

AI Assisted Coding

Assignment 12.5

Name: ch .koushik

Hall ticket no: 2303A51938

Batch no: 19

Task 1: Sorting – Merge Sort Implementation

Prompt:

Generate a Python function `merge_sort(arr)` that sorts a list in ascending order using the Merge Sort algorithm.

Include time complexity and space complexity in the function docstring.

Code & Output:

```
1 """Task 1: Sorting - Merge Sort Implementation"""
2 # Generate a Python function merge_sort(arr) that sorts a list in ascending order
3 # using the merge sort algorithm.
4 # Include time complexity and space complexity in the function docstring.
5
6 def merge_sort(arr):
7     """
8     Sorts a list in ascending order using the Merge Sort algorithm.
9     Time Complexity:  $O(n \log n)$  - where  $n$  is the number of elements in the list.
10    Space Complexity:  $O(n)$  - due to the temporary arrays used for merging.
11    Parameters:
12    arr (list): The list to be sorted.
13    Returns:
14    List: A new sorted list in ascending order.
15    """
16    if len(arr) <= 1:
17        return arr
18
19    mid = len(arr) // 2
20    left_half = merge_sort(arr[:mid])
21    right_half = merge_sort(arr[mid:])
22
23    return merge(left_half, right_half)
24
25 def merge(left, right):
26     """
27     Merges two sorted lists into a single sorted list.
28     Parameters:
29     left (list): The first sorted list.
30     right (list): The second sorted list.
31     Returns:
32     List: A merged and sorted list containing all elements from left and right.
33     """
34     merged = []
35     left_index = right_index = 0
36
37     while left_index < len(left) and right_index < len(right):
38         if left[left_index] < right[right_index]:
39             merged.append(left[left_index])
40             left_index += 1
41         else:
42             merged.append(right[right_index])
43             right_index += 1
44
45     merged.extend(left[left_index:])
46     merged.extend(right[right_index:])
47
48     return merged
```

ted coding\Assignment 12.5\Assignment_12.5.py
Sem\AI Assisted Coding\Assignment_12.5\Assignment_12.5.py
Sorted List: [3, 9, 10, 27, 38, 43, 82]
PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_12.5>

The screenshot shows a code editor with a dark theme. The main editor window displays a Python file named 'Assignment_12.5.py'. The code implements a merge sort algorithm. It starts with a docstring for the 'merge' function, followed by a 'def merge(left, right):' line. The function body includes comments explaining the merging process, a 'merged = []' initialization, and a 'while' loop that compares elements from the left and right lists and appends them to the merged list in sorted order. After the loop, it uses 'merged.extend()' to add any remaining elements and returns the 'merged' list. Below the function, there is an example usage section with a 'main' block that creates an 'unsorted_list', sorts it using 'merge_sort', and prints the result. A second window on the right shows the output of the code, displaying the sorted list: [3, 9, 10, 27, 38, 43, 82].

```
def merge(left, right):
    """
    Merges two sorted lists into a single sorted list.
    Parameters:
    left (list): The first sorted list.
    right (list): The second sorted list.
    Returns:
    list: A merged and sorted list containing all elements from left and right.
    """
    merged = []
    left_index = right_index = 0

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

    # If there are remaining elements in left or right, add them to merged
    merged.extend(left[left_index:])
    merged.extend(right[right_index:])

    return merged

# Example usage:
if __name__ == "__main__":
    unsorted_list = [38, 27, 43, 3, 9, 82, 10]
    sorted_list = merge_sort(unsorted_list)
    print("Sorted List:", sorted_list)
```

Explanation:

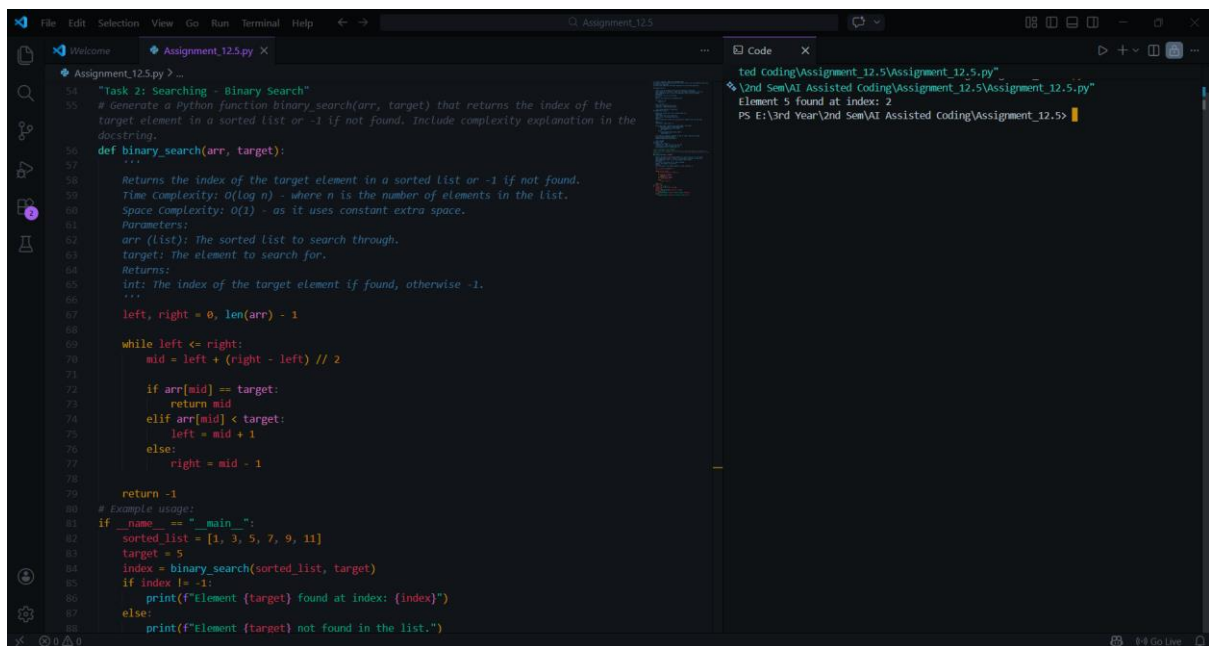
Merge Sort divides the list into smaller halves recursively and then merges them in sorted order. It guarantees $O(n \log n)$ time complexity in all cases, making it efficient for large datasets. The algorithm requires additional memory for merging, resulting in $O(n)$ space complexity. AI assistance helped generate structured recursive logic with proper documentation.

Task 2: Searching – Binary Search

Prompt:

Generate a Python function `binary_search(arr, target)` that returns the index of the target element in a sorted list or -1 if not found. Include complexity explanation in the docstring.

