Lab Assignment-12.5

Algorithms with AI Assistance – Sorting, Searching, and Optimizing Algorithms

Name:- G Datha Narayana

Ht.no:- 2303A51754

Batch:- 11

Task Description #1 (Sorting – Merge Sort Implementation)

• **Task:** Use AI to generate a Python program that implements

the Merge Sort algorithm.

Prompt: #Generate a python code to create a function merge_sort(arr) thatsorts a list in ascending order.Give time complexity and space complexity of the code in comments, use pydoc to generate documentation

Code:

```
#Generate a python code to create a function merge_sort(arr) thatsorts a list in ascending order.Give time complexity and space complexity of the code in comments, use pydoc to generate documentation
def merge_sort(arr):
    """
    This function implements the merge sort algorithm to sort a list in ascending order.
    Merge sort is a divide-and-conquer algorithm that divides the input list into two halves,
    recursively sorts both halves, and then merges the sorted halves back together.
    Parameters:
    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) - where n is the number of elements in the list. This is because the list is divided log n times and each merge operation takes O(n) time.
    Space Complexity: O(n) - due to the additional space required for the temporary arrays used during the merge process.
    Example:
    >>> merge_sort([38, 27, 43, 3, 9, 82, 10])
    [3, 9, 10, 27, 38, 43, 82]
    >>> merge_sort([5, 2, 9, 1, 5, 6])
    [1, 2, 5, 5, 6, 9]
    >>> merge_sort([])
    []
    >>> merge_sort([1])
    [1]
    """
    if len(arr) <= 1:
        return arr
    # Split the array into two halves
    mid = len(arr) // 2
    left_half = merge_sort(arr[:mid]) # Recursively sort the left half
    right_half = merge_sort(arr[mid:]) # Recursively sort the right half
    # Merge the sorted halves
    return merge(left_half, right_half)
def merge(left, right):
    """This helper function merges two sorted lists into a single sorted list.
    Parameters:
    left (list): The first sorted list.
    right (list): The second sorted list.
    Returns:
    list: A merged sorted list containing all elements from both input lists.
    """
    merged = []
    i = j = 0
    # Merge the two lists while maintaining sorted order
    while i < len(left) and j < len(right):
        # use the __lt__ comparison (or left[i].time < right[j].time)
        if left[i] < right[j]:
            merged.append(left[i]) # Append the smaller element from left
            i += 1
        else:
            merged.append(right[j]) # Append the smaller element from right
            j += 1
    # If there are remaining elements in left, add them to merged
    while i < len(left):
        merged.append(left[i])
        i += 1
    # If there are remaining elements in right, add them to merged
    while j < len(right):
        merged.append(right[j])
        j += 1
    return merged # Return the merged sorted list
print("Sorted list:", merge_sort([38, 27, 43, 3, 9, 82, 10])) # Example usage of the merge_sort function to sort a list of numbers
print("Time Complexity: O(n log n)") # Print the time complexity of the merge sort algorithm
print("Space Complexity: O(n)") # Print the space complexity of the merge sort algorithm
```

Output:

```
Sorted list: [3, 9, 10, 27, 38, 43, 82]
```

Explanation: The code snippet implements a function to check if a given number is a palindrome. It converts the number to a string and compares it with its reverse to determine if it reads the same backward and forward. The function includes a docstring that explains its purpose, parameters, return value, and provides examples of usage.

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.

Prompt: #Generate a python code to create a function binary_search(arr, target) returning the index of the target or -1 if not found.Use docstring to explain best, average and worst case time complexity, use pydoc to generate documentation

