AI Assisted Coding

Assignment 12.5

Name: ch .koushik

Hall ticket no: 2303A51938

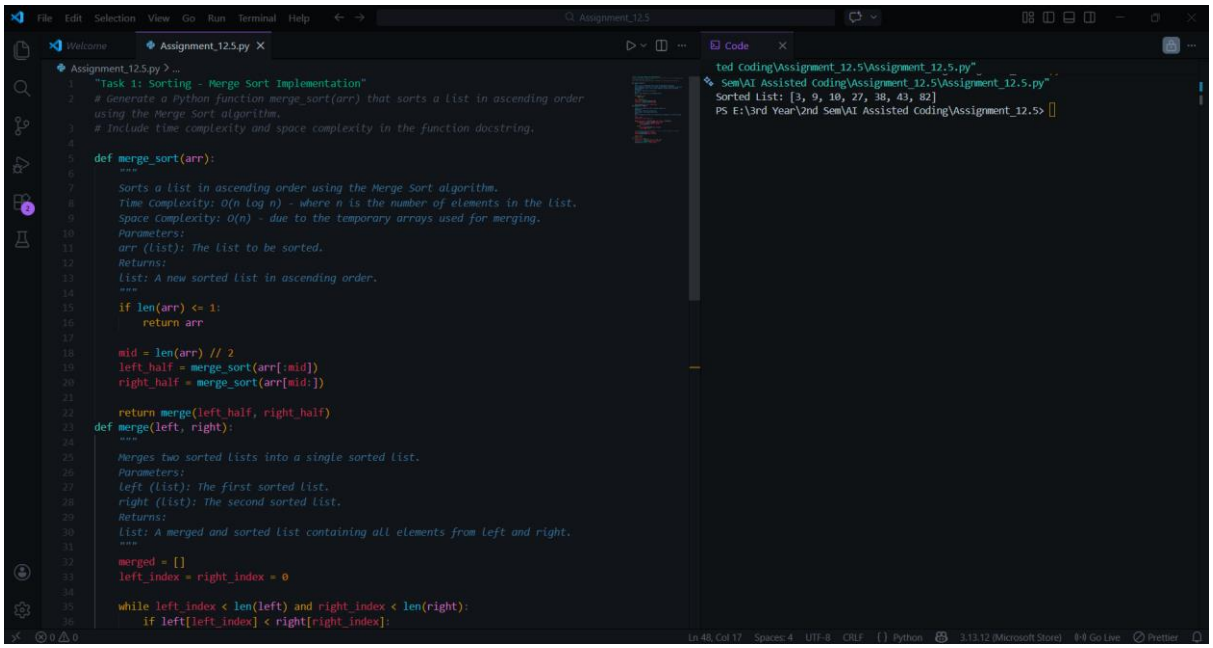
Batch no: 19

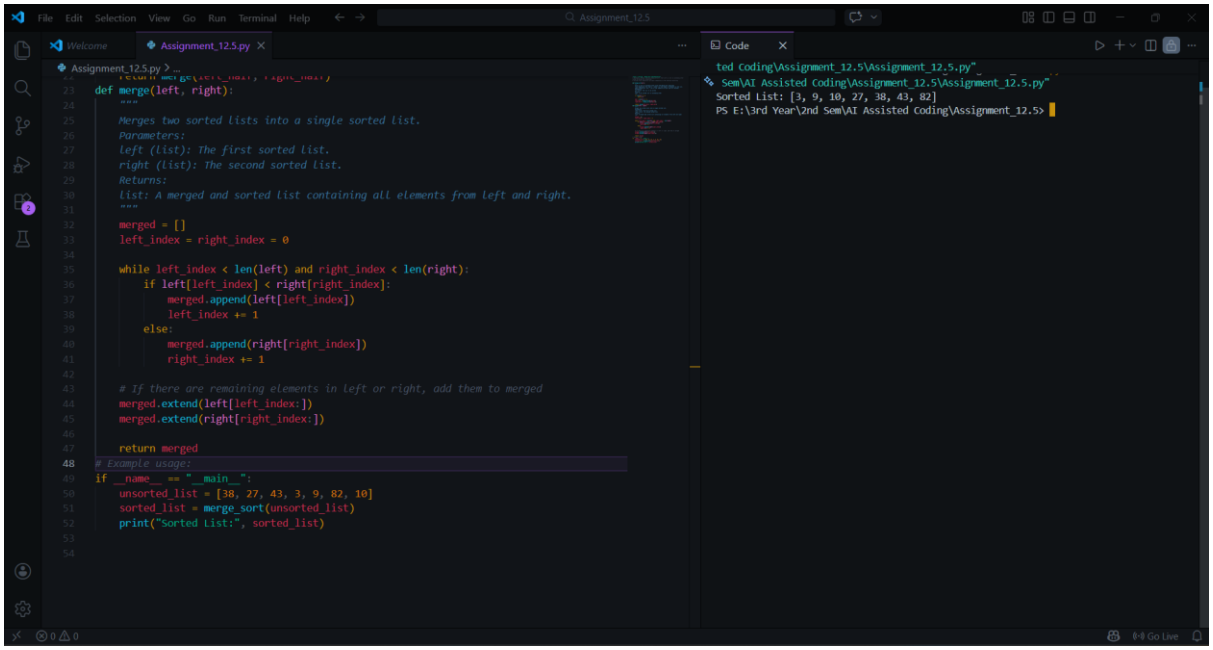
Task 1: Sorting – Merge Sort Implementation

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:





Explanation:

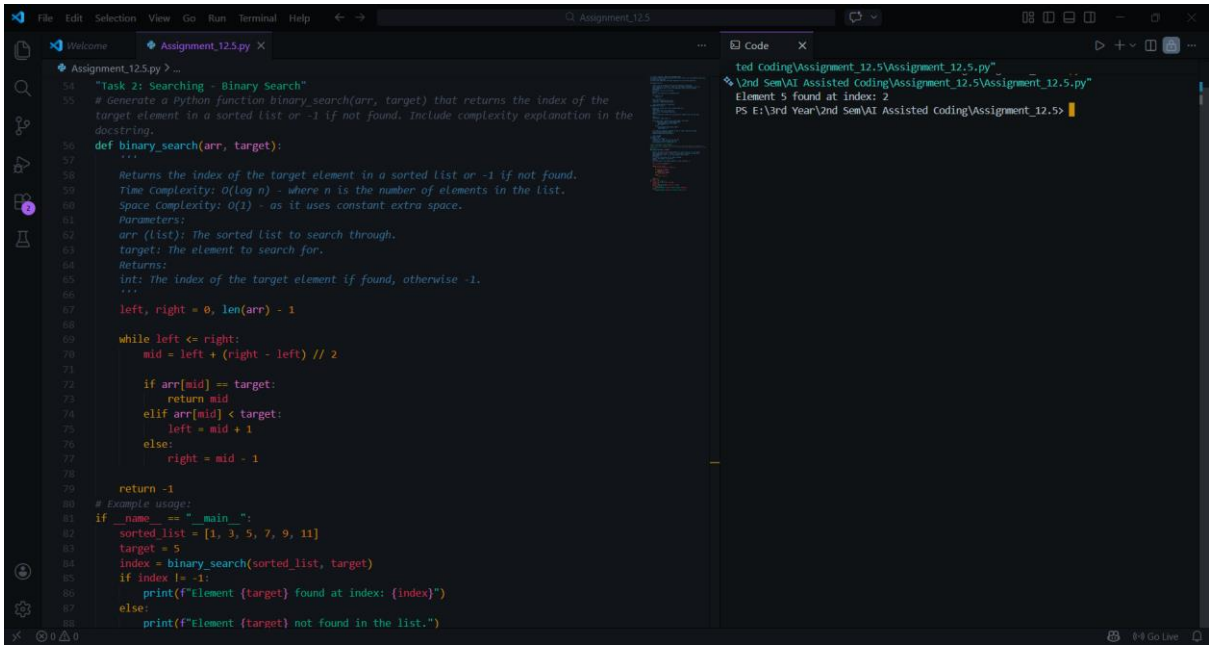
Merge Sort divides the list into smaller halves recursively and then merges them in sorted order. It guarantees O(n log n) time complexity in all cases, making it efficient for large datasets. The algorithm requires additional memory for merging, resulting in O(n) space complexity. AI assistance helped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



