
        movies_list (list): The list of movie dictionaries to be sorted.
        key (str): The dictionary key to sort by (e.g., 'rating', 'release_year').
        ascending (bool): If True, sort in ascending order; otherwise, descending.

    Returns:
        list: A new list containing the sorted movie dictionaries.

    Time Complexity:
```

```
[16]
✓ 0s
        O(n log n) in all cases (best, average, worst).

    Space Complexity:
        O(n) for the temporary arrays created during the merging process.
    """
    if len(movies_list) <= 1:
        return movies_list

    mid = len(movies_list) // 2
    left_half = movies_list[:mid]
    right_half = movies_list[mid:]

    left_half = merge_sort_movies(left_half, key, ascending)
    right_half = merge_sort_movies(right_half, key, ascending)

    return merge_movie_halves(left_half, right_half, key, ascending)

def merge_movie_halves(left, right, key, ascending):
    merged_list = []
    left_index = 0
    right_index = 0

    while left_index < len(left) and right_index < len(right):
        # Compare elements based on the specified key and sorting order
        if ascending:
            if left[left_index][key] < right[right_index][key]:
                merged_list.append(left[left_index])
                left_index += 1
            else:
                merged_list.append(right[right_index])
                right_index += 1
```

```
                merged_list.append(right[right_index])
                right_index += 1
            else:  # Descending order
                if left[left_index][key] > right[right_index][key]:
                    merged_list.append(left[left_index])
                    left_index += 1
                else:
                    merged_list.append(right[right_index])
                    right_index += 1

        # Append any remaining elements
        while left_index < len(left):
            merged_list.append(left[left_index])
            left_index += 1

        while right_index < len(right):
            merged_list.append(right[right_index])
            right_index += 1

        return merged_list

# Demonstrate sorting by rating (Descending)
print("\n--- Movies Sorted by Rating (Descending) ---")
sorted_by_rating_desc = merge_sort_movies(movies, "rating", ascending=False)
for movie in sorted_by_rating_desc:
    print(movie)

# Demonstrate sorting by release year (Ascending)
print("\n--- Movies Sorted by Release Year (Ascending) ---")
sorted_by_year_asc = merge_sort_movies(movies, "release_year", ascending=True)
for movie in sorted_by_year_asc:
```

OUTPUT:

```
--- Movies Sorted by Rating (Descending) ---
{'movie_id': 'M001', 'title': 'The Shawshank Redemption', 'genre': 'Drama', 'rating': 9.3, 'release_year': 1994}
{'movie_id': 'M002', 'title': 'The Godfather', 'genre': 'Crime', 'rating': 9.2, 'release_year': 1972}
{'movie_id': 'M003', 'title': 'The Dark Knight', 'genre': 'Action', 'rating': 9.0, 'release_year': 2008}
{'movie_id': 'M005', 'title': 'Pulp Fiction', 'genre': 'Crime', 'rating': 8.9, 'release_year': 1994}
{'movie_id': 'M004', 'title': 'Forrest Gump', 'genre': 'Drama', 'rating': 8.8, 'release_year': 1994}