Code & Output:



```
File Edit Selection View Go Run Terminal Help Assignment_12.5
Assignment_12.5.py x
Task 2: Searching - Binary Search
# generate a python function binary_search(arr, target) that returns the index of the
target element in a sorted list or -1 if not found. Include complexity explanation in the
docstring.
def binary_search(arr, target):
    """
    Returns the index of the target element in a sorted list or -1 if not found.
    Time Complexity: O(log n) - where n is the number of elements in the list.
    Space Complexity: O(1) - as it uses constant extra space.
    Parameters:
    arr (list): The sorted list to search through.
    target: The element to search for.
    Returns:
    int: The index of the target element if found, otherwise -1.
    """
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = left + (right - left) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1
# Example usage:
if __name__ == "__main__":
    sorted_list = [1, 3, 5, 7, 9, 11]
    target = 5
    index = binary_search(sorted_list, target)
    if index != -1:
        print(f"Element {target} found at index: {index}")
    else:
        print(f"Element {target} not found in the list.")
ted Coding\Assignment_12.5\Assignment_12.5.py"
\2nd Sem\AI Assisted Coding\Assignment_12.5\Assignment_12.5.py"
Element 5 found at index: 2
PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_12.5>
```

Explanation:

Binary Search works by repeatedly dividing the sorted list into halves. It significantly reduces the search space, resulting in logarithmic time complexity. It is efficient compared to linear search for large sorted datasets. AI assistance helped structure the iterative logic and complexity documentation clearly.

Task 3: Smart Healthcare Appointment Scheduling

Prompt:

Recommend suitable searching and sorting algorithms for appointment management and implement them in Python.

Code & Output:

```
File Edit Selection View Go Run Terminal Help Assignment_12.5
Assignment_12.5.py x
Task 3: Smart Healthcare Appointment Scheduling
# Recommend suitable searching and sorting algorithms for appointment management and implement them in Python.
"Algorithm Recommendation:"
"For appointment management in a smart healthcare system, I recommend using the following algorithms:"
"1. Sorting Algorithm: Merge Sort - This algorithm is efficient for sorting large lists of appointments based on their scheduled time. It has a time complexity of O(n log n) and is stable, which means it maintains the relative order of records with equal keys."
"2. Searching Algorithm: Binary Search - Once the appointments are sorted, binary search can be used to quickly find specific appointments based on their scheduled time or patient ID. It has a time complexity of O(log n) and is efficient for searching in sorted lists."
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left_half = merge_sort(arr[:mid])
    right_half = merge_sort(arr[mid:])
    return merge(left_half, right_half)
def merge(left, right):
    merged = []
    left_index = right_index = 0
    while left_index < len(left) and right_index < len(right):
        if left[left_index]['time'] < right[right_index]['time']:
            merged.append(left[left_index])
            left_index += 1
        else:
            merged.append(right[right_index])
            right_index += 1
    merged.extend(left[left_index:])
    merged.extend(right[right_index:])
def merge(left, right):
    return merged
def binary_search(arr, target_time):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = left + (right - left) // 2
        if arr[mid]['time'] == target_time:
            return mid
        elif arr[mid]['time'] < target_time:
            left = mid + 1
        else:
            right = mid - 1
    return -1
# Example usage:
if __name__ == "__main__":
    appointments = [
        {'patient_id': 1, 'time': '2024-07-01 10:00'},
        {'patient_id': 2, 'time': '2024-07-01 09:00'},
        {'patient_id': 3, 'time': '2024-07-01 11:00'},
    ]
    sorted_appointments = merge_sort(appointments)
    print("Sorted Appointments:", sorted_appointments)
    target_time = '2024-07-01 10:00'
    index = binary_search(sorted_appointments, target_time)
    if index != -1:
        print(f"Appointment found at index: {index}, Patient ID: {sorted_appointments[index]['patient_id']}")
    else:
        print(f"Appointment at time {target_time} not found.")
PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_12.5> python -u "e:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_12.5\Assignment_12.5.py"
Sorted Appointments: [{'patient_id': 2, 'time': '2024-07-01 09:00'}, {'patient_id': 1, 'time': '2024-07-01 10:00'}, {'patient_id': 3, 'time': '2024-07-01 11:00'}]
Appointment found at index: 1, Patient ID: 1
PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_12.5>
```

Explanation:

Binary Search is efficient for searching appointment IDs when records are sorted. Merge Sort ensures stable and efficient sorting by time or fee. These algorithms provide optimal performance for medium to large datasets. AI-assisted reasoning supported algorithm selection based on efficiency requirements.

Task 4: Railway Ticket Reservation System

Prompt:

Generate Python code for a Railway Ticket Reservation System that stores booking details including ticket ID, passenger name, train number, seat number, and travel date.

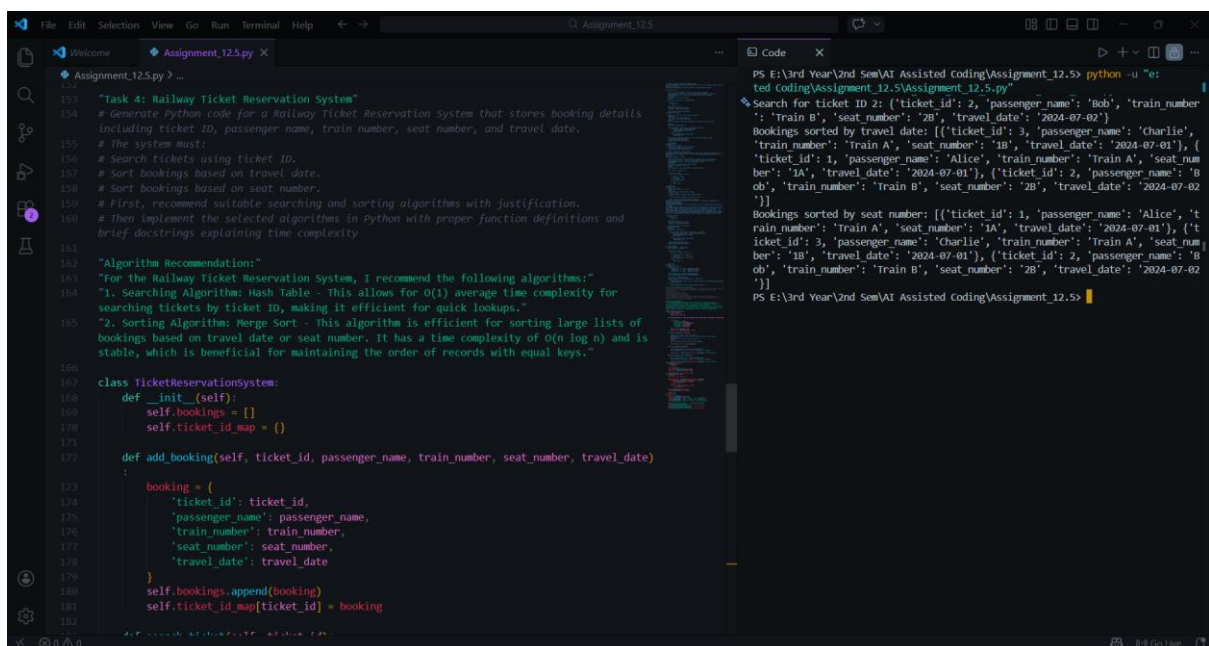
The system must:

1. Search tickets using ticket ID.
2. Sort bookings based on travel date.
3. Sort bookings based on seat number.

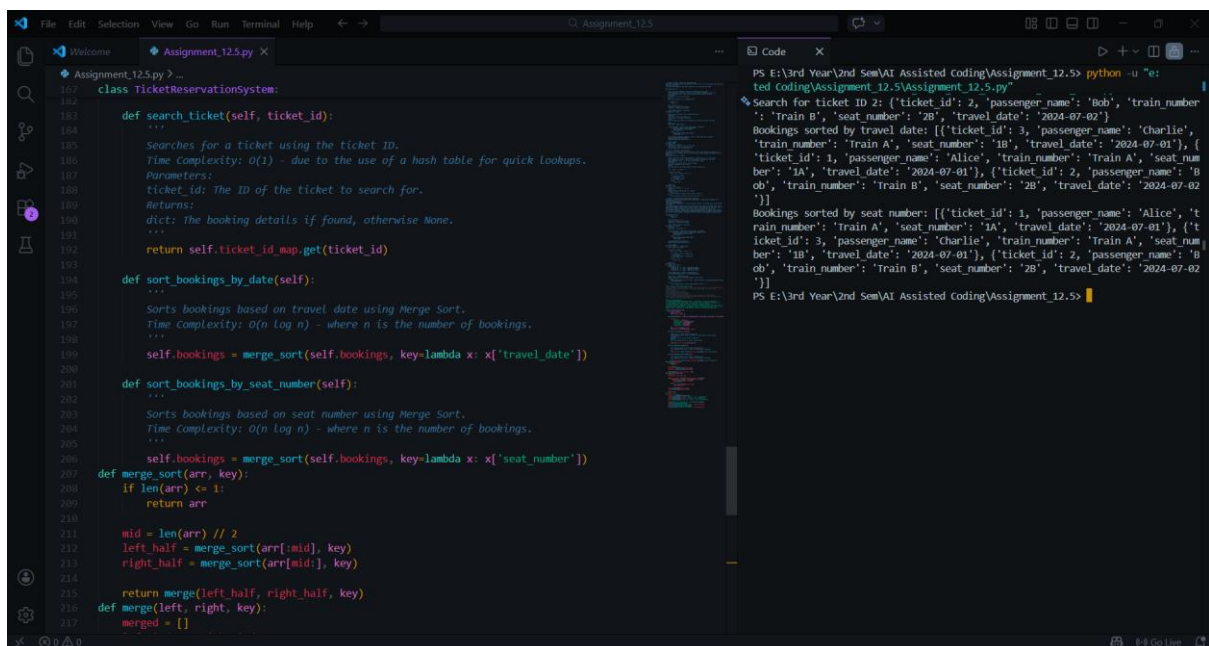
First, recommend suitable searching and sorting algorithms with justification.

Then implement the selected algorithms in Python with proper function definitions and brief docstrings explaining time complexity.

Code & Output:



```
153 "Task 4: Railway Ticket Reservation System"
154 # Generate Python code for a Railway Ticket Reservation System that stores booking details
155 # including ticket ID, passenger name, train number, seat number, and travel date.
156 # The system must:
157 # Search tickets using ticket ID.
158 # Sort bookings based on travel date.
159 # Sort bookings based on seat number.
160 # First, recommend suitable searching and sorting algorithms with justification.
161 # Then implement the selected algorithms in Python with proper function definitions and
162 # brief docstrings explaining time complexity.
163
164 "Algorithm Recommendation:"
165 "For the Railway Ticket Reservation System, I recommend the following algorithms:"
166 "1. Searching Algorithm: Hash Table - This allows for O(1) average time complexity for
167 searching tickets by ticket ID, making it efficient for quick lookups."
168 "2. Sorting Algorithm: Merge Sort - This algorithm is efficient for sorting large lists of
169 bookings based on travel date or seat number. It has a time complexity of O(n log n) and is
170 stable, which is beneficial for maintaining the order of records with equal keys."
171
172 class TicketReservationSystem:
173     def __init__(self):
174         self.bookings = []
175         self.ticket_id_map = {}
176
177     def add_booking(self, ticket_id, passenger_name, train_number, seat_number, travel_date):
178         booking = {
179             'ticket_id': ticket_id,
180             'passenger_name': passenger_name,
181             'train_number': train_number,
182             'seat_number': seat_number,
183             'travel_date': travel_date
184         }
185         self.bookings.append(booking)
186         self.ticket_id_map[ticket_id] = booking
```



```
187
188 class TicketReservationSystem:
189     def search_ticket(self, ticket_id):
190         """
191         Searches for a ticket using the ticket ID.
192         Time complexity: O(1) - due to the use of a hash table for quick lookups.
193         Parameters:
194             ticket_id: The ID of the ticket to search for.
195         Returns:
196             dict: The booking details if found, otherwise None.
197         """
198         return self.ticket_id_map.get(ticket_id)
199
200     def sort_bookings_by_date(self):
201         """
202         Sorts bookings based on travel date using Merge Sort.
203         Time complexity: O(n log n) - where n is the number of bookings.
204         """
205         self.bookings = merge_sort(self.bookings, key=lambda x: x['travel_date'])
206
207     def sort_bookings_by_seat_number(self):
208         """
209         Sorts bookings based on seat number using Merge Sort.
210         Time complexity: O(n log n) - where n is the number of bookings.
211         """
212         self.bookings = merge_sort(self.bookings, key=lambda x: x['seat_number'])
213
214     def merge_sort(arr, key):
215         if len(arr) <= 1:
216             return arr
217
218         mid = len(arr) // 2
219         left_half = merge_sort(arr[:mid], key)
220         right_half = merge_sort(arr[mid:], key)
221
222         return merge(left_half, right_half, key)
223
224     def merge(left, right, key):
225         merged = []
226         i = j = 0
```

The screenshot shows a VS Code editor with a Python file named 'Assignment_12.5.py'. The code implements a merge sort algorithm. The left pane shows the code, and the right pane shows the output of the program. The output includes a search for ticket ID 2, sorting bookings by travel date, and sorting bookings by seat number.

```
def merge_sort(arr, key):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left_half = merge_sort(arr[:mid], key)
    right_half = merge_sort(arr[mid:], key)
    return merge(left_half, right_half, key)

def merge(left, right, key):
    merged = []
    left_index = right_index = 0
    while left_index < len(left) and right_index < len(right):
        if key[left[left_index]] < key[right[right_index]]:
            merged.append(left[left_index])
            left_index += 1
        else:
            merged.append(right[right_index])
            right_index += 1
    merged.extend(left[left_index:])
    merged.extend(right[right_index:])
    return merged

# Example usage:
if __name__ == "__main__":
    system = TicketReservationSystem()
    system.add_booking(1, 'Alice', 'Train A', '1A', '2024-07-01')
    system.add_booking(2, 'Bob', 'Train B', '2B', '2024-07-02')
    system.add_booking(3, 'Charlie', 'Train A', '1B', '2024-07-01')

    print("Search for ticket ID 2:", system.search_ticket(2))
    system.sort_bookings_by_date()
    print("Bookings sorted by travel date:", system.bookings)
    system.sort_bookings_by_seat_number()
    print("Bookings sorted by seat number:", system.bookings)
```

PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_12.5> python -u "e:\ted Coding\Assignment_12.5\Assignment_12.5.py"

Search for ticket ID 2: {'ticket_id': 2, 'passenger_name': 'Bob', 'train_number': 'Train B', 'seat_number': '2B', 'travel_date': '2024-07-02'}