Explanation:

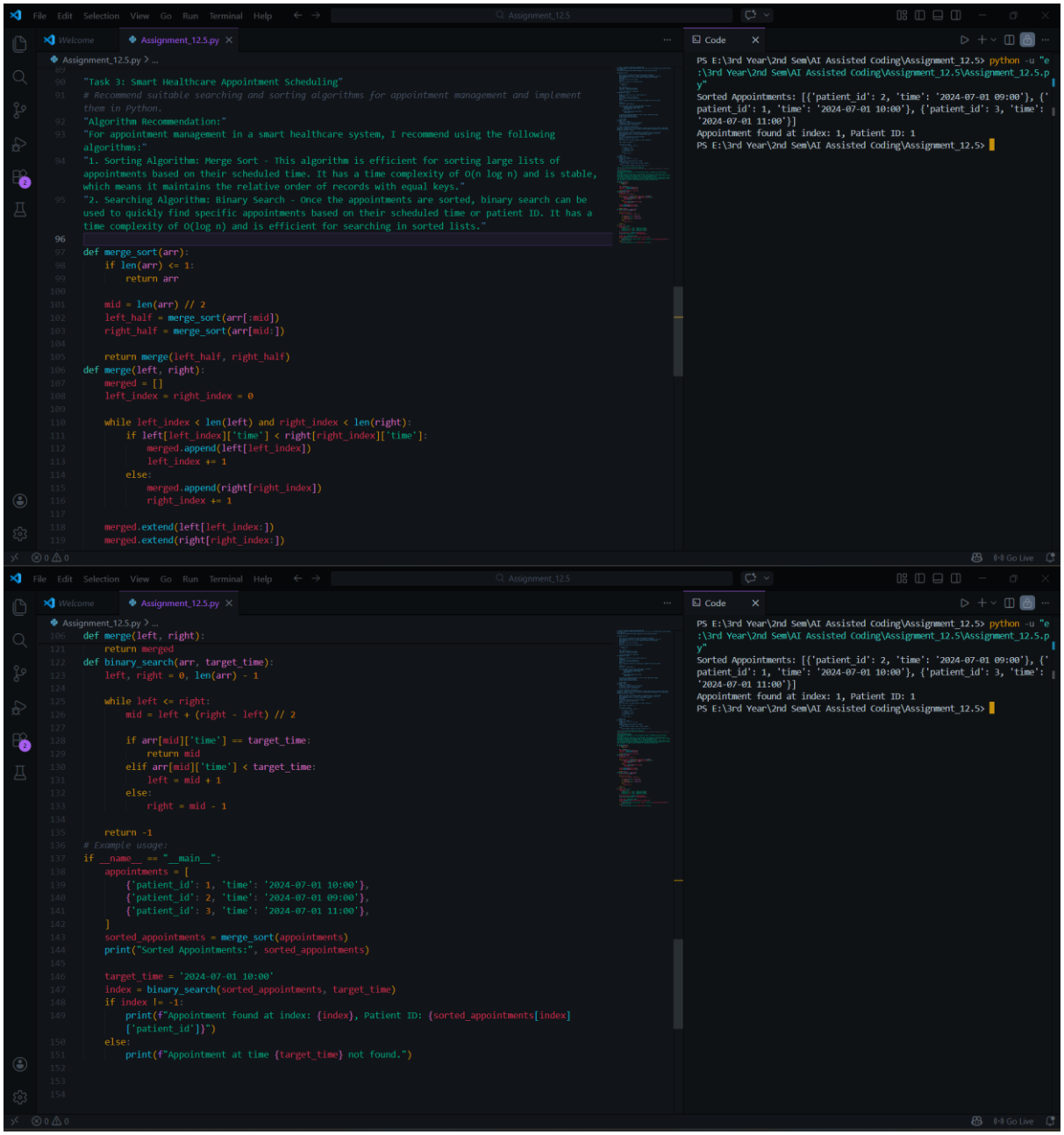
Binary Search works by repeatedly dividing the sorted list into halves. It significantly reduces the search space, resulting in logarithmic time complexity. It is efficient compared to linear search for large sorted datasets. AI assistance helped structure the iterative logic and complexity documentation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

Task 3: Smart Healthcare Appointment Scheduling

Prompt:

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

Code & Output:



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:

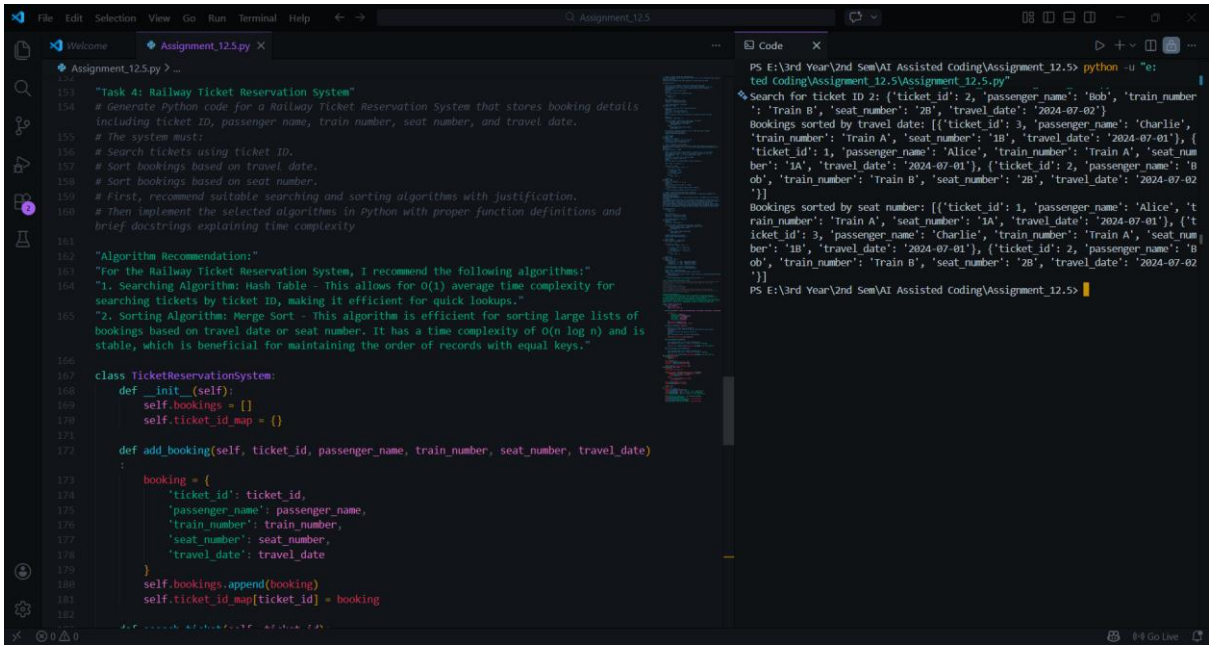
Generate Python code for a Railway Ticket Reservation System that stores booking details including ticket ID, passenger name, train number, seat number, and travel date.