--- Movies Sorted by Release Year (Ascending) ---
{'movie_id': 'M002', 'title': 'The Godfather', 'genre': 'Crime', 'rating': 9.2, 'release_year': 1972}
{'movie_id': 'M004', 'title': 'Forrest Gump', 'genre': 'Drama', 'rating': 8.8, 'release_year': 1994}
{'movie_id': 'M005', 'title': 'Pulp Fiction', 'genre': 'Crime', 'rating': 8.9, 'release_year': 1994}
{'movie_id': 'M001', 'title': 'The Shawshank Redemption', 'genre': 'Drama', 'rating': 9.3, 'release_year': 1994}
{'movie_id': 'M003', 'title': 'The Dark Knight', 'genre': 'Action', 'rating': 9.0, 'release_year': 2008}
```

Task Description #7: Smart Agriculture Crop Monitoring System

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.

2. Sort crops based on moisture level or yield estimate.

Student Task

• Use AI-assisted reasoning to select algorithms.

• Justify algorithm suitability.

• Implement searching and sorting in Python.

CODE:

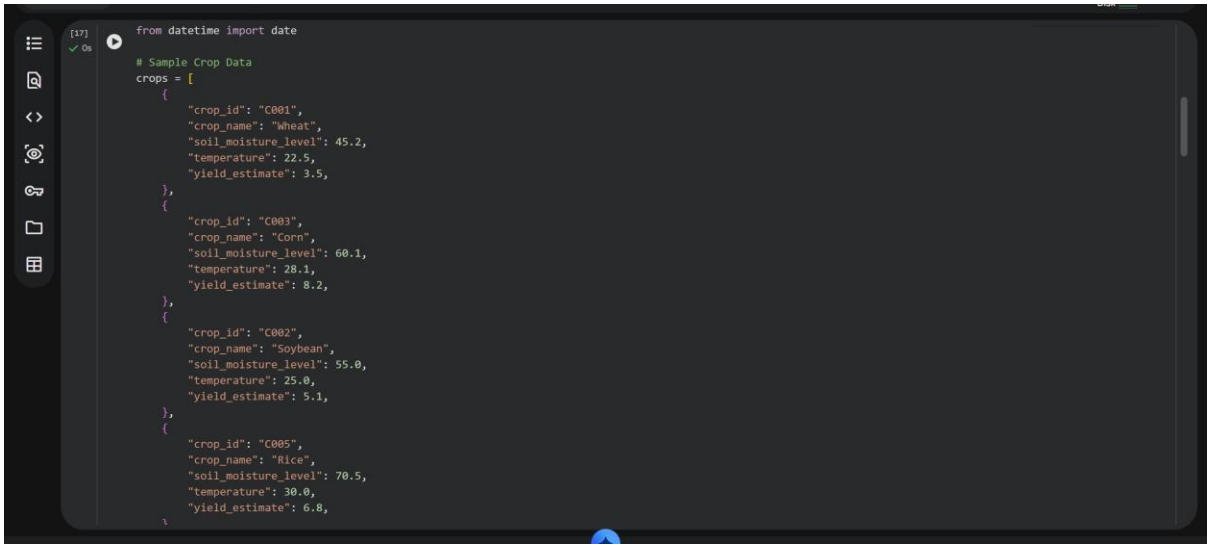
1. Searching Crop Details by Crop ID

- **Algorithm:** Linear Search
- **Justification:** For a list of crop records where crop_ids are unique but not necessarily stored in a sorted order, a linear search is a straightforward and easy-to-implement

approach. It involves iterating through each crop record until the matching crop_id is found. Its time complexity is O(n) in the worst case, which is generally acceptable for typical list sizes in such a system. For very large datasets requiring extremely fast lookups, a hash map (Python dictionary) that maps crop IDs to their respective crop objects would offer average O(1) time complexity.

2. Sorting Crops by Moisture Level or Yield Estimate

- **Algorithm:** Merge Sort
- **Justification:** Merge Sort is an excellent choice for sorting crop records based on soil_moisture_level or yield_estimate due to its consistent and efficient performance:
    - **Time Complexity:** It consistently provides an optimal O(n log n) time complexity across all cases (best, average, and worst). This makes it very efficient for handling larger datasets and ensures predictable performance, which is valuable in monitoring systems.
    - **Stability:** Merge Sort is a stable sorting algorithm. This means that if multiple crops have the same moisture level or yield estimate, their relative order from the original list will be preserved in the sorted output. This can be beneficial for maintaining other inherent orderings within the data (e.g., if you want to see crops with the same moisture level ordered by their planting date).



```
[17]
    {
        "crop_id": "C004",
        "crop_name": "Barley",
        "soil_moisture_level": 40.0,
        "temperature": 20.0,
        "yield_estimate": 2.9,
    },
]

print("Original Crops:")
for crop in crops:
    print(crop)
```

```
Original Crops:
{'crop_id': 'C001', 'crop_name': 'Wheat', 'soil_moisture_level': 45.2, 'temperature': 22.5, 'yield_estimate': 3.5}
{'crop_id': 'C003', 'crop_name': 'Corn', 'soil_moisture_level': 60.1, 'temperature': 28.1, 'yield_estimate': 8.2}
{'crop_id': 'C002', 'crop_name': 'Soybean', 'soil_moisture_level': 55.0, 'temperature': 25.0, 'yield_estimate': 5.1}
{'crop_id': 'C005', 'crop_name': 'Rice', 'soil_moisture_level': 70.5, 'temperature': 30.0, 'yield_estimate': 6.8}
{'crop_id': 'C004', 'crop_name': 'Barley', 'soil_moisture_level': 40.0, 'temperature': 20.0, 'yield_estimate': 2.9}
```

```
[18]
def search_crop_by_id(crops_list, crop_id):
    """
    Searches for a crop record by its ID using a linear search.