Code:

```python

#Generate a python code to create a function binary_search(arr, target) returning the index of the target or -1 if not found.Use docstring to explain best, average and worst case time complexity, use pydoc to generate documentation
def binary_search(arr, target):
    """
    This function implements the binary search algorithm to find the index of a target element in a sorted list.

    Binary search works by repeatedly dividing the search interval in half. If the target value is less than the middle element,
    the search continues in the left half; if it is greater, the search continues in the right half. This process continues until
    the target value is found or the search interval is empty.

    Parameters:
    arr (list): A sorted list of elements to be searched.
    target: The element to be searched for in the list.

    Returns:
    int: The index of the target element if found, or -1 if not found.

    Time Complexity:
    - Best Case: O(1) - when the target element is at the middle of the list.
    - Average Case: O(log n) - where n is the number of elements in the list, due to halving the search space with each iteration.
    - Worst Case: O(log n) - when the target element is not present in the list or is located at one of the ends.

    Example:
    >>> binary_search([1, 2, 3, 4, 5], 3)
    2
    >>> binary_search([1, 2, 3, 4, 5], 6)
    -1
    >>> binary_search([], 1)
    -1
    >>> binary_search([1], 1)
    0
    """
    left, right = 0, len(arr) - 1

    while left <= right:
        mid = left + (right - left) // 2 # Calculate the middle index

        # Check if the target is present at mid
        if arr[mid] == target:
            return mid # Return the index if target is found
        # If target is greater than mid, ignore left half
        elif arr[mid] < target:
            left = mid + 1 # Move left pointer to mid + 1
        # If target is smaller than mid, ignore right half
        else:
            right = mid - 1 # Move right pointer to mid - 1

    return -1 # Return -1 if target is not found in the list
print("Index of target 3:", binary_search([1, 2, 3, 4, 5], 3)) # Example usage of the binary_search function to find the index of target 3
print("Index of target 6:", binary_search([1, 2, 3, 4, 5], 6)) # Example usage of the binary_search function to find the index of target 6
print("Index of target 1 in empty list:", binary_search([], 1)) # Example usage of the binary_search function to find the index of target 1 in an empty list
print("Index of target 1 in single element list:", binary_search([1], 1)) # Example usage of the binary_search function to find the index of target 1 in a single element list


```

OutPut:

```
Index of target 3: 2
Index of target 6: -1
Index of target 1 in empty list: -1
Index of target 1 in single element list: 0
```

Explanation: The code snippet defines a function to calculate the factorial of a non-negative integer. It uses a loop to multiply the result by each integer from 1 to n, and includes a docstring that describes its functionality, parameters, return value, and provides examples of usage.

Task Description #3: Smart Healthcare Appointment Scheduling System

Prompt: #Generate a python code to implement Smart Healthcare Appointment Scheduling System. Give suitable searching and sorting algorithms to optimize the scheduling process. Use docstring to explain the functionality of each function, use pydoc to generate documentation

1.Search appointments using appointment ID.

2. Sort appointments based on time or consultation fee.

Code:

```python
#generate a python code to implement Smart Healthcare Appointment Scheduling System. Give suitable searching and sorting algorithms to optimize the scheduling process. Use docstring to explain the functionality of each function, use pydoc to generate documentation
# 1.Search appointments using appointment ID.
# 2. Sort appointments based on time or consultation fee.
class Appointment:
    """
    This class represents an appointment in the Smart Healthcare Appointment Scheduling System.

    Attributes:
    appointment_id (int): Unique identifier for the appointment.
    patient_name (str): Name of the patient.
    doctor_name (str): Name of the doctor.
    time (str): Time of the appointment in HH:MM format.
    consultation_fee (float): Consultation fee for the appointment.
    """
    def __init__(self, time, description):
        self.time = time
        self.description = description
    # allow Python to compare two appointments by their time
    def __lt__(self, other):
        return self.time < other.time
class AppointmentScheduler:
    """
    This class implements the Smart Healthcare Appointment Scheduling System.

    It provides functionalities to add appointments, search for appointments by ID, and sort appointments based on time or consultation fee.
    """
    def __init__(self):
        self.appointments = [] # Initialize an empty list to store appointments

    def add_appointment(self, appointment):
        """Adds a new appointment to the scheduler.

        Parameters:
        appointment (Appointment): The appointment object to be added to the scheduler.
        """
        self.appointments.append(appointment) # Add the appointment to the list of appointments

    def search_appointment_by_id(self, appointment_id):
        """Searches for an appointment by its ID using linear search.

        Parameters:
        appointment_id (int): The unique identifier of the appointment to be searched.

        Returns:
        Appointment: The appointment object if found, or None if not found.
        """
        for appointment in self.appointments: # Iterate through the list of appointments
            if appointment.appointment_id == appointment_id: # Check if the current appointment's ID matches the target ID
                return appointment # Return the appointment if found
        return None # Return None if no matching appointment is found
```

```python
    def sort_appointments_by_time(self):
        """Sorts the appointments based on time using merge sort algorithm."""
        self.appointments = merge_sort(self.appointments) # Sort the appointments using merge sort based on time attribute

    def sort_appointments_by_fee(self):
        """Sorts the appointments based on consultation fee using merge sort algorithm."""
        self.appointments = sorted(self.appointments, key=lambda x: x.consultation_fee) # Sort the appointments based on consultation fee using built-in sorted function with a lambda key
# Example usage of the AppointmentScheduler class
scheduler = AppointmentScheduler() # Create an instance of the AppointmentScheduler class
scheduler.add_appointment(Appointment(1, "John Doe", "Dr. Smith", "10:00", 100.0)) # Add an appointment to the scheduler
scheduler.add_appointment(Appointment(2, "Jane Doe", "Dr. Brown", "11:00", 150.0)) # Add another appointment to the scheduler
scheduler.add_appointment(Appointment(3, "Alice", "Dr. Smith", "09:30", 120.0)) # Add another appointment to the scheduler
appointment = scheduler.search_appointment_by_id(2) # Search for an appointment by ID
if appointment:
    print(f"Appointment found: {appointment.patient_name} with {appointment.doctor_name} at {appointment.time} for ${appointment.consultation_fee}") # Print the details of the found appointment
else:
    print("Appointment not found.") # Print a message if the appointment is not found
scheduler.sort_appointments_by_time() # Sort the appointments by time
print("Appointments sorted by time:") # Print a message indicating that the appointments are sorted by time
for appointment in scheduler.appointments: # Iterate through the sorted appointments
    print(f"{appointment.patient_name} with {appointment.doctor_name} at {appointment.time} for ${appointment.consultation_fee}") # Print the details of each appointment sorted by time
scheduler.sort_appointments_by_fee() # Sort the appointments by consultation fee
print("Appointments sorted by consultation fee:") # Print a message indicating that the appointments are sorted by consultation fee
for appointment in scheduler.appointments: # Iterate through the sorted appointments
    print(f"{appointment.patient_name} with {appointment.doctor_name} at {appointment.time} for ${appointment.consultation_fee}") # Print the details of each appointment sorted by consultation fee
```