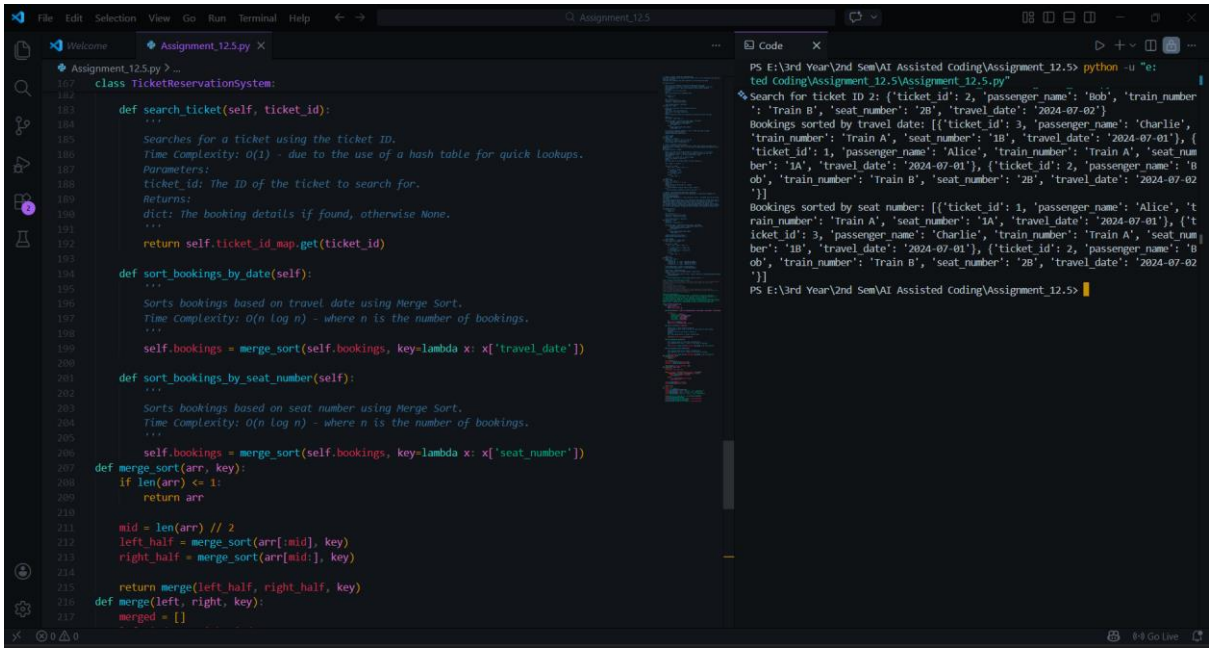
The system must:

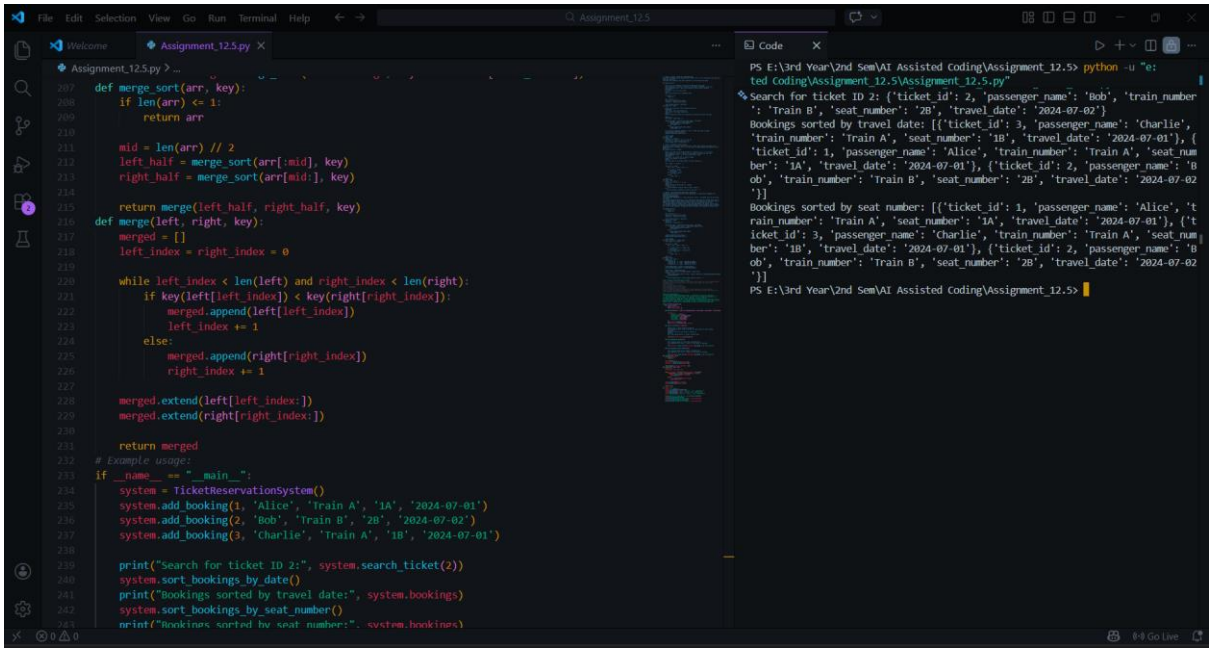
1. Search tickets using ticket ID.
2. Sort bookings based on travel date.
3. Sort bookings based on seat number.