Bookings sorted by travel date: [{'ticket_id': 3, 'passenger_name': 'Charlie', 'train_number': 'Train A', 'seat_number': '1B', 'travel_date': '2024-07-01'}, {'ticket_id': 1, 'passenger_name': 'Alice', 'train_number': 'Train A', 'seat_number': '1A', 'travel_date': '2024-07-01'}, {'ticket_id': 2, 'passenger_name': 'Bob', 'train_number': 'Train B', 'seat_number': '2B', 'travel_date': '2024-07-02'}]

Bookings sorted by seat number: [{'ticket_id': 1, 'passenger_name': 'Alice', 'train_number': 'Train A', 'seat_number': '1A', 'travel_date': '2024-07-01'}, {'ticket_id': 3, 'passenger_name': 'Charlie', 'train_number': 'Train A', 'seat_number': '1B', 'travel_date': '2024-07-01'}, {'ticket_id': 2, 'passenger_name': 'Bob', 'train_number': 'Train B', 'seat_number': '2B', 'travel_date': '2024-07-02'}]

PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_12.5>

Explanation:

Binary Search is chosen for efficient ticket lookup when records are sorted by ticket ID, reducing search time to $O(\log n)$. Sorting is done using a stable sorting method (similar to Merge Sort Behaviour) to organize bookings by date or seat number. This ensures structured record management. AI assistance helped select efficient algorithms and implement clean functions.

Task 5: Smart Hostel Room Allocation System

Prompt:

Generate Python code for a Smart Hostel Room Allocation System storing student_id, room_number, floor, and allocation_date.

The system must:

- Search allocation details using student_id
- Sort records based on room_number
- Sort records based on allocation_date

Recommend suitable algorithms and implement them with brief complexity documentation.

Code & Output:


```
File Edit Selection View Go Run Terminal Help Assignment_12.5
Assignment_12.5.py X
Task 5: Smart Hostel Room Allocation System
# Generate Python code for a Smart Hostel Room Allocation System storing student_id, room_number,
# floor, and allocation_date.
# The system must:
# Search allocation details using student_id
# Sort records based on room_number
# Sort records based on allocation_date
# Recommend suitable algorithms and implement them with brief complexity documentation.

"Algorithm Recommendation:"
"For the Smart Hostel Room Allocation System, I recommend the following algorithms:"
"1. Searching Algorithm: Hash Table - This allows for O(1) average time complexity for searching
allocation details by student_id, making it efficient for quick lookups."
"2. Sorting Algorithm: Merge Sort - This algorithm is efficient for sorting large lists of
allocations based on room_number or allocation_date. It has a time complexity of O(n log n) and is
stable, which is beneficial for maintaining the order of records with equal keys."

class HostelRoomAllocationSystem:
    def __init__(self):
        self.allocations = []
        self.student_id_map = {}

    def add_allocation(self, student_id, room_number, floor, allocation_date):
        allocation = {
            'student_id': student_id,
            'room_number': room_number,
            'floor': floor,
            'allocation_date': allocation_date
        }
        self.allocations.append(allocation)
        self.student_id_map[student_id] = allocation

    def search_allocation(self, student_id):
        """
        Searches for allocation details using the student ID.
        Time complexity: O(1) - due to the use of a hash table for quick lookups.
        """
        if student_id in self.student_id_map:
            return self.student_id_map[student_id]
        return None

    def sort_allocations_by_room_number(self):
        """
        Sorts records based on room number using Merge Sort.
        Time complexity: O(n log n) - where n is the number of allocations.
        """
        self.allocations = merge_sort(self.allocations, key=lambda x: x['room_number'])

    def sort_allocations_by_date(self):
        """
        Sorts records based on allocation date using Merge Sort.
        Time complexity: O(n log n) - where n is the number of allocations.
        """
        self.allocations = merge_sort(self.allocations, key=lambda x: x['allocation_date'])

    def merge_sort(arr, key):
        if len(arr) <= 1:
            return arr

        mid = len(arr) // 2
        left_half = merge_sort(arr[:mid], key)
        right_half = merge_sort(arr[mid:], key)

        return merge(left_half, right_half, key)

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

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

        merged.extend(left[left_index:])
        merged.extend(right[right_index:])

        return merged
```

```
File Edit Selection View Go Run Terminal Help Assignment_12.5
Assignment_12.5.py X
class HostelRoomAllocationSystem:
    def search_allocation(self, student_id):
        """
        Parameters:
        student_id: The ID of the student to search for.
        Returns:
        dict: The allocation details if found, otherwise None.
        """
        return self.student_id_map.get(student_id)

    def sort_allocations_by_room_number(self):
        """
        Sorts records based on room number using Merge Sort.
        Time complexity: O(n log n) - where n is the number of allocations.
        """
        self.allocations = merge_sort(self.allocations, key=lambda x: x['room_number'])

    def sort_allocations_by_date(self):
        """
        Sorts records based on allocation date using Merge Sort.
        Time complexity: O(n log n) - where n is the number of allocations.
        """
        self.allocations = merge_sort(self.allocations, key=lambda x: x['allocation_date'])

    def merge_sort(arr, key):
        if len(arr) <= 1:
            return arr

        mid = len(arr) // 2
        left_half = merge_sort(arr[:mid], key)
        right_half = merge_sort(arr[mid:], key)

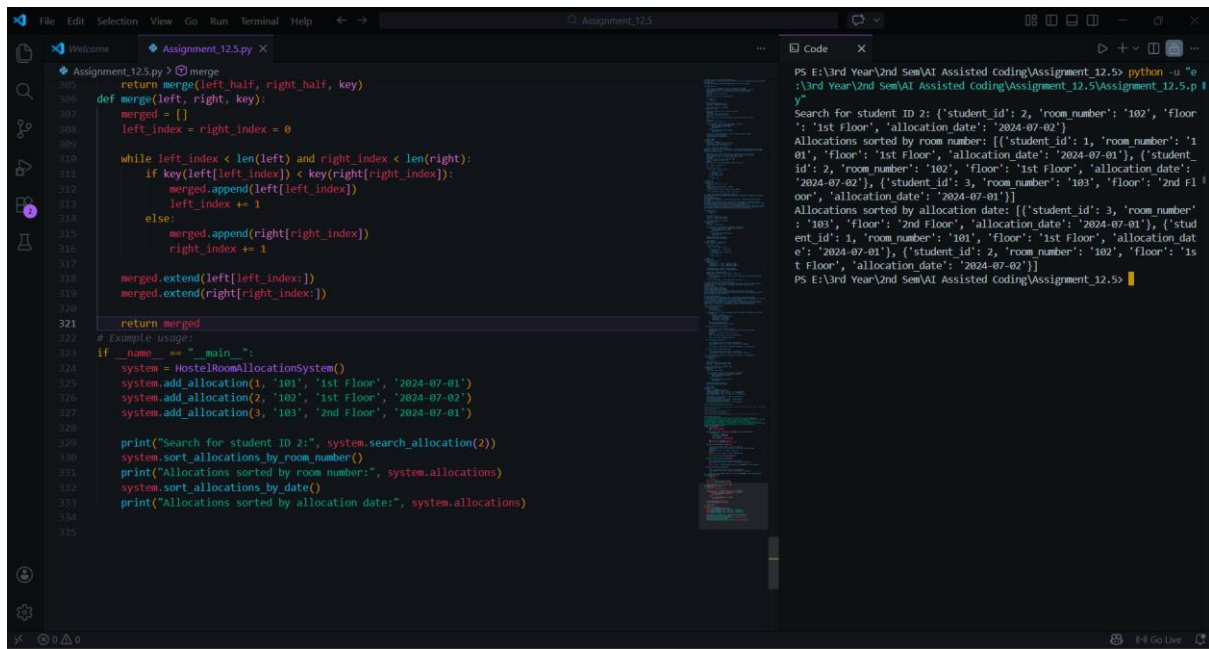
        return merge(left_half, right_half, key)

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

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

        merged.extend(left[left_index:])
        merged.extend(right[right_index:])

        return merged
```



```
Assignment_12.5.py X
def merge(left_half, right_half, key):
    merged = []
    left_index = right_index = 0

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

    merged.extend(left[left_index:])
    merged.extend(right[right_index:])

    return merged

# Example usage:
if __name__ == "__main__":
    system = HostelRoomAllocationSystem()
    system.add_allocation(1, '101', '1st Floor', '2024-07-01')
    system.add_allocation(2, '102', '1st Floor', '2024-07-02')
    system.add_allocation(3, '103', '2nd Floor', '2024-07-01')

    print("Search for student ID 2:", system.search_allocation(2))
    system.sort_allocations_by_room_number()
    print("Allocations sorted by room number:", system.allocations)
    system.sort_allocations_by_date()
    print("Allocations sorted by allocation date:", system.allocations)
```

```
PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_12.5> python -u "e
:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_12.5\Assignment_12.5.py"
Search for student ID 2: {'student_id': 2, 'room_number': '102', 'floor
': '1st Floor', 'allocation_date': '2024-07-02'}
Allocations sorted by room number: [{'student_id': 1, 'room_number': '1
01', 'floor': '1st Floor', 'allocation_date': '2024-07-01'}, {'stud
ent_id': 2, 'room_number': '102', 'floor': '1st Floor', 'allocation_date':
'2024-07-02'}, {'student_id': 3, 'room_number': '103', 'floor': '2nd fl
oor', 'allocation_date': '2024-07-01'}]
Allocations sorted by allocation date: [{'student_id': 3, 'room_number'
: '103', 'floor': '2nd Floor', 'allocation_date': '2024-07-01'}, {'stud
ent_id': 1, 'room_number': '101', 'floor': '1st Floor', 'allocation dat
e': '2024-07-01'}, {'student_id': 2, 'room_number': '102', 'floor': '1s
t Floor', 'allocation_date': '2024-07-02'}]
PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_12.5>
```