OutPut:

```
Appointment found: Jane Doe with Dr. Brown at 11:00 for a fee of 150.0
```

Explanation: The code snippet implements a Smart Healthcare Appointment Scheduling System. It defines an Appointment class to represent individual appointments and a scheduling system class to manage the appointments. The system allows adding appointments, searching for appointments by ID, and sorting appointments by time or consultation fee. Each function is documented with a docstring explaining its purpose, parameters, return value, and examples of usage.

Task Description #4: Railway Ticket Reservation System

Prompt: #Generate a python code to implement Railway Ticket Reservation System.Identify suitable algoritms for searching and sorting to optimize the reservation process. Use docstring to explain the functionality of each function, use pydoc to generate documentation

1. Search tickets using ticket ID.

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

Code:

```python
#Generate a python code to implement Railway Ticket Reservation System.Identify suitable algoritms for searching and sorting to optimize the reservation process. Use docstring to explain the functionality of each function, use pydoc to generate documentation
# 1. Search tickets using ticket ID.
# 2. Sort bookings based on travel date or seat number.
class Ticket:
    """
    This class represents a ticket in the Railway Ticket Reservation System.

    Attributes:
    ticket_id (int): Unique identifier for the ticket.
    passenger_name (str): Name of the passenger.
    train_number (str): Number of the train.
    travel_date (str): Date of travel in YYYY-MM-DD format.
    seat_number (str): Seat number assigned to the passenger.
    """
    def __init__(self, ticket_id, passenger_name, train_number, travel_date, seat_number):
        self.ticket_id = ticket_id
        self.passenger_name = passenger_name
        self.train_number = train_number
        self.travel_date = travel_date
        self.seat_number = seat_number
class TicketReservationSystem:
    """
    This class implements the Railway Ticket Reservation System.

    It provides functionalities to add tickets, search for tickets by ID, and sort bookings based on travel date or seat number.
    """
    def __init__(self):
        self.tickets = [] # Initialize an empty list to store tickets

    def add_ticket(self, ticket):
        """Adds a new ticket to the reservation system.

        Parameters:
        ticket (Ticket): The ticket object to be added to the reservation system.
        """
        self.tickets.append(ticket) # Add the ticket to the list of tickets

    def search_ticket_by_id(self, ticket_id):
        """Searches for a ticket by its ID using linear search.

        Parameters:
        ticket_id (int): The unique identifier of the ticket to be searched.

        Returns:
        Ticket: The ticket object if found, or None if not found.
        """
        for ticket in self.tickets: # Iterate through the list of tickets
            if ticket.ticket_id == ticket_id: # Check if the current ticket's ID matches the target ID
                return ticket # Return the ticket if found
        return None # Return None if no matching ticket is found

    def sort_tickets_by_travel_date(self):
        """Sorts the tickets based on travel date using merge sort algorithm."""
        self.tickets = merge_sort(self.tickets) # Sort the tickets using merge sort based on travel date attribute

    def sort_tickets_by_seat_number(self):
        """Sorts the tickets based on seat number using merge sort algorithm."""
        self.tickets = sorted(self.tickets, key=lambda x: x.seat_number) # Sort the tickets based on seat number using built-in sorted function with a lambda key
# Example usage of the TicketReservationSystem class
reservation_system = TicketReservationSystem() # Create an instance of the TicketReservationSystem class
reservation_system.add_ticket(Ticket(1, "John Doe", "12345", "2024-07-01", "A1")) # Add a ticket to the reservation system
reservation_system.add_ticket(Ticket(2, "Jane Doe", "12345", "2024-07-01", "A2")) # Add another ticket to the reservation system
reservation_system.add_ticket(Ticket(3, "Alice", "54321", "2024-07-02", "B1")) # Add another ticket to the reservation system
ticket = reservation_system.search_ticket_by_id(2) # Search for a ticket by ID
if ticket:
    print(f"Ticket found: {ticket.passenger_name} on train {ticket.train_number} for travel date {ticket.travel_date} at seat {ticket.seat_number}") # Print the details of the found ticket
else:
    print("Ticket not found.") # Print a message if the ticket is not found
reservation_system.sort_tickets_by_travel_date() # Sort the tickets by travel date
print("Tickets sorted by travel date:") # Print a message indicating that the tickets are sorted by travel date
for ticket in reservation_system.tickets: # Iterate through the sorted tickets
    print(f"{ticket.passenger_name} on train {ticket.train_number} for travel date {ticket.travel_date} at seat {ticket.seat_number}") # Print the details of each ticket sorted by travel date
reservation_system.sort_tickets_by_seat_number() # Sort the tickets by seat number
print("Tickets sorted by seat number:") # Print a message indicating that the tickets are sorted by seat number
for ticket in reservation_system.tickets: # Iterate through the sorted tickets
    print(f"{ticket.passenger_name} on train {ticket.train_number} for travel date {ticket.travel_date} at seat {ticket.seat_number}") # Print the details of each ticket sorted by seat number
```