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:







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

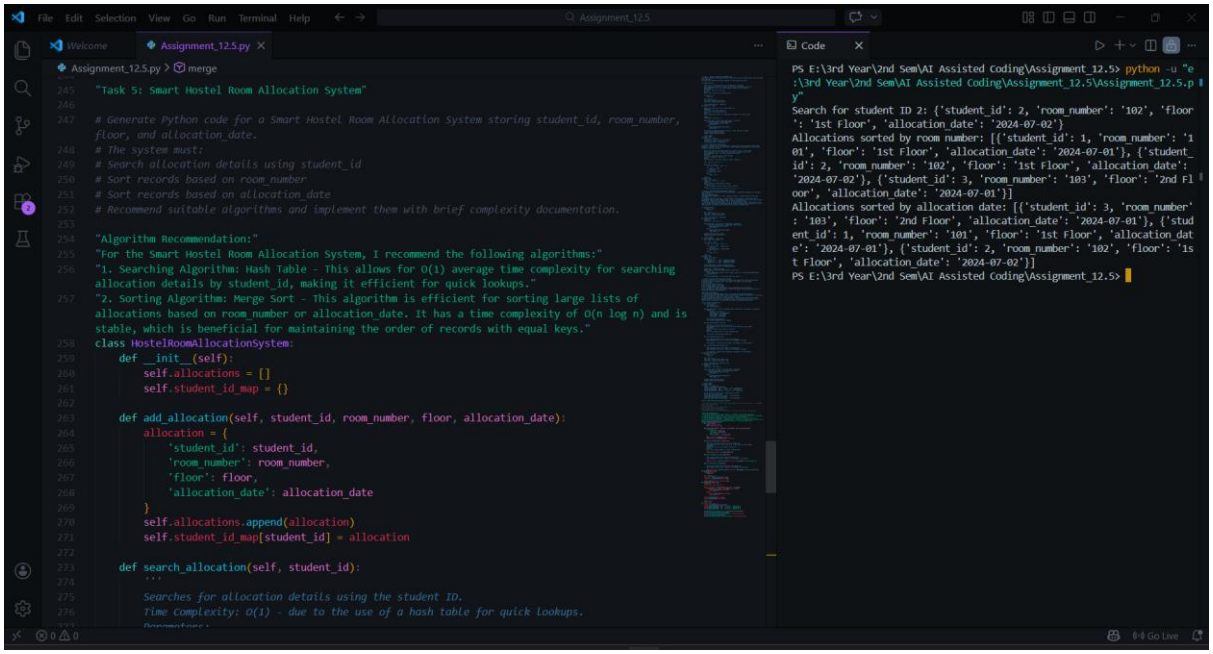
Prompt:

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

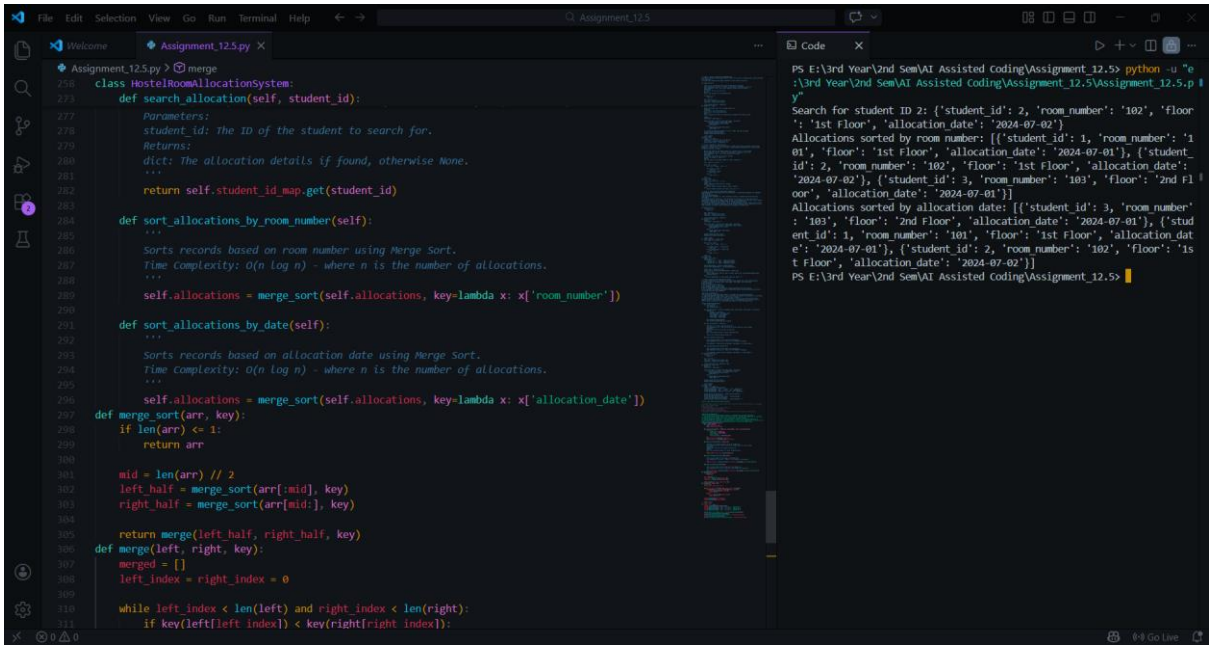
Recommend suitable algorithms and implement them with brief complexity documentation.

Code & Output:



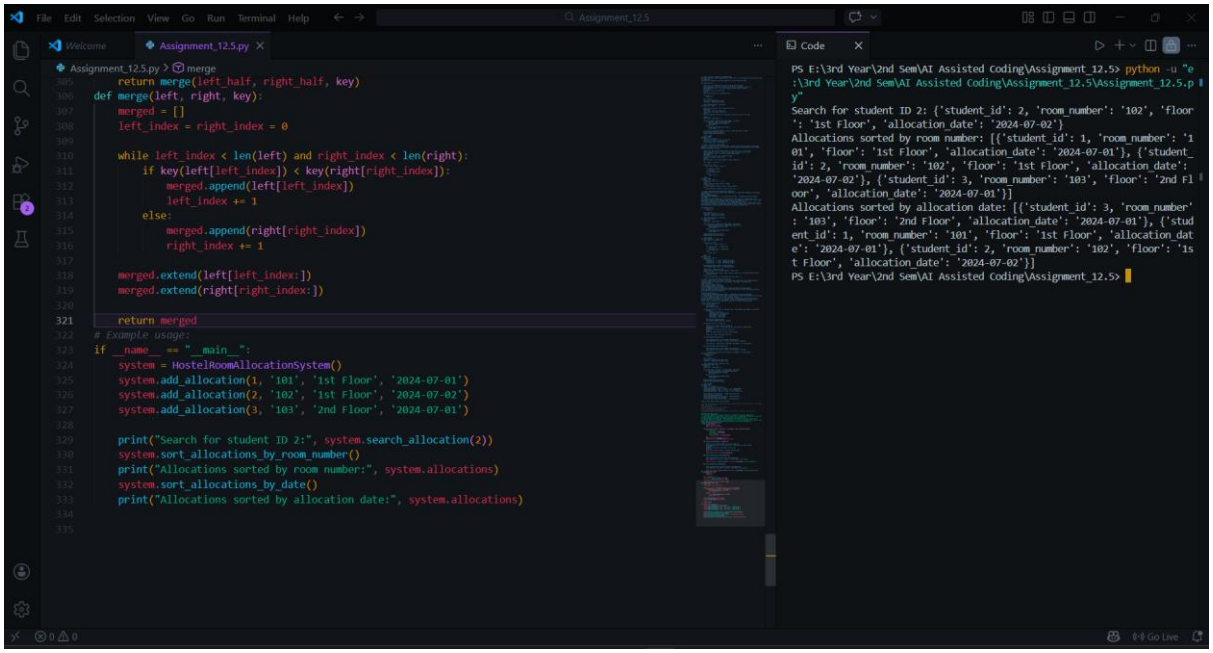
```
"Task 5: Smart Hostel Room Allocation System"

# 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.
```

```
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.p
y"
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'}, {'student_
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>
```



```
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]):
```

```
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.p
y"
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'}, {'student_
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

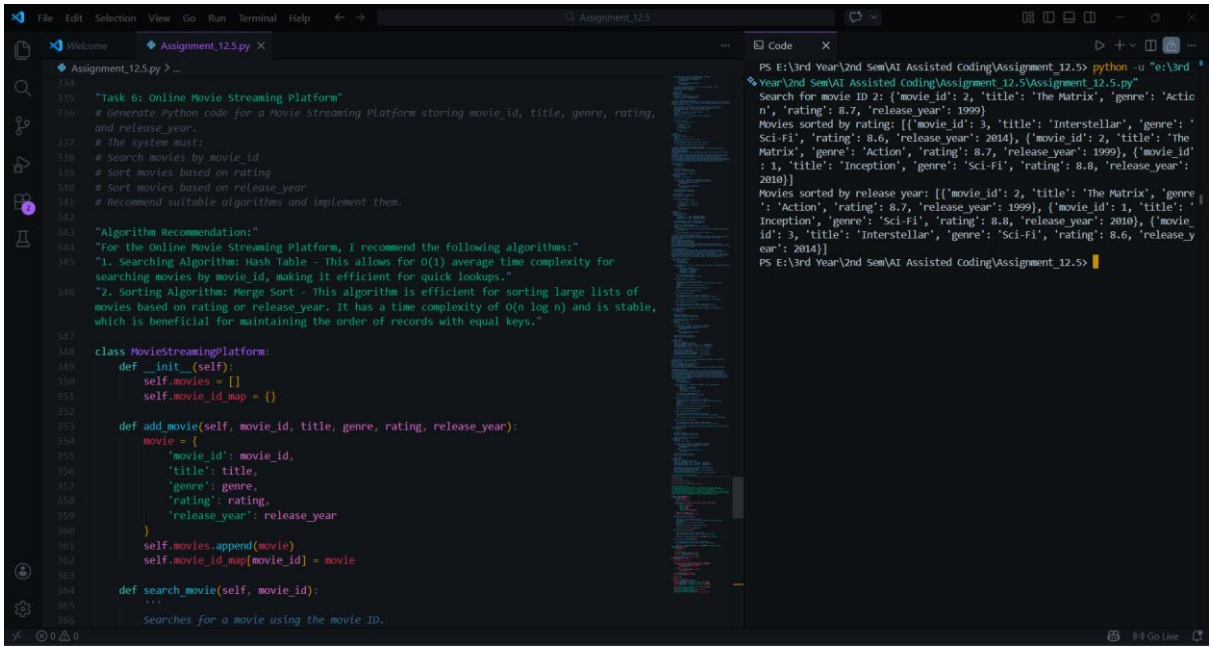
Prompt:

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

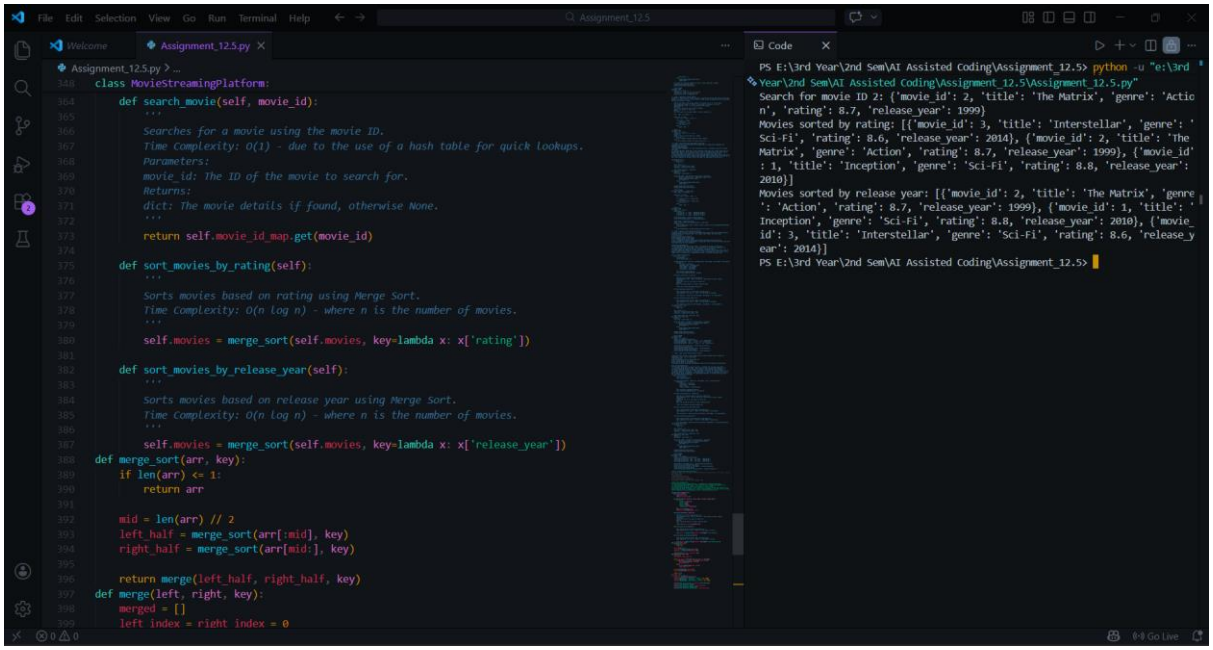
Recommend suitable algorithms and implement them.

Code & Output:



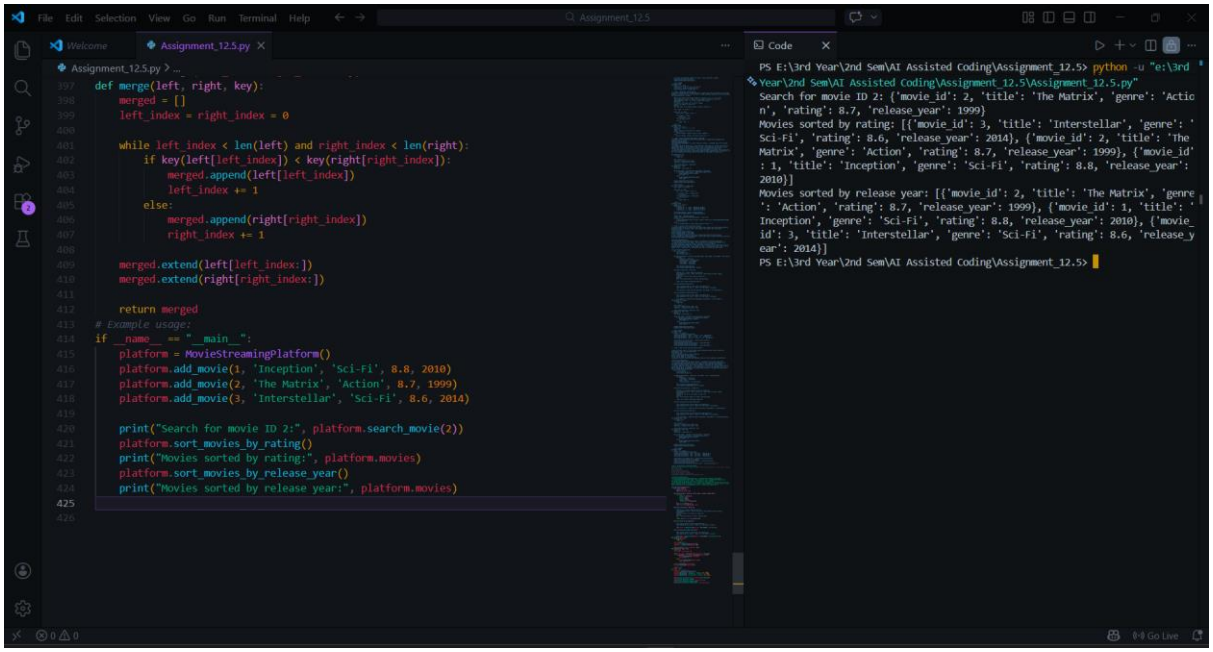
```
"Task 6: Online Movie Streaming Platform"
# Generate Python code for a Movie Streaming Platform storing movie_id, title, genre, rating,
and release_year.
# The system must:
# Search movies by movie_id
# Sort movies based on rating
# Sort movies based on release_year
# Recommend suitable algorithms and implement them.