    Args:
        crops_list (list): A list of crop dictionaries.
        crop_id (str): The ID of the crop to search for.

    Returns:
        dict or None: The crop dictionary if found, otherwise None.
```

Commands + Code + Text Run all Disk

```
[18]
    Time Complexity:
        Worst Case: O(n) - where n is the number of crop records.
        Average Case: O(n)
        Best Case: O(1) - if the target crop is the first element.

    Space Complexity:
        O(1) - uses a constant amount of extra space.
    """
    for crop in crops_list:
        if crop["crop_id"] == crop_id:
            return crop
    return None

# Demonstrate searching crops
print("\n--- Searching Crops ---")

search_crop_id_1 = "C002"
found_crop_1 = search_crop_by_id(crops, search_crop_id_1)
if found_crop_1:
    print(f"Found crop with ID {search_crop_id_1}: {found_crop_1}")
else:
    print(f"Crop with ID {search_crop_id_1} not found.")

search_crop_id_2 = "C006"
found_crop_2 = search_crop_by_id(crops, search_crop_id_2)
if found_crop_2:
    print(f"Found crop with ID {search_crop_id_2}: {found_crop_2}")
else:
    print(f"Crop with ID {search_crop_id_2} not found.")
```

```
--- Searching Crops ---
Found crop with ID C002: {'crop_id': 'C002', 'crop_name': 'Soybean', 'soil_moisture_level': 55.0, 'temperature': 25.0, 'yield_estimate': 5.1}
Crop with ID C006 not found.
```

```
[19]
def merge_sort_crops(crops_list, key, ascending=True):
    """
    Sorts a list of crop dictionaries in ascending or descending order
    based on a specified key using the Merge Sort algorithm.

    Args:
        crops_list (list): The list of crop dictionaries to be sorted.
        key (str): The dictionary key to sort by (e.g., 'soil_moisture_level', 'yield_estimate').
        ascending (bool): If True, sort in ascending order; otherwise, descending.

    Returns:
        list: A new list containing the sorted crop dictionaries.

    Time Complexity:
        O(n log n) in all cases (best, average, worst).

    Space Complexity:
        O(n) for the temporary arrays created during the merging process.
    """
    if len(crops_list) <= 1:
        return crops_list

    mid = len(crops_list) // 2
    left_half = crops_list[:mid]
    right_half = crops_list[mid:]
```

```
    left_half = merge_sort_crops(left_half, key, ascending)
    right_half = merge_sort_crops(right_half, key, ascending)

    return merge_crop_halves(left_half, right_half, key, ascending)

def merge_crop_halves(left, right, key, ascending):
    merged_list = []
    left_index = 0
    right_index = 0

    while left_index < len(left) and right_index < len(right):
        # Compare elements based on the specified key and sorting order
        if ascending:
            if left[left_index][key] < right[right_index][key]:
                merged_list.append(left[left_index])
                left_index += 1
            else:
                merged_list.append(right[right_index])
                right_index += 1
        else:  # Descending order
            if left[left_index][key] > right[right_index][key]:
                merged_list.append(left[left_index])
                left_index += 1
            else:
                merged_list.append(right[right_index])
                right_index += 1

    # Append any remaining elements
    while left_index < len(left):
        merged_list.append(left[left_index])
        left_index += 1
```

```
    while right_index < len(right):
        merged_list.append(right[right_index])
        right_index += 1

    return merged_list

# Demonstrate sorting by soil moisture level (Ascending)
print("\n--- Crops Sorted by Soil Moisture Level (Ascending) ---")
sorted_by_moisture = merge_sort_crops(crops, "soil_moisture_level")
for crop in sorted_by_moisture:
    print(crop)