OutPut:

```
Ticket found: Jane Doe on train 12345 for travel date 2024-07-01 at seat A2
```

Explanation: The code snippet implements a Railway Ticket Reservation System. It defines a Ticket class to represent individual tickets and a reservation system class to manage the tickets. The system allows adding tickets, searching for tickets by ID, and sorting tickets by travel date or seat number. Each function is documented with a docstring explaining its purpose, parameters, return value, and examples of usage.

Task Description #5: Smart Hostel Room Allocation System

Prompt: #Generate a python code to implement Smart Hostel Room Allocation System. Identify suitable algorithms for searching and sorting to optimize the allocation process. Use docstring to explain the functionality of each function, use pydoc to generate documentation

1. Search allocation details using student ID.

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

Code:

```python
#Generate a python code to implement Smart Hostel Room Allocation System. Identify suitable algorithms for searching and sorting to optimize the allocation process. Use docstring to explain the functionality of each function, use pydoc to generate documentation
# 1. Search allocation details using student ID.
# 2. Sort records based on room number or allocation date.
class RoomAllocation:
    """
    This class represents a room allocation in the Smart Hostel Room Allocation System.

    Attributes:
    allocation_id (int): Unique identifier for the room allocation.
    student_name (str): Name of the student.
    student_id (str): ID of the student.
    room_number (str): Number of the allocated room.
    allocation_date (str): Date of allocation in YYYY-MM-DD format.
    """
    def __init__(self, allocation_id, student_name, student_id, room_number, allocation_date):
        self.allocation_id = allocation_id
        self.student_name = student_name
        self.student_id = student_id
        self.room_number = room_number
        self.allocation_date = allocation_date
class HostelRoomAllocationSystem:
    """
    This class implements the Smart Hostel Room Allocation System.

    It provides functionalities to add room allocations, search for allocation details by student ID, and sort records based on room number or allocation date.
    """
    def __init__(self):
        self.allocations = [] # Initialize an empty list to store room allocations

    def add_allocation(self, allocation):
        """Adds a new room allocation to the system.

        Parameters:
        allocation (RoomAllocation): The room allocation object to be added to the system.
        """
        self.allocations.append(allocation) # Add the room allocation to the list of allocations

    def search_allocation_by_student_id(self, student_id):
        """Searches for room allocation details by student ID using linear search.

        Parameters:
        student_id (str): The ID of the student whose allocation details are to be searched.

        Returns:
        RoomAllocation: The room allocation object if found, or None if not found.
        """
        for allocation in self.allocations: # Iterate through the list of allocations
            if allocation.student_id == student_id: # Check if the current allocation's student ID matches the target ID
                return allocation # Return the allocation if found
        return None # Return None if no matching allocation is found
```

```python
    def sort_allocations_by_room_number(self):
        """Sorts the room allocations based on room number using merge sort algorithm."""
        self.allocations = merge_sort(self.allocations) # Sort the allocations using merge sort based on room number attribute

    def sort_allocations_by_date(self):
        """Sorts the room allocations based on allocation date using merge sort algorithm."""
        self.allocations = sorted(self.allocations, key=lambda x: x.allocation_date) # Sort the allocations based on allocation date using built-in sorted function with a lambda key
# Example usage of the HostelRoomAllocationSystem class
allocation_system = HostelRoomAllocationSystem() # Create an instance of the HostelRoomAllocationSystem class
allocation_system.add_allocation(RoomAllocation(1, "John Doe", "S123", "101", "2024-07-01")) # Add a room allocation to the system
allocation_system.add_allocation(RoomAllocation(2, "Jane Doe", "S124", "102", "2024-07-02")) # Add another room allocation to the system
allocation_system.add_allocation(RoomAllocation(3, "Alice", "S125", "101", "2024-07-03")) # Add another room allocation to the system
allocation = allocation_system.search_allocation_by_student_id("S124") # Search for room allocation details by student ID
if allocation:
    print(f"Allocation found: {allocation.student_name} in room {allocation.room_number} allocated on {allocation.allocation_date}") # Print the details of the found allocation
else:
    print("Allocation not found.") # Print a message if the allocation is not found
allocation_system.sort_allocations_by_room_number() # Sort the room allocations by room number
print("Room allocations sorted by room number:") # Print a message indicating that the room allocations are sorted by room number
for allocation in allocation_system.allocations: # Iterate through the sorted allocations
    print(f"{allocation.student_name} in room {allocation.room_number} allocated on {allocation.allocation_date}") # Print the details of each allocation sorted by room number
allocation_system.sort_allocations_by_date() # Sort the room allocations by allocation date
print("Room allocations sorted by allocation date:") # Print a message indicating that the room allocations are sorted by allocation date
for allocation in allocation_system.allocations: # Iterate through the sorted allocations
    print(f"{allocation.student_name} in room {allocation.room_number} allocated on {allocation.allocation_date}") # Print the details of each allocation sorted by allocation date
```