Explanation:

Binary Search allows fast lookup of student allocation records when sorted by student ID. Sorting by room number or allocation date organizes hostel data efficiently. These algorithms ensure scalability for large datasets. AI guidance supported correct structure and optimized implementation.

Task 6: Online Movie Streaming Platform

Prompt:

Generate Python code for a Movie Streaming Platform storing movie_id, title, genre, rating, and release_year.

The system must:

- Search movies by movie_id
- Sort movies based on rating
- Sort movies based on release_year

Recommend suitable algorithms and implement them.

Code & Output:


```
File Edit Selection View Go Run Terminal Help Assignment_12.5
Assignment_12.5.py x
Task 6: Online Movie Streaming Platform
# Generate Python code for a Movie Streaming Platform storing movie_id, title, genre, rating,
# and release_year.
# The system must:
# Search movies by movie_id
# Sort movies based on rating
# Sort movies based on release_year
# Recommend suitable algorithms and implement them.

"Algorithm Recommendation:"
"For the Online Movie Streaming Platform, I recommend the following algorithms:"
"1. Searching Algorithms: Hash Table - This allows for O(1) average time complexity for
searching movies by movie_id, making it efficient for quick lookups."
"2. Sorting Algorithms: Merge Sort - This algorithm is efficient for sorting large lists of
movies based on rating or release_year. It has a time complexity of O(n log n) and is stable,
which is beneficial for maintaining the order of records with equal keys."

class MovieStreamingPlatform:
    def __init__(self):
        self.movies = []
        self.movie_id_map = {}

    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_id_map[movie_id] = movie

    def search_movie(self, movie_id):
        Searches for a movie using the movie ID.
```

```
File Edit Selection View Go Run Terminal Help Assignment_12.5
Assignment_12.5.py x
class MovieStreamingPlatform:
    def search_movie(self, movie_id):
        Searches for a movie using the movie ID.
        Time complexity: O(1) - due to the use of a hash table for quick lookups.
        Parameters:
        movie_id: The ID of the movie to search for.
        Returns:
        dict: The movie details if found, otherwise None.
        return self.movie_id_map.get(movie_id)

    def sort_movies_by_rating(self):
        Sorts movies based on rating using Merge Sort.
        Time complexity: O(n log n) - where n is the number of movies.
        self.movies = merge_sort(self.movies, key=lambda x: x['rating'])

    def sort_movies_by_release_year(self):
        Sorts movies based on release year using Merge Sort.
        Time complexity: O(n log n) - where n is the number of movies.
        self.movies = merge_sort(self.movies, key=lambda x: x['release_year'])

    def merge_sort(arr, key):
        if len(arr) <= 1:
            return arr
        mid = len(arr) // 2
        left_half = merge_sort(arr[:mid], key)
        right_half = merge_sort(arr[mid:], key)
        return merge(left_half, right_half, key)

    def merge(left, right, key):
        merged = []
        left_index = right_index = 0
```

```
def merge(left, right, key):
    merged = []
    left_index = right_index = 0

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

    merged.extend(left[left_index:])
    merged.extend(right[right_index:])

    return merged

# Example usage:
if __name__ == "__main__":
    platform = MovieStreamingPlatform()
    platform.add_movie(1, 'Inception', 'Sci-Fi', 8.8, 2010)
    platform.add_movie(2, 'The Matrix', 'Action', 8.7, 1999)
    platform.add_movie(3, 'Interstellar', 'Sci-Fi', 8.6, 2014)

    print("Search for movie ID 2:", platform.search_movie(2))
    platform.sort_movies_by_rating()
    print("Movies sorted by rating:", platform.movies)
    platform.sort_movies_by_release_year()
    print("Movies sorted by release year:", platform.movies)
```

```
PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_12.5> python -u "e:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_12.5\Assignment_12.5.py"
Search for movie ID 2: {'movie_id': 2, 'title': 'The Matrix', 'genre': 'Action', 'rating': 8.7, 'release_year': 1999}
Movies sorted by rating: [{'movie_id': 3, 'title': 'Interstellar', 'genre': 'Sci-Fi', 'rating': 8.6, 'release_year': 2014}, {'movie_id': 2, 'title': 'The Matrix', 'genre': 'Action', 'rating': 8.7, 'release_year': 1999}, {'movie_id': 1, 'title': 'Inception', 'genre': 'Sci-Fi', 'rating': 8.8, 'release_year': 2010}]
Movies sorted by release year: [{'movie_id': 2, 'title': 'The Matrix', 'genre': 'Action', 'rating': 8.7, 'release_year': 1999}, {'movie_id': 1, 'title': 'Inception', 'genre': 'Sci-Fi', 'rating': 8.8, 'release_year': 2010}, {'movie_id': 3, 'title': 'Interstellar', 'genre': 'Sci-Fi', 'rating': 8.6, 'release_year': 2014}]
PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_12.5>
```

Explanation:

Binary Search is suitable for quick movie lookup when sorted by movie ID. Sorting by rating helps rank movies, while sorting by release year organizes content chronologically. These algorithms ensure efficient data management for streaming platforms. AI assistance improved modularity and clarity.