"Algorithm Recommendation:"
"For the Online Movie Streaming Platform, I recommend the following algorithms:"
"1. Searching Algorithm: Hash Table - This allows for O(1) average time complexity for
searching movies by movie_id, making it efficient for quick lookups."
"2. Sorting Algorithm: 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.
```

```
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': 'Actio
n', '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_y
ear': 2014}]
PS E:\3rd Year\2nd Sem\AI Assisted Coding\Assignment_12.5>
```



```
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
```

```
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': 'Actio
n', '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_y
ear': 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

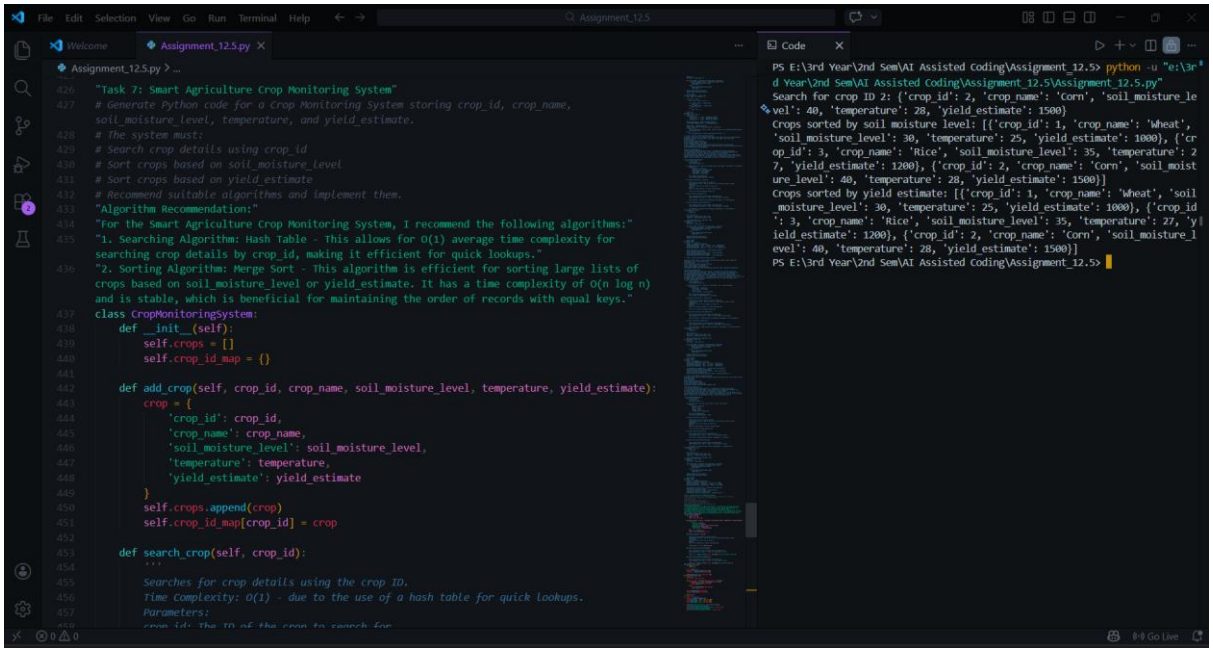
Prompt:

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

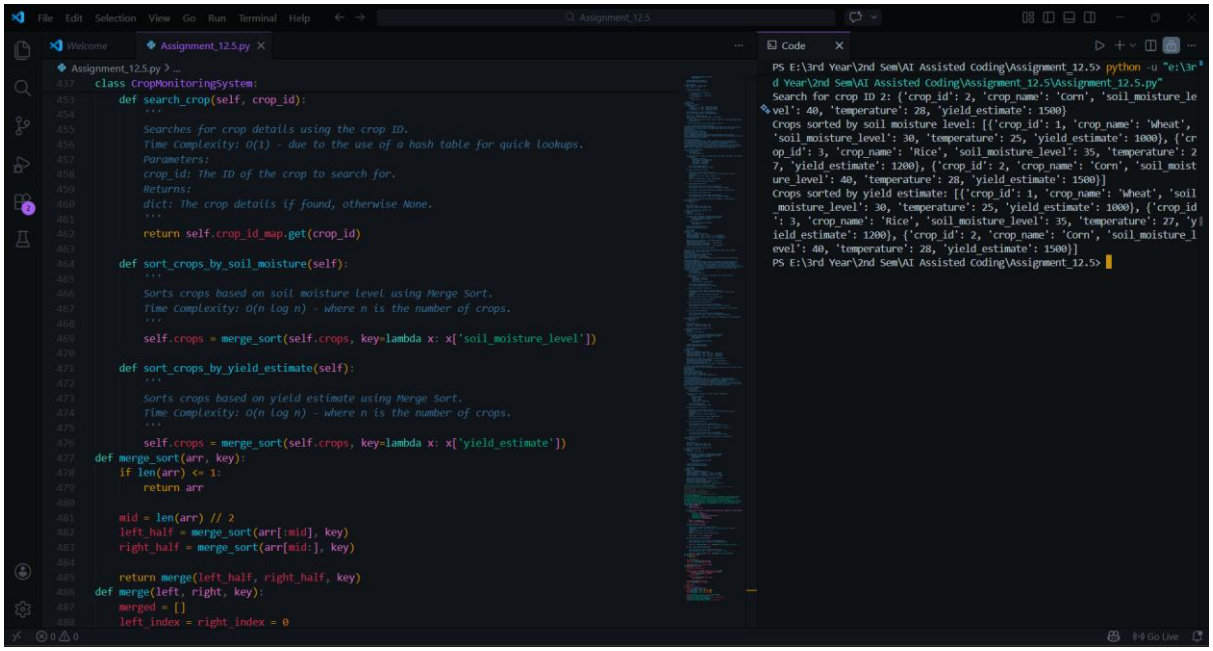
Recommend suitable algorithms and implement them.