# Demonstrate sorting by yield estimate (Descending)
print("\n--- Crops Sorted by Yield Estimate (Descending) ---")
sorted_by_yield_desc = merge_sort_crops(crops, "yield_estimate", ascending=False)
for crop in sorted_by_yield_desc:
    print(crop)
```

OUTPUT:

```
--- Crops Sorted by Soil Moisture Level (Ascending) ---
{'crop_id': 'C004', 'crop_name': 'Barley', 'soil_moisture_level': 40.0, 'temperature': 20.0, 'yield_estimate': 2.9}
{'crop_id': 'C001', 'crop_name': 'Wheat', 'soil_moisture_level': 45.2, 'temperature': 22.5, 'yield_estimate': 3.5}
{'crop_id': 'C002', 'crop_name': 'Soybean', 'soil_moisture_level': 55.0, 'temperature': 25.0, 'yield_estimate': 5.1}
{'crop_id': 'C003', 'crop_name': 'Corn', 'soil_moisture_level': 60.1, 'temperature': 28.1, 'yield_estimate': 8.2}
{'crop_id': 'C005', 'crop_name': 'Rice', 'soil_moisture_level': 70.5, 'temperature': 30.0, 'yield_estimate': 6.8}

--- Crops Sorted by Yield Estimate (Descending) ---
{'crop_id': 'C003', 'crop_name': 'Corn', 'soil_moisture_level': 60.1, 'temperature': 28.1, 'yield_estimate': 8.2}
{'crop_id': 'C005', 'crop_name': 'Rice', 'soil_moisture_level': 70.5, 'temperature': 30.0, 'yield_estimate': 6.8}
{'crop_id': 'C002', 'crop_name': 'Soybean', 'soil_moisture_level': 55.0, 'temperature': 25.0, 'yield_estimate': 5.1}
{'crop_id': 'C001', 'crop_name': 'Wheat', 'soil_moisture_level': 45.2, 'temperature': 22.5, 'yield_estimate': 3.5}
{'crop_id': 'C004', 'crop_name': 'Barley', 'soil_moisture_level': 40.0, 'temperature': 20.0, 'yield_estimate': 2.9}
```

Task Description #8: Airport Flight Management System

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.

2. Sort flights based on departure time or arrival time.

Student Task

• Use AI to recommend algorithms.

• Justify the algorithm selection.

• Implement searching and sorting logic in Python.

CODE:

Recommended Algorithms and Justifications for Airport System

1. Searching Flight Details by Flight ID

- Algorithm: Linear Search
- Justification: For a list of flight records where flight_ids are unique but not necessarily stored in any specific order, a linear search is a practical and easy-to-implement solution. It involves iterating through each flight record until the matching flight_id is found. Its time complexity is O(n) in the worst case, which is generally acceptable for typical list sizes in such a system. For very large datasets requiring extremely fast lookups, a hash map (Python dictionary) that maps flight IDs to their respective flight objects would offer average O(1) time complexity.

2. Sorting Flights by Departure Time or Arrival Time

- Algorithm: Merge Sort
- Justification: Merge Sort is an excellent choice for sorting flight records based on departure_time or arrival_time due to its consistent and efficient performance:
    - Time Complexity: It consistently provides an optimal O(n log n) time complexity across all cases (best, average, and worst). This makes it very efficient for handling larger datasets and ensures predictable performance, which is valuable in real-time systems like airport management.
    - Stability: Merge Sort is a stable sorting algorithm. This means that if multiple flights have the same departure or arrival time, their relative order from the original list will be preserved in the sorted output. This can be beneficial for maintaining other inherent orderings within the data.