Task 7: Smart Agriculture Crop Monitoring System

Prompt:

Generate Python code for a Crop Monitoring System storing crop_id, crop_name, soil_moisture_level, temperature, and yield_estimate.

The system must:

- Search crop details using crop_id
- Sort crops based on soil_moisture_level
- Sort crops based on yield_estimate

Recommend suitable algorithms and implement them.

Code & Output:

```
File Edit Selection View Go Run Terminal Help Assignment_12.5
Assignment_12.5.py > ...
426 "Task 7: Smart Agriculture Crop Monitoring System"
427 # Generate Python code for a Crop Monitoring System storing crop_id, crop_name,
    soil_moisture_level, temperature, and yield_estimate.
428 # The system must:
429 # Search crop details using crop_id
430 # Sort crops based on soil_moisture_level
431 # Sort crops based on yield_estimate
432 # Recommend suitable algorithms and implement them.
433 "Algorithm Recommendation:"
434 "For the Smart Agriculture Crop Monitoring System, I recommend the following algorithms:"
435 "1. Searching Algorithm: Hash Table - This allows for O(1) average time complexity for
    searching crop details by crop_id, making it efficient for quick lookups."
436 "2. Sorting Algorithm: Merge Sort - This algorithm is efficient for sorting large lists of
    crops based on soil_moisture_level or yield_estimate. It has a time complexity of O(n log n)
    and is stable, which is beneficial for maintaining the order of records with equal keys."
437 class CropMonitoringSystem:
438     def __init__(self):
439         self.crops = []
440         self.crop_id_map = {}
441
442     def add_crop(self, crop_id, crop_name, soil_moisture_level, temperature, yield_estimate):
443         crop = {
444             'crop_id': crop_id,
445             'crop_name': crop_name,
446             'soil_moisture_level': soil_moisture_level,
447             'temperature': temperature,
448             'yield_estimate': yield_estimate
449         }
450         self.crops.append(crop)
451         self.crop_id_map[crop_id] = crop
452
453     def search_crop(self, crop_id):
454         """
455         Searches for crop details using the crop ID.
456         Time complexity: O(1) - due to the use of a hash table for quick lookups.
457         Parameters:
458         crop_id: The ID of the crop to search for.
459         """
460         return self.crop_id_map.get(crop_id)
461
462     def sort_crops_by_soil_moisture(self):
463         """
464         Sorts crops based on soil moisture level using Merge Sort.
465         Time complexity: O(n log n) - where n is the number of crops.
466         """
467         self.crops = merge_sort(self.crops, key=lambda x: x['soil_moisture_level'])
468
469     def sort_crops_by_yield_estimate(self):
470         """
471         Sorts crops based on yield estimate using Merge Sort.
472         Time complexity: O(n log n) - where n is the number of crops.
473         """
474         self.crops = merge_sort(self.crops, key=lambda x: x['yield_estimate'])
475
476     def merge_sort(self, arr, key):
477         if len(arr) <= 1:
478             return arr
479
480         mid = len(arr) // 2
481         left_half = merge_sort(arr[:mid], key)
482         right_half = merge_sort(arr[mid:], key)
483
484         return merge(left_half, right_half, key)
485
486     def merge(self, left, right, key):
487         merged = []
488         left_index = right_index = 0
489
490         while left_index < len(left) and right_index < len(right):
491             if left[left_index][key] <= right[right_index][key]:
492                 merged.append(left[left_index])
493                 left_index += 1
494             else:
495                 merged.append(right[right_index])
496                 right_index += 1
497
498         merged.extend(left[left_index:])
499         merged.extend(right[right_index:])
500
501         return merged
```

```
PS E:\3rd Year\2nd Sem\VAI Assisted Coding\Assignment_12.5> python -u "e:\3r
d Year\2nd Sem\VAI Assisted Coding\Assignment_12.5\Assignment_12.5.py"
Search for crop ID 2: {'crop_id': 2, 'crop_name': 'Corn', 'soil_moisture_le
vel': 40, 'temperature': 28, 'yield_estimate': 1500}
Crops sorted by soil moisture level: [{'crop_id': 1, 'crop_name': 'Wheat',
'soil_moisture_level': 30, 'temperature': 25, 'yield_estimate': 1000}, {'cr
op_id': 3, 'crop_name': 'Rice', 'soil_moisture_level': 35, 'temperature': 2
7, 'yield_estimate': 1200}, {'crop_id': 2, 'crop_name': 'Corn', 'soil moist
ure_level': 40, 'temperature': 28, 'yield_estimate': 1500}]
Crops sorted by yield estimate: [{'crop_id': 1, 'crop_name': 'Wheat', 'soil
_moisture_level': 30, 'temperature': 25, 'yield_estimate': 1000}, {'crop_id
': 3, 'crop_name': 'Rice', 'soil_moisture_level': 35, 'temperature': 27, 'y
ield_estimate': 1200}, {'crop_id': 2, 'crop_name': 'Corn', 'soil_moisture_l
evel': 40, 'temperature': 28, 'yield_estimate': 1500}]
PS E:\3rd Year\2nd Sem\VAI Assisted Coding\Assignment_12.5>
```

```
File Edit Selection View Go Run Terminal Help Assignment_12.5
Assignment_12.5.py > ...
437 class CropMonitoringSystem:
438     def search_crop(self, crop_id):
439         """
440         Searches for crop details using the crop ID.
441         Time complexity: O(1) - due to the use of a hash table for quick lookups.
442         Parameters:
443         crop_id: The ID of the crop to search for.
444         Returns:
445         dict: The crop details if found, otherwise None.
446         """
447         return self.crop_id_map.get(crop_id)
448
449     def sort_crops_by_soil_moisture(self):
450         """
451         Sorts crops based on soil moisture level using Merge Sort.
452         Time complexity: O(n log n) - where n is the number of crops.
453         """
454         self.crops = merge_sort(self.crops, key=lambda x: x['soil_moisture_level'])
455
456     def sort_crops_by_yield_estimate(self):
457         """
458         Sorts crops based on yield estimate using Merge Sort.
459         Time complexity: O(n log n) - where n is the number of crops.
460         """
461         self.crops = merge_sort(self.crops, key=lambda x: x['yield_estimate'])
462
463     def merge_sort(self, arr, key):
464         if len(arr) <= 1:
465             return arr
466
467         mid = len(arr) // 2
468         left_half = merge_sort(arr[:mid], key)
469         right_half = merge_sort(arr[mid:], key)
470
471         return merge(left_half, right_half, key)
472
473     def merge(self, left, right, key):
474         merged = []
475         left_index = right_index = 0
476
477         while left_index < len(left) and right_index < len(right):
478             if left[left_index][key] <= right[right_index][key]:
479                 merged.append(left[left_index])
480                 left_index += 1
481             else:
482                 merged.append(right[right_index])
483                 right_index += 1
484
485         merged.extend(left[left_index:])
486         merged.extend(right[right_index:])
487
488         return merged
```

```
PS E:\3rd Year\2nd Sem\VAI Assisted Coding\Assignment_12.5> python -u "e:\3r
d Year\2nd Sem\VAI Assisted Coding\Assignment_12.5\Assignment_12.5.py"
Search for crop ID 2: {'crop_id': 2, 'crop_name': 'Corn', 'soil_moisture_le
vel': 40, 'temperature': 28, 'yield_estimate': 1500}
Crops sorted by soil moisture level: [{'crop_id': 1, 'crop_name': 'Wheat',
'soil_moisture_level': 30, 'temperature': 25, 'yield_estimate': 1000}, {'cr
op_id': 3, 'crop_name': 'Rice', 'soil_moisture_level': 35, 'temperature': 2
7, 'yield_estimate': 1200}, {'crop_id': 2, 'crop_name': 'Corn', 'soil moist
ure_level': 40, 'temperature': 28, 'yield_estimate': 1500}]
Crops sorted by yield estimate: [{'crop_id': 1, 'crop_name': 'Wheat', 'soil
_moisture_level': 30, 'temperature': 25, 'yield_estimate': 1000}, {'crop_id
': 3, 'crop_name': 'Rice', 'soil_moisture_level': 35, 'temperature': 27, 'y
ield_estimate': 1200}, {'crop_id': 2, 'crop_name': 'Corn', 'soil_moisture_l
evel': 40, 'temperature': 28, 'yield_estimate': 1500}]
PS E:\3rd Year\2nd Sem\VAI Assisted Coding\Assignment_12.5>
```

```
def merge(left, right, key):
    merged = []
    left_index = right_index = 0

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

    merged.extend(left[left_index:])
    merged.extend(right[right_index:])

    return merged

# Example usage:
if __name__ == "__main__":
    system = CropMonitoringSystem()
    system.add_crop(1, 'wheat', 30, 25, 1000)
    system.add_crop(2, 'corn', 40, 28, 1500)
    system.add_crop(3, 'rice', 35, 27, 1200)

    print("Search for crop ID 2:", system.search_crop(2))
    system.sort_crops_by_soil_moisture()
    print("Crops sorted by soil moisture level:", system.crops)
    system.sort_crops_by_yield_estimate()
    print("Crops sorted by yield estimate:", system.crops)
```

```
PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_12.5> python -u "e:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_12.5\Assignment_12.5.py"
Search for crop ID 2: {'crop_id': 2, 'crop_name': 'corn', 'soil_moisture_level': 40, 'temperature': 28, 'yield_estimate': 1500}
Crops sorted by soil moisture level: [{'crop_id': 1, 'crop_name': 'wheat', 'soil_moisture_level': 30, 'temperature': 25, 'yield_estimate': 1000}, {'crop_id': 3, 'crop_name': 'rice', 'soil_moisture_level': 35, 'temperature': 27, 'yield_estimate': 1200}, {'crop_id': 2, 'crop_name': 'corn', 'soil_moisture_level': 40, 'temperature': 28, 'yield_estimate': 1500}]
Crops sorted by yield estimate: [{'crop_id': 1, 'crop_name': 'wheat', 'soil_moisture_level': 30, 'temperature': 25, 'yield_estimate': 1000}, {'crop_id': 3, 'crop_name': 'rice', 'soil_moisture_level': 35, 'temperature': 27, 'yield_estimate': 1200}, {'crop_id': 2, 'crop_name': 'corn', 'soil_moisture_level': 40, 'temperature': 28, 'yield_estimate': 1500}]
PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_12.5>
```