Code & Output:



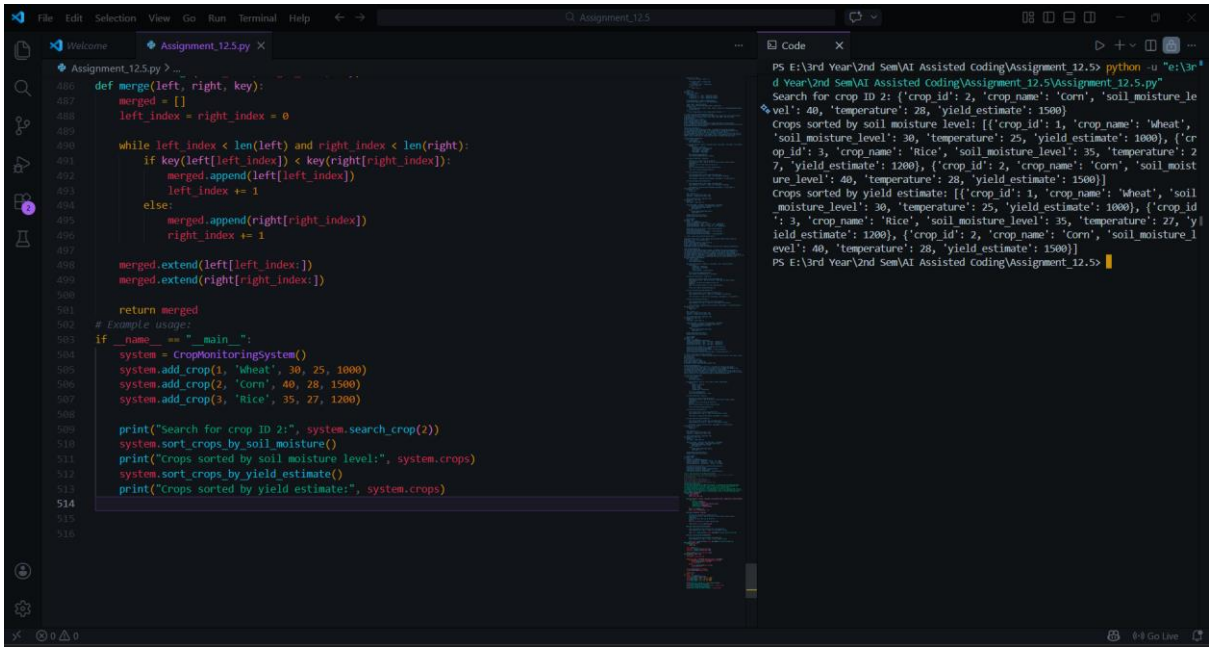
```
"Task 7: Smart Agriculture Crop Monitoring System"
# Generate Python code for a Crop Monitoring System storing crop_id, crop_name,
soil_moisture_level, temperature, and yield_estimate.
# The system must:
# Search crop details using crop_id
# Sort crops based on soil_moisture_level
# Sort crops based on yield_estimate
# Recommend suitable algorithms and implement them.
"Algorithm Recommendation:"
"For the Smart Agriculture Crop Monitoring System, I recommend the following algorithms:"
"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."
"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."
class CropMonitoringSystem:
    def __init__(self):
        self.crops = []
        self.crop_id_map = {}

    def add_crop(self, crop_id, crop_name, soil_moisture_level, temperature, yield_estimate):
        crop = {
            'crop_id': crop_id,
            'crop_name': crop_name,
            'soil_moisture_level': soil_moisture_level,
            'temperature': temperature,
            'yield_estimate': yield_estimate
        }
        self.crops.append(crop)
        self.crop_id_map[crop_id] = crop

    def search_crop(self, crop_id):
        '''
        Searches for crop details using the crop ID.
        Time Complexity: O(1) - due to the use of a hash table for quick lookups.
        Parameters:
```

```
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>
```



```
class CropMonitoringSystem:
    def search_crop(self, crop_id):
        '''
        Searches for crop details using the crop ID.
        Time Complexity: O(1) - due to the use of a hash table for quick lookups.
        Parameters:
        crop_id: The ID of the crop to search for.
        Returns:
        dict: The crop details if found, otherwise None.
        '''
        return self.crop_id_map.get(crop_id)

    def sort_crops_by_soil_moisture(self):
        '''
        Sorts crops based on soil moisture level using Merge Sort.
        Time Complexity: O(n Log n) - where n is the number of crops.
        '''
        self.crops = merge_sort(self.crops, key=lambda x: x['soil_moisture_level'])

    def sort_crops_by_yield_estimate(self):
        '''
        Sorts crops based on yield estimate using Merge Sort.
        Time Complexity: O(n Log n) - where n is the number of crops.
        '''
        self.crops = merge_sort(self.crops, key=lambda x: x['yield_estimate'])
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
```

```
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

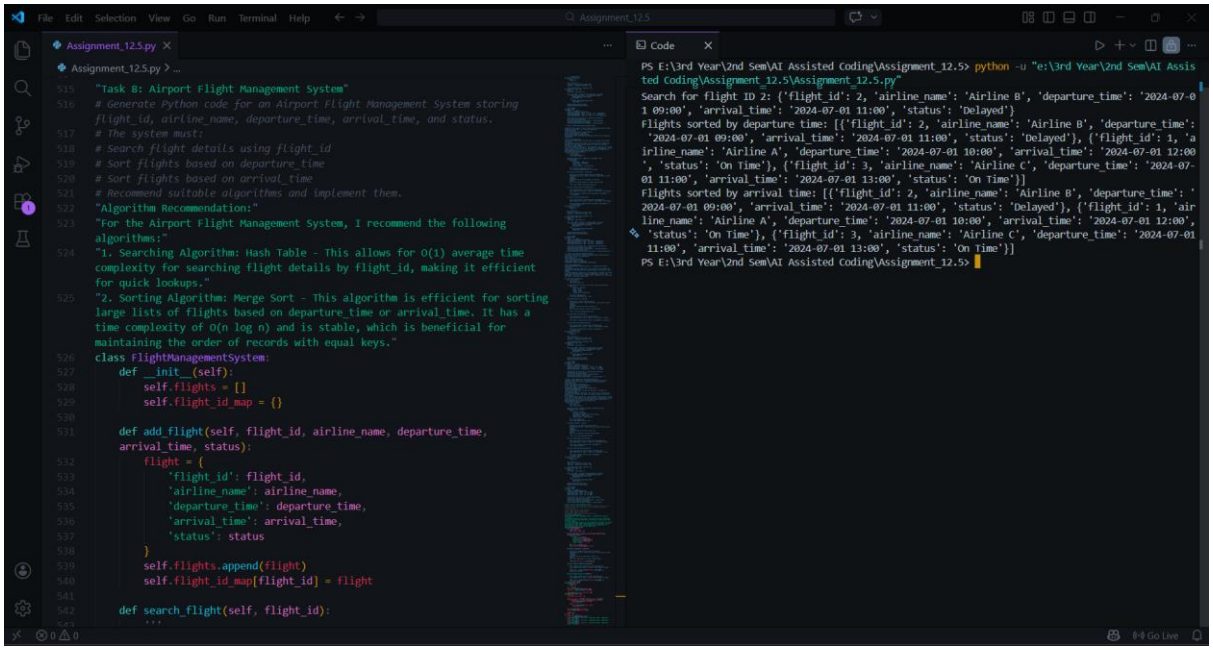
Prompt:

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

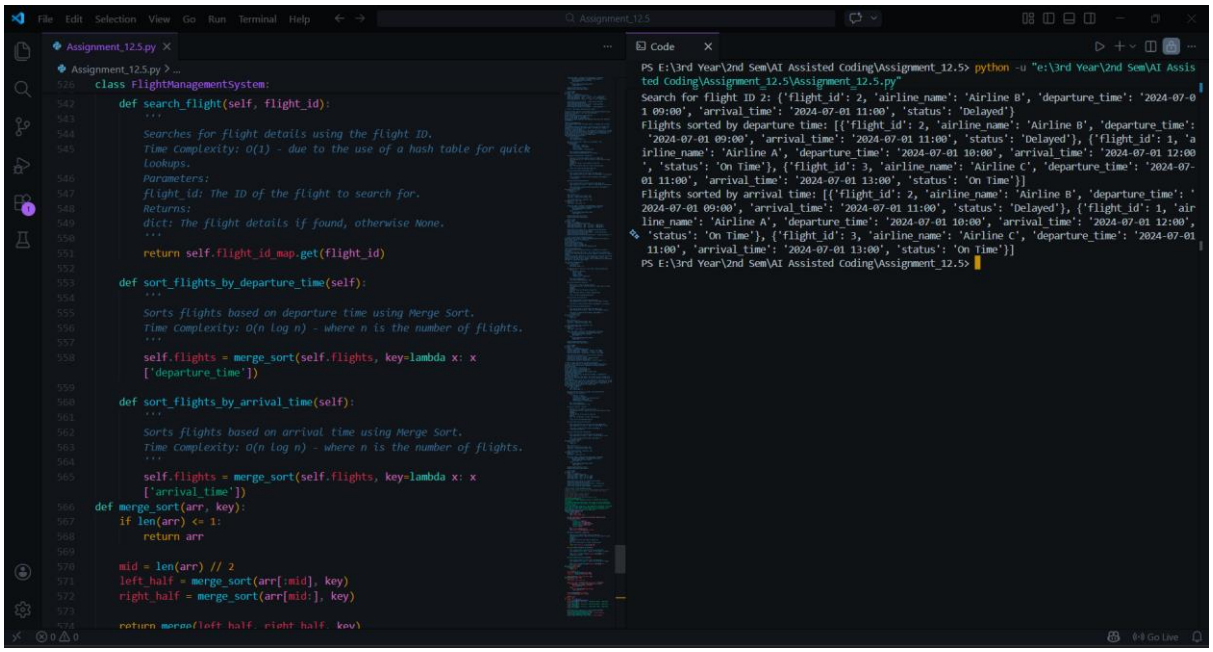
Recommend suitable algorithms and implement them.

Code & Output:



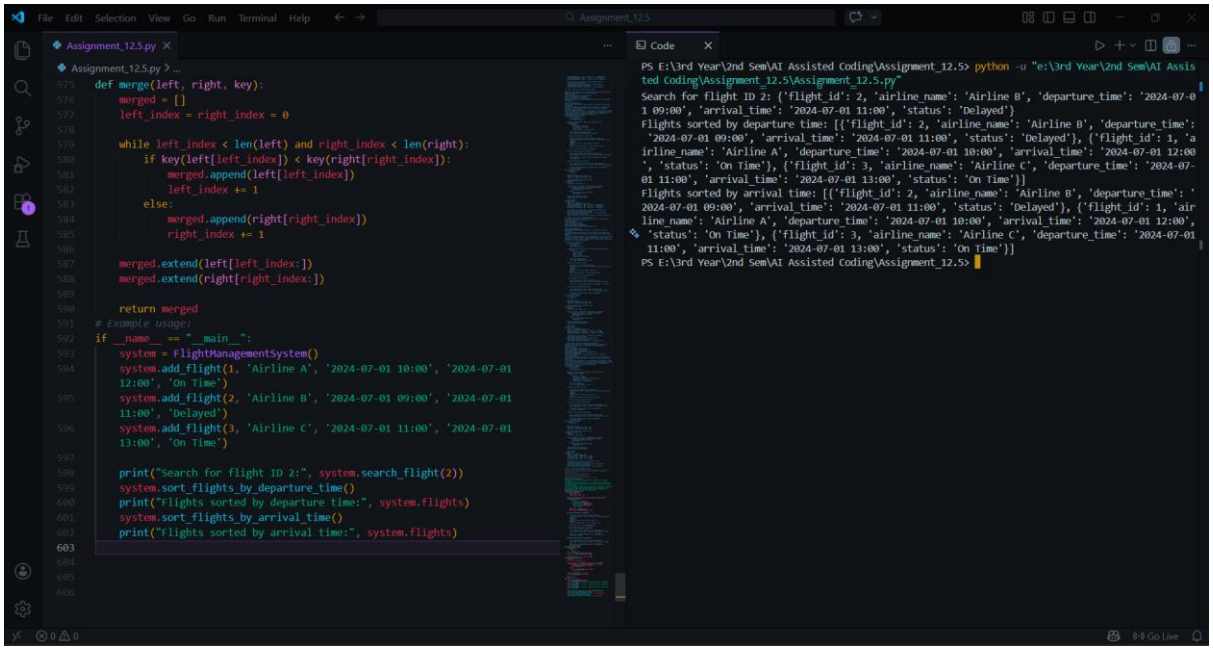
```
"Task 8: Airport Flight Management System"
# Generate Python code for an Airport Flight Management System storing
flight_id, airline_name, departure_time, arrival_time, and status.
# The system must:
# Search flight details using flight_id
# Sort flights based on departure_time
# Sort flights based on arrival_time
# Recommend suitable algorithms and implement them.
"Algorithm Recommendation:"
"For the Airport Flight Management System, I recommend the following
algorithms:"
"1. Searching Algorithm: Hash Table - This allows for O(1) average time
complexity for searching flight details by flight_id, making it efficient
for quick lookups."
"2. Sorting Algorithm: Merge Sort - This algorithm is efficient for sorting
large lists of flights based on departure_time or arrival_time. 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 FlightManagementSystem:
    def __init__(self):
        self.flights = []
        self.flight_id_map = {}