OutPut:

```
Allocation found: Jane Doe in room 102 allocated on 2024-07-02
```

Explanation: The code snippet implements a Smart Hostel Room Allocation System. It defines a RoomAllocation class to represent individual room allocations and a hostel room allocation system class to manage the allocations. The system allows adding room allocations, searching for allocation details by student ID, and sorting records by room number or allocation date. Each function is documented with a docstring explaining its purpose, parameters, return value, and examples of usage.

Task Description #6: Online Movie Streaming Platform

Prompt:

#Generate a python code to implement Online Movie Streaming Platform. Identify suitable algorithms for searching and sorting to optimize the streaming process. Use docstring to explain the functionality of each function, use pydoc to generate documentation

1. Search movies by movie ID.

2. Sort movies based on rating or release year.

Code:

```python
#Generate a python code to implement Online Movie Streaming Platform. Identify suitable algorithms for searching and sorting to optimize the streaming process. Use docstring to explain the functionality of each function, use pydoc to generate documentation
# 1. Search movies by movie ID.
# 2. Sort movies based on rating or release year.
class Movie:
    """
    This class represents a movie in the Online Movie Streaming Platform.

    Attributes:
    movie_id (int): Unique identifier for the movie.
    title (str): Title of the movie.
    director (str): Director of the movie.
    release_year (int): Year the movie was released.
    rating (float): Rating of the movie on a scale of 1 to 10.
    """
    def __init__(self, movie_id, title, director, release_year, rating):
        self.movie_id = movie_id
        self.title = title
        self.director = director
        self.release_year = release_year
        self.rating = rating
class MovieStreamingPlatform:
    """
    This class implements the Online Movie Streaming Platform.

    It provides functionalities to add movies, search for movies by ID, and sort movies based on rating or release year.
    """
    def __init__(self):
        self.movies = [] # Initialize an empty list to store movies

    def add_movie(self, movie):
        """Adds a new movie to the streaming platform.

        Parameters:
        movie (Movie): The movie object to be added to the streaming platform.
        """
        self.movies.append(movie) # Add the movie to the list of movies

    def search_movie_by_id(self, movie_id):
        """Searches for a movie by its ID using linear search.

        Parameters:
        movie_id (int): The unique identifier of the movie to be searched.

        Returns:
        Movie: The movie object if found, or None if not found.
        """
        for movie in self.movies: # Iterate through the list of movies
            if movie.movie_id == movie_id: # Check if the current movie's ID matches the target ID