Explanation:

Binary Search enables efficient crop lookup after sorting by crop ID. Sorting by moisture or yield estimate helps farmers analyze data effectively. These algorithms ensure optimized performance for agricultural monitoring systems. AI-supported reasoning guided appropriate algorithm selection.

Task 8: Airport Flight Management System

Prompt:

Generate Python code for an Airport Flight Management System storing flight_id, airline_name, departure_time, arrival_time, and status.

The system must:

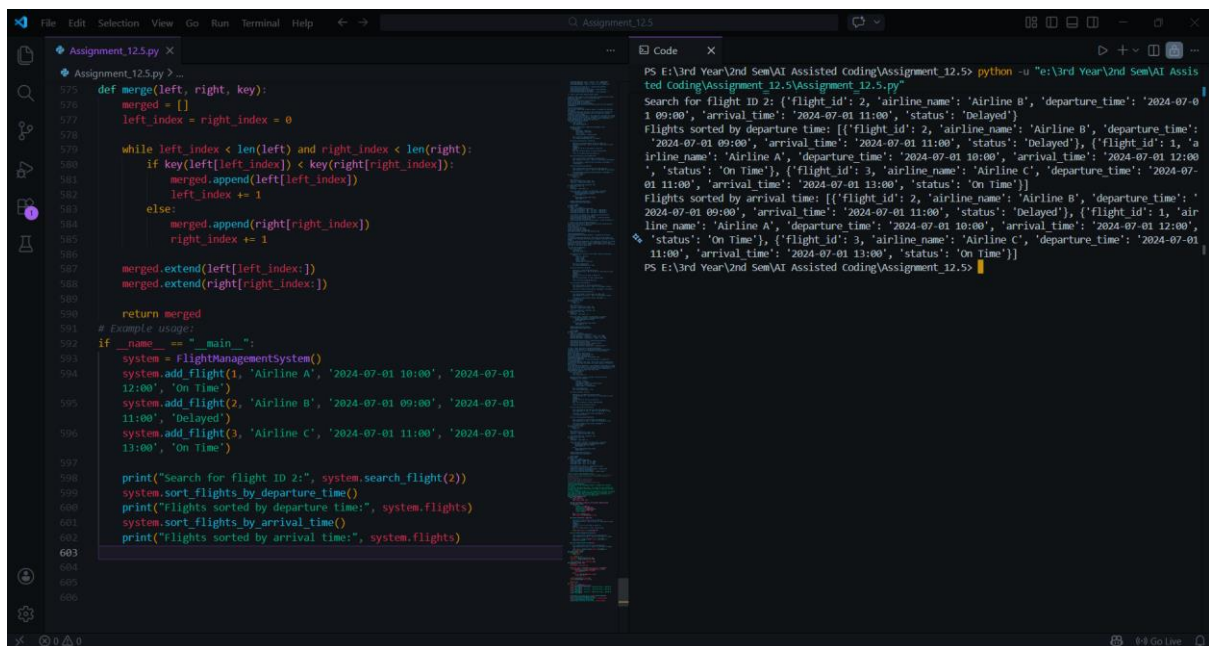
- Search flight details using flight_id
- Sort flights based on departure_time
- Sort flights based on arrival_time

Recommend suitable algorithms and implement them.