```python
from datetime import datetime

# Sample Flight Data
flights = [
    {
        "flight_id": "FA101",
        "airline": "FlyAway",
        "departure_time": datetime(2024, 2, 20, 8, 0),
        "arrival_time": datetime(2024, 2, 20, 10, 30),
        "status": "On Time",
    },
    {
        "flight_id": "SA205",
        "airline": "SkyAir",
        "departure_time": datetime(2024, 2, 20, 12, 15),
        "arrival_time": datetime(2024, 2, 20, 15, 0),
        "status": "Delayed",
    },
    {
        "flight_id": "EA303",
        "airline": "EastWing",
        "departure_time": datetime(2024, 2, 20, 9, 30),
        "arrival_time": datetime(2024, 2, 20, 11, 45),
        "status": "On Time",
    },
    {
        "flight_id": "WA404",
        "airline": "WestJet",
        "departure_time": datetime(2024, 2, 20, 18, 0),
        "arrival_time": datetime(2024, 2, 20, 20, 10),
        "status": "Scheduled",
    },
    {
        "flight_id": "NA502",
        "airline": "NorthStar",
        "departure_time": datetime(2024, 2, 20, 7, 0),
        "arrival_time": datetime(2024, 2, 20, 9, 15),
        "status": "Departed",
    },
]

print("Original Flights:")
for flight in flights:
    print(flight)
```

```
Original Flights:
{'flight_id': 'FA101', 'airline': 'FlyAway', 'departure_time': datetime.datetime(2024, 2, 20, 8, 0), 'arrival_time': datetime.datetime(2024, 2, 20, 10, 30), 'stat
{'flight_id': 'SA205', 'airline': 'SkyAir', 'departure_time': datetime.datetime(2024, 2, 20, 12, 15), 'arrival_time': datetime.datetime(2024, 2, 20, 15, 0), 'stat
{'flight_id': 'EA303', 'airline': 'EastWing', 'departure_time': datetime.datetime(2024, 2, 20, 9, 30), 'arrival_time': datetime.datetime(2024, 2, 20, 11, 45), 'st
{'flight_id': 'WA404', 'airline': 'WestJet', 'departure_time': datetime.datetime(2024, 2, 20, 18, 0), 'arrival_time': datetime.datetime(2024, 2, 20, 20, 10), 'sta
{'flight_id': 'NA502', 'airline': 'NorthStar', 'departure_time': datetime.datetime(2024, 2, 20, 7, 0), 'arrival_time': datetime.datetime(2024, 2, 20, 9, 15), 'sta
```

```python
        dict or None: The flight dictionary if found, otherwise None.

    Time Complexity:
        Worst Case: O(n) - where n is the number of flight records.
        Average Case: O(n)
        Best Case: O(1) - if the target flight is the first element.

    Space Complexity:
        O(1) - uses a constant amount of extra space.
    """
    for flight in flights_list:
        if flight["flight_id"] == flight_id:
            return flight
    return None

# Demonstrate searching flights
print("\n--- Searching Flights ---")

search_flight_id_1 = "EA303"
found_flight_1 = search_flight_by_id(flights, search_flight_id_1)
if found_flight_1:
    print(f"Found flight with ID {search_flight_id_1}: {found_flight_1}")
else:
    print(f"Flight with ID {search_flight_id_1} not found.")

search_flight_id_2 = "ZZ999"
found_flight_2 = search_flight_by_id(flights, search_flight_id_2)
if found_flight_2:
    print(f"Found flight with ID {search_flight_id_2}: {found_flight_2}")
else:
    print(f"Flight with ID {search_flight_id_2} not found.")
```

```
--- Searching Flights ---
Found flight with ID EA303: {'flight_id': 'EA303', 'airline': 'EastWing', 'departure_time': datetime.datetime(2024, 2, 20, 9, 30), 'arrival_time': datetime.dateti
Flight with ID ZZ999 not found.
```

+ Code + Text

```
[22]
✓ 0s
def merge_sort_flights(flights_list, key, ascending=True):
    """
    Sorts a list of flight dictionaries in ascending or descending order
    based on a specified key using the Merge Sort algorithm.

    Args:
        flights_list (list): The list of flight dictionaries to be sorted.
        key (str): The dictionary key to sort by (e.g., 'departure_time', 'arrival_time').
        ascending (bool): If True, sort in ascending order; otherwise, descending.

    Returns:
        list: A new list containing the sorted flight dictionaries.

    Time Complexity:
        O(n log n) in all cases (best, average, worst).