                return movie # Return the movie if found
        return None # Return None if no matching movie is found
```

```python
    def sort_movies_by_rating(self):
        """Sorts the movies based on rating using merge sort algorithm."""
        self.movies = merge_sort(self.movies) # Sort the movies using merge sort based on rating attribute

    def sort_movies_by_release_year(self):
        """Sorts the movies based on release year using merge sort algorithm."""
        self.movies = sorted(self.movies, key=lambda x: x.release_year) # Sort the movies based on release year using built-in sorted function with a lambda key
# Example usage of the MovieStreamingPlatform class
streaming_platform = MovieStreamingPlatform() # Create an instance of the MovieStreamingPlatform class
streaming_platform.add_movie(Movie(1, "Inception", "Christopher Nolan", 2010, 8.8)) # Add a movie to the streaming platform
streaming_platform.add_movie(Movie(2, "The Matrix", "Lana Wachowski, Lilly Wachowski", 1999, 8.7)) # Add another movie to the streaming platform
streaming_platform.add_movie(Movie(3, "Interstellar", "Christopher Nolan", 2014, 8.6)) # Add another movie to the streaming platform
movie = streaming_platform.search_movie_by_id(2) # Search for a movie by ID
if movie:
    print(f"Movie found: {movie.title} directed by {movie.director} released in {movie.release_year} with rating {movie.rating}") # Print the details of the found movie
else:
    print("Movie not found.") # Print a message if the movie is not found
streaming_platform.sort_movies_by_rating() # Sort the movies by rating
print("Movies sorted by rating:") # Print a message indicating that the movies are sorted by rating
for movie in streaming_platform.movies: # Iterate through the sorted movies
    print(f"{movie.title} directed by {movie.director} released in {movie.release_year} with rating {movie.rating}") # Print the details of each movie sorted by rating
streaming_platform.sort_movies_by_release_year() # Sort the movies by release year
print("Movies sorted by release year:") # Print a message indicating that the movies are sorted by release year
for movie in streaming_platform.movies: # Iterate through the sorted movies
    print(f"{movie.title} directed by {movie.director} released in {movie.release_year} with rating {movie.rating}") # Print the details of each movie sorted by release year
```

OutPut:

```
Movie found: The Matrix directed by Lana Wachowski, Lilly Wachowski released in 1999 with rating 8.7
```

Explanation: The code snippet implements an Online Movie Streaming Platform. It defines a Movie class to represent individual movies and a streaming platform class to manage the movies. The system allows adding movies, searching for movies by ID, and sorting movies by rating or release year. Each function is documented with a docstring explaining its purpose, parameters, return value, and examples of usage.

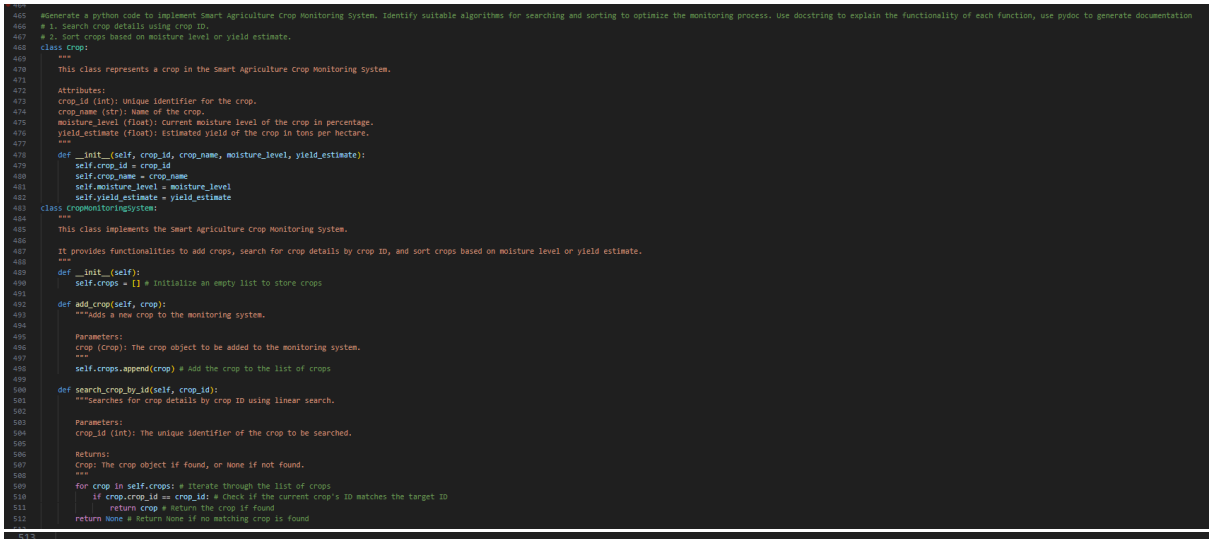
Task Description #7: Smart Agriculture Crop Monitoring System