    def add_flight(self, flight_id, airline_name, departure_time,
    arrival_time, status):
        flight = {
            'flight_id': flight_id,
            'airline_name': airline_name,
            'departure_time': departure_time,
            'arrival_time': arrival_time,
            'status': status
        }
        self.flights.append(flight)
        self.flight_id_map[flight_id] = flight

    def search_flight(self, flight_id):
        '''
```

```
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>
```



```
class FlightManagementSystem:
    def search_flight(self, flight_id):
        '''
        Searches for flight details using the flight ID.
        Time Complexity: O(1) - due to the use of a hash table for quick
        lookups.
        Parameters:
        flight_id: The ID of the flight to search for.
        Returns:
        dict: The flight details if found, otherwise None.
        '''
        return self.flight_id_map.get(flight_id)

    def sort_flights_by_departure_time(self):
        '''
        Sorts flights based on departure time using Merge Sort.
        Time Complexity: O(n log n) - where n is the number of flights.
        '''
        self.flights = merge_sort(self.flights, key=lambda x: x
        ['departure_time'])

    def sort_flights_by_arrival_time(self):
        '''
        Sorts flights based on arrival time using Merge Sort.
        Time Complexity: O(n log n) - where n is the number of flights.
        '''
        self.flights = merge_sort(self.flights, key=lambda x: x
        ['arrival_time'])
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)
```

```
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.