    Space Complexity:
        O(n) for the temporary arrays created during the merging process.
    """
    if len(flights_list) <= 1:
        return flights_list

    mid = len(flights_list) // 2
    left_half = flights_list[:mid]
    right_half = flights_list[mid:]
```

```
[22]
✓ 0s
        right_half = flights_list[mid:]

        left_half = merge_sort_flights(left_half, key, ascending)
        right_half = merge_sort_flights(right_half, key, ascending)

        return merge_flight_halves(left_half, right_half, key, ascending)

    def merge_flight_halves(left, right, key, ascending):
        merged_list = []
        left_index = 0
        right_index = 0

        while left_index < len(left) and right_index < len(right):
            # Compare elements based on the specified key and sorting order
            if ascending:
                if left[left_index][key] < right[right_index][key]:
                    merged_list.append(left[left_index])
                    left_index += 1
                else:
                    merged_list.append(right[right_index])
                    right_index += 1
            else:  # Descending order
                if left[left_index][key] > right[right_index][key]:
                    merged_list.append(left[left_index])
                    left_index += 1
                else:
                    merged_list.append(right[right_index])
                    right_index += 1

        # Append any remaining elements
        while left_index < len(left):
```

```
[22]
✓ 0s
            merged_list.append(left[left_index])
            left_index += 1

        while right_index < len(right):
            merged_list.append(right[right_index])
            right_index += 1

        return merged_list

# Demonstrate sorting by departure time (Ascending)
print("\n--- Flights Sorted by Departure Time (Ascending) ---")
sorted_by_departure = merge_sort_flights(flights, "departure_time")
for flight in sorted_by_departure:
    print(flight)

# Demonstrate sorting by arrival time (Descending)
print("\n--- Flights Sorted by Arrival Time (Descending) ---")
sorted_by_arrival_desc = merge_sort_flights(flights, "arrival_time", ascending=False)
for flight in sorted_by_arrival_desc:
    print(flight)
```

OUTPUT:

```
--- Flights Sorted by Departure Time (Ascending) ---
{'flight_id': 'NA502', 'airline': 'NorthStar', 'departure_time': datetime.datetime(2024, 2, 20, 7, 0), 'arrival_time': datetime.datetime(2024, 2, 20, 9, 15), 'sta
{'flight_id': 'FA101', 'airline': 'FlyAway', 'departure_time': datetime.datetime(2024, 2, 20, 8, 0), 'arrival_time': datetime.datetime(2024, 2, 20, 10, 30), 'stat
{'flight_id': 'EA303', 'airline': 'EastWing', 'departure_time': datetime.datetime(2024, 2, 20, 9, 30), 'arrival_time': datetime.datetime(2024, 2, 20, 11, 45), 'st
{'flight_id': 'SA205', 'airline': 'SkyAir', 'departure_time': datetime.datetime(2024, 2, 20, 12, 15), 'arrival_time': datetime.datetime(2024, 2, 20, 15, 0), 'stat
{'flight_id': 'WA404', 'airline': 'WestJet', 'departure_time': datetime.datetime(2024, 2, 20, 18, 0), 'arrival_time': datetime.datetime(2024, 2, 20, 20, 10), 'sta

--- Flights Sorted by Arrival Time (Descending) ---
{'flight_id': 'WA404', 'airline': 'WestJet', 'departure_time': datetime.datetime(2024, 2, 20, 18, 0), 'arrival_time': datetime.datetime(2024, 2, 20, 20, 10), 'sta
{'flight_id': 'SA205', 'airline': 'SkyAir', 'departure_time': datetime.datetime(2024, 2, 20, 12, 15), 'arrival_time': datetime.datetime(2024, 2, 20, 15, 0), 'stat
{'flight_id': 'EA303', 'airline': 'EastWing', 'departure_time': datetime.datetime(2024, 2, 20, 9, 30), 'arrival_time': datetime.datetime(2024, 2, 20, 11, 45), 'st
{'flight_id': 'FA101', 'airline': 'FlyAway', 'departure_time': datetime.datetime(2024, 2, 20, 8, 0), 'arrival_time': datetime.datetime(2024, 2, 20, 10, 30), 'stat
{'flight_id': 'NA502', 'airline': 'NorthStar', 'departure_time': datetime.datetime(2024, 2, 20, 7, 0), 'arrival_time': datetime.datetime(2024, 2, 20, 9, 15), 'sta
```