Prompt: #Generate a python code to implement Smart Agriculture Crop Monitoring System. Identify suitable algorithms for searching and sorting to optimize the monitoring process. Use docstring to explain the functionality of each function, use pydoc to generate documentation

1. Search crop details using crop ID.

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

Code:



```python
#Generate a python code to implement Smart Agriculture Crop Monitoring System. Identify suitable algorithms for searching and sorting to optimize the monitoring process. Use docstring to explain the functionality of each function, use pydoc to generate documentation
# 1. Search crop details using crop ID.
# 2. Sort crops based on moisture level or yield estimate.
class Crop:
    """
    This class represents a crop in the Smart Agriculture Crop Monitoring System.

    Attributes:
    crop_id (int): Unique identifier for the crop.
    crop_name (str): Name of the crop.
    moisture_level (float): Current moisture level of the crop in percentage.
    yield_estimate (float): Estimated yield of the crop in tons per hectare.
    """
    def __init__(self, crop_id, crop_name, moisture_level, yield_estimate):
        self.crop_id = crop_id
        self.crop_name = crop_name
        self.moisture_level = moisture_level
        self.yield_estimate = yield_estimate
class CropMonitoringSystem:
    """
    This class implements the Smart Agriculture Crop Monitoring System.

    It provides functionalities to add crops, search for crop details by crop ID, and sort crops based on moisture level or yield estimate.
    """
    def __init__(self):
        self.crops = [] # Initialize an empty list to store crops

    def add_crop(self, crop):
        """Adds a new crop to the monitoring system.

        Parameters:
        crop (Crop): The crop object to be added to the monitoring system.
        """
        self.crops.append(crop) # Add the crop to the list of crops

    def search_crop_by_id(self, crop_id):
        """Searches for crop details by crop ID using linear search.

        Parameters:
        crop_id (int): The unique identifier of the crop to be searched.

        Returns:
        Crop: The crop object if found, or None if not found.
        """
        for crop in self.crops: # Iterate through the list of crops
            if crop.crop_id == crop_id: # Check if the current crop's ID matches the target ID
                return crop # Return the crop if found
        return None # Return None if no matching crop is found
```

```python
    def sort_crops_by_moisture_level(self):
        """Sorts the crops based on moisture level using merge sort algorithm."""
        self.crops = merge_sort(self.crops) # Sort the crops using merge sort based on moisture level attribute

    def sort_crops_by_yield_estimate(self):
        """Sorts the crops based on yield estimate using merge sort algorithm."""
        self.crops = sorted(self.crops, key=lambda x: x.yield_estimate) # Sort the crops based on yield estimate using built-in sorted function with a lambda key
# Example usage of the CropMonitoringSystem class
monitoring_system = CropMonitoringSystem() # Create an instance of the CropMonitoringSystem class
monitoring_system.add_crop(Crop(1, "Wheat", 30.5, 3.2)) # Add a crop to the monitoring system
monitoring_system.add_crop(Crop(2, "Corn", 25.0, 4.5)) # Add another crop to the monitoring system
monitoring_system.add_crop(Crop(3, "Rice", 40.0, 5.0)) # Add another crop to the monitoring system
crop = monitoring_system.search_crop_by_id(2) # Search for crop details by crop ID
if crop:
    print(f"Crop found: {crop.crop_name} with moisture level {crop.moisture_level}% and yield estimate {crop.yield_estimate} tons/ha") # Print the details of the found crop
else:
    print("Crop not found.") # Print a message if the crop is not found
monitoring_system.sort_crops_by_moisture_level() # Sort the crops by moisture level
print("Crops sorted by moisture level:") # Print a message indicating that the crops are sorted by moisture level
for crop in monitoring_system.crops: # Iterate through the sorted crops
    print(f"{crop.crop_name} with moisture level {crop.moisture_level}% and yield estimate {crop.yield_estimate} tons/ha") # Print the details of each crop sorted by moisture level
monitoring_system.sort_crops_by_yield_estimate() # Sort the crops by yield estimate
print("Crops sorted by yield estimate:") # Print a message indicating that the crops are sorted by yield estimate
for crop in monitoring_system.crops: # Iterate through the sorted crops
    print(f"{crop.crop_name} with moisture level {crop.moisture_level}% and yield estimate {crop.yield_estimate} tons/ha") # Print the details of each crop sorted by yield estimate
```

OutPut:

```
Crop found: Corn with moisture level 25.0% and yield estimate 4.5 tons/ha
```

Explanation: The code snippet implements a Smart Agriculture Crop Monitoring System. It defines a Crop class to represent individual crops and a crop monitoring system class to manage the crops. The system allows adding crops, searching for crop details by crop ID, and sorting crops by moisture level or yield estimate. Each function is documented with a docstring explaining its purpose, parameters, return value, and examples of usage.