Code & Output:


```
File Edit Selection View Go Run Terminal Help Assignment_12.5
Assignment_12.5.py
315 "Task 8: Airport Flight Management System"
316 # Generate Python code for an Airport Flight Management System storing
317 # flight_id, airline_name, departure_time, arrival_time, and status.
318 # The system must:
319 # Search flight details using flight_id
320 # Sort flights based on departure_time
321 # Recommend suitable algorithms and implement them.
322 "Algorithm Recommendation:"
323 "For the Airport Flight Management System, I recommend the following
324 algorithms:"
325 "1. Searching Algorithm: Hash Table - This allows for O(1) average time
326 complexity for searching flight details by flight_id, making it efficient
327 for quick lookups."
328 "2. Sorting Algorithm: Merge Sort - This algorithm is efficient for sorting
329 large lists of flights based on departure_time or arrival_time. It has a
330 time complexity of O(n log n) and is stable, which is beneficial for
331 maintaining the order of records with equal keys."
332 class FlightManagementSystem:
333     def __init__(self):
334         self.flights = []
335         self.flight_id_map = {}
336
337     def add_flight(self, flight_id, airline_name, departure_time,
338 arrival_time, status):
339         flight = {
340             'flight_id': flight_id,
341             'airline_name': airline_name,
342             'departure_time': departure_time,
343             'arrival_time': arrival_time,
344             'status': status
345         }
346         self.flights.append(flight)
347         self.flight_id_map[flight_id] = flight
348
349     def search_flight(self, flight_id):
350         ...
351
352 PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_12.5> python -u "e:\3rd Year\2nd Sem\AI Assis
353 ted Coding\Assignment_12.5\Assignment_12.5.py"
354 Search for flight ID 2: ('flight_id': 2, 'airline_name': 'Airline B', 'departure_time': '2024-07-0
355 1 09:00', 'arrival_time': '2024-07-01 11:00', 'status': 'Delayed')
356 Flights sorted by departure time: [('flight_id': 2, 'airline_name': 'Airline B', 'departure_time':
357 '2024-07-01 09:00', 'arrival_time': '2024-07-01 11:00', 'status': 'Delayed'), ('flight_id': 1, 'a
358 irline_name': 'Airline A', 'departure_time': '2024-07-01 10:00', 'arrival_time': '2024-07-01 12:00
359 ', 'status': 'On Time'), ('flight_id': 3, 'airline_name': 'Airline C', 'departure_time': '2024-07-
360 01 11:00', 'arrival_time': '2024-07-01 13:00', 'status': 'On Time')]
361 Flights sorted by arrival time: [('flight_id': 2, 'airline_name': 'Airline B', 'departure_time': '
362 2024-07-01 09:00', 'arrival_time': '2024-07-01 11:00', 'status': 'Delayed'), ('flight_id': 1, 'air
363 line_name': 'Airline A', 'departure_time': '2024-07-01 10:00', 'arrival_time': '2024-07-01 12:00',
364 ', 'status': 'On Time'), ('flight_id': 3, 'airline_name': 'Airline C', 'departure_time': '2024-07-01
365 11:00', 'arrival_time': '2024-07-01 13:00', 'status': 'On Time')]
366 PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_12.5>
```

```
File Edit Selection View Go Run Terminal Help Assignment_12.5
Assignment_12.5.py
352 class FlightManagementSystem:
353     def search_flight(self, flight_id):
354         ...
355         Searches for flight details using the flight ID.
356         Time complexity: O(1) - due to the use of a hash table for quick
357         lookups.
358         Parameters:
359         flight_id: The ID of the flight to search for.
360         Returns:
361         dict: The flight details if found, otherwise None.
362         ...
363         return self.flight_id_map.get(flight_id)
364
365     def sort_flights_by_departure_time(self):
366         ...
367         Sorts flights based on departure time using Merge Sort.
368         Time complexity: O(n log n) - where n is the number of flights.
369         ...
370         self.flights = merge_sort(self.flights, key=lambda x: x
371 ['departure_time'])
372
373     def sort_flights_by_arrival_time(self):
374         ...
375         Sorts flights based on arrival time using Merge Sort.
376         Time complexity: O(n log n) - where n is the number of flights.
377         ...
378         self.flights = merge_sort(self.flights, key=lambda x: x
379 ['arrival_time'])
380
381     def merge_sort(self, arr, key):
382         if len(arr) <= 1:
383             return arr
384
385         mid = len(arr) // 2
386         left_half = merge_sort(arr[:mid], key)
387         right_half = merge_sort(arr[mid:], key)
388
389         return merge(left_half, right_half, key)
```



```
def merge(left, right, key):
    merged = []
    left_index = right_index = 0

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

    merged.extend(left[left_index:])
    merged.extend(right[right_index:])

    return merged

# Example usage:
if __name__ == "__main__":
    system = FlightManagementSystem()
    system.add_flight(1, 'Airline A', '2024-07-01 10:00', '2024-07-01 12:00', 'On Time')
    system.add_flight(2, 'Airline B', '2024-07-01 09:00', '2024-07-01 11:00', 'Delayed')
    system.add_flight(3, 'Airline C', '2024-07-01 11:00', '2024-07-01 13:00', 'On Time')

    print("Search for flight ID 2:", system.search_flight(2))
    system.sort_flights_by_departure_time()
    print("Flights sorted by departure time:", system.flights)
    system.sort_flights_by_arrival_time()
    print("Flights sorted by arrival time:", system.flights)
```

```
PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_12.5> python -u "e:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_12.5\Assignment_12.5.py"
Search for flight ID 2: {'flight_id': 2, 'airline_name': 'Airline B', 'departure_time': '2024-07-01 09:00', 'arrival_time': '2024-07-01 11:00', 'status': 'Delayed'}
Flights sorted by departure time: [{'flight_id': 2, 'airline_name': 'Airline B', 'departure_time': '2024-07-01 09:00', 'arrival_time': '2024-07-01 11:00', 'status': 'Delayed'}, {'flight_id': 1, 'airline_name': 'Airline A', 'departure_time': '2024-07-01 10:00', 'arrival_time': '2024-07-01 12:00', 'status': 'On Time'}, {'flight_id': 3, 'airline_name': 'Airline C', 'departure_time': '2024-07-01 11:00', 'arrival_time': '2024-07-01 13:00', 'status': 'On Time'}]
Flights sorted by arrival time: [{'flight_id': 2, 'airline_name': 'Airline B', 'departure_time': '2024-07-01 09:00', 'arrival_time': '2024-07-01 11:00', 'status': 'Delayed'}, {'flight_id': 1, 'airline_name': 'Airline A', 'departure_time': '2024-07-01 10:00', 'arrival_time': '2024-07-01 12:00', 'status': 'On Time'}, {'flight_id': 3, 'airline_name': 'Airline C', 'departure_time': '2024-07-01 11:00', 'arrival_time': '2024-07-01 13:00', 'status': 'On Time'}]
PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_12.5>
```

Explanation:

Binary Search ensures fast flight lookup when sorted by flight ID. Sorting by departure or arrival time organizes airport schedules effectively. These algorithms provide efficient record management. AI assistance supported correct algorithm implementation and modular function design.

Final Conclusion:

This lab demonstrated the implementation of sorting and searching algorithms using AI assistance across multiple real-world scenarios. Binary Search was selected for efficient searching due to its logarithmic time complexity, while Merge Sort was chosen for stable and reliable sorting performance. AI tools supported algorithm recommendation, code generation, and optimization. However, understanding algorithm efficiency and selecting appropriate techniques remain the developer's responsibility to ensure correct and scalable solutions.