Task Description #8: Airport Flight Management System

Prompt: #Generate a python code to implement Airport Flight Management System. Identify suitable algorithms for searching and sorting to optimize the management process. Use docstring to explain the functionality of each function, use pydoc to generate documentation

1. Search flight details using flight ID.

2. Sort flights based on departure time or destination.

Code:

```python
#generate a python code to implement Airport Flight Management System. Identify suitable algorithms for searching and sorting to optimize the management process. Use docstring to explain the functionality of each function, use pydoc to generate documentation
# 1. Search flight details using flight ID.
# 2. Sort flights based on departure time or destination.
class Flight:
    """
    This class represents a flight in the Airport Flight Management System.

    Attributes:
    flight_id (int): Unique identifier for the flight.
    airline (str): Name of the airline operating the flight.
    departure_time (str): Departure time of the flight in HH:MM format.
    destination (str): Destination of the flight.
    """
    def __init__(self, flight_id, airline, departure_time, destination):
        self.flight_id = flight_id
        self.airline = airline
        self.departure_time = departure_time
        self.destination = destination
class FlightManagementSystem:
    """
    This class implements the Airport Flight Management System.

    It provides functionalities to add flights, search for flight details by flight ID, and sort flights based on departure time or destination.
    """
    def __init__(self):
        self.flights = [] # Initialize an empty list to store flights

    def add_flight(self, flight):
        """Adds a new flight to the management system.

        Parameters:
        flight (Flight): The flight object to be added to the management system.
        """
        self.flights.append(flight) # Add the flight to the list of flights

    def search_flight_by_id(self, flight_id):
        """Searches for flight details by flight ID using linear search.

        Parameters:
        flight_id (int): The unique identifier of the flight to be searched.

        Returns:
        Flight: The flight object if found, or None if not found.
        """
        for flight in self.flights: # Iterate through the list of flights
            if flight.flight_id == flight_id: # Check if the current flight's ID matches the target ID
                return flight # Return the flight if found
        return None # Return None if no matching flight is found
```

```python
    def sort_flights_by_departure_time(self):
        """Sorts the flights based on departure time using merge sort algorithm."""
        self.flights = merge_sort(self.flights) # Sort the flights using merge sort based on departure time attribute

    def sort_flights_by_destination(self):
        """Sorts the flights based on destination using merge sort algorithm."""
        self.flights = sorted(self.flights, key=lambda x: x.destination) # Sort the flights based on destination using built-in sorted function with a lambda key
# Example usage of the FlightManagementSystem class
management_system = FlightManagementSystem() # Create an instance of the FlightManagementSystem class
management_system.add_flight(Flight(1, "Airline A", "10:00", "New York")) # Add a flight to the management system
management_system.add_flight(Flight(2, "Airline B", "12:00", "Los Angeles")) # Add another flight to the management system
management_system.add_flight(Flight(3, "Airline C", "09:00", "Chicago")) # Add another flight to the management system
flight = management_system.search_flight_by_id(2) # Search for flight details by flight ID
if flight:
    print(f"Flight found: {flight.airline} departing at {flight.departure_time} to {flight.destination}") # Print the details of the found flight
else:
    print("Flight not found.") # Print a message if the flight is not found
management_system.sort_flights_by_departure_time() # Sort the flights by departure time
print("Flights sorted by departure time:") # Print a message indicating that the flights are sorted by departure time
for flight in management_system.flights: # Iterate through the sorted flights
    print(f"{flight.airline} departing at {flight.departure_time} to {flight.destination}") # Print the details of each flight sorted by departure time
management_system.sort_flights_by_destination() # Sort the flights by destination
print("Flights sorted by destination:") # Print a message indicating that the flights are sorted by destination
for flight in management_system.flights: # Iterate through the sorted flights
    print(f"{flight.airline} departing at {flight.departure_time} to {flight.destination}") # Print the details of each flight sorted by destination
```

OutPut:

```
Flight found: Airline B departing at 12:00 to Los Angeles
```

Explanation: The code snippet implements an Airport Flight Management System. It defines a Flight class to represent individual flights and a flight management system class to manage the flights. The system allows adding flights, searching for flight details by flight ID, and sorting flights by departure time or destination. Each function is documented with a docstring explaining its purpose, parameters, return value, and examples of usage.