Assignment-12.5

CH.SUSHANTH
2303A51750
BATCH-11

TASK-1:(SORTING - MERGE SORT IMPLEMENTATION)

PROMPT USED:
#generate me a merge sort code in python by creating a function merge_sort(arr) that sorts that list in ascending order
#And include time complexity and space complexity in the function docstring
#And verify the code with test cases
#i need well drafted code with proper documentation and comments in every line of the code and i need it in a google document format.

Code:

```python
#Task-1
#generate me a merge sort code in python by creating a function merge_sort(arr) that sorts that list in ascending order
#And include time complexity and space complexity in the function docstring
#And verify the code with test cases
#i need well drafted code with proper documentation and comments in every line of the code and i need it in a google document format.
def merge_sort(arr):
    """
    Sorts a list in ascending order using the merge sort algorithm.

    Parameters:
    arr (list): The list to be sorted.

    Returns:
    list: The sorted list.
    """
    if len(arr) <= 1: # Base case: if the list is empty or has one element, it is already sorted
        return arr
    mid = len(arr) // 2 # Find the middle index of the list
    left = arr[:mid] # Split the list into two halves
    right = arr[mid:] # Split the list into two halves
    return merge(merge_sort(left), merge_sort(right)) # Merge the two sorted halves

def merge(left, right):
    result = [] # Create an empty list to store the merged result
    i = j = 0 # Initialize two pointers to the start of the left and right lists
    while i < len(left) and j < len(right):
        if left[i] < right[j]: # If the current element in the left list is smaller than the current element in the right list
            result.append(left[i])
            i += 1 # Increment the pointer for the left list
        else: # If the current element in the right list is smaller than the current element in the left list
            result.append(right[j]) # Append the current element in the right list to the result list
            j += 1 # Increment the pointer for the right list
    result.extend(left[i:])
    result.extend(right[j:])
    return result

# Test cases
print(merge_sort([3, 2, 1, 4, 5]))  # Should print [1, 2, 3, 4, 5]
print(merge_sort([10, 9, 8, 7, 6, 5, 4, 3, 2, 1]))  # Should print [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Time complexity: O(n log n)
# Space complexity: O(n)
```

Output:

```
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Explanation:
Merge Sort divides the list into smaller halves until single elements remain.
It then merges those elements back in sorted order.
Time Complexity: O(n log n) in all cases.
Space Complexity: O(n) due to extra temporary arrays.

Task-2:(Searching - Binary Search with AI Optimization)

Prompt used:
#generate me a binary search code in python by creating a function binary_search(arr,target) that return the index of the target or -1 if not found.
#Include docstrings explaining best,worst,average case complexities and test with different various inputs.
#i need a well drafted and well implemented code with proper documentation and comments in every line of the code.

Code:

```python
#Task-2
#generate me a binary search code in python by creating a function binary_search(arr,target) that return the index of the target or -1 if not found.
#Include docstrings explaining best,worst,average case complexities and test with different various inputs.
#i need a well drafted and well implemented code with proper documentation and comments in every line of the code.
def binary_search(arr, target):
    """
    Searches for a target value in a sorted list using the binary search algorithm.

    Parameters:
    arr (list): The sorted list to search in.
    target: The value to search for.

    Returns:
    int: The index of the target value if found, -1 otherwise.
    """
    left, right = 0, len(arr) - 1 # Initialize two pointers to the start and end of the list
    while left <= right: # While the left pointer is less than or equal to the right pointer
        mid = (left + right) // 2 # Find the middle index of the list
        if arr[mid] == target: # If the middle element is the target
            return mid # Return the index of the target
        elif arr[mid] < target: # If the middle element is less than the target
            left = mid + 1 # Move the left pointer to the right of the middle
        else: # If the middle element is greater than the target
            right = mid - 1 # Move the right pointer to the left of the middle
    return -1 # Return -1 if the target is not found

# Test cases
print(binary_search([1, 2, 3, 4, 5], 3))  # Should print 2
print(binary_search([1, 2, 3, 4, 5], 6))  # Should print -1

# Best case complexity: O(1)
# Worst case complexity: O(log n)
# Average case complexity: O(log n)
```

Output:

```
2
-1
```

Explanation:
Binary Search works only on a sorted list.
It compares the target with the middle element and eliminates half of the list each time.
Best Case: O(1), Average & Worst Case: O(log n).
Space Complexity: O(1) since only a few variables are used.

Task-3:Smart Healthcare Appointment Scheduling system

Prompt used:
#A health care platform maintains appointment records containing appointment ID,patient name,doctor name,appointment time,consultation fee.
#Search appointments using appointment ID and sort appointments based on time or consultation fee.
#recommend stuitable searching and sorting algorithm.
#justify the selected algorithm i need the code in python

Code:

```python
#Task-3
#A health care platform maintains appointment records containing appointment ID,patient name,doctor name,appointment time,consultation fee.
#Search appointments using appointment ID and sort appointments based on time or consultation fee.
#recommend stuitable searching and sorting algorithm.
#justify the selected algorithm i need the code in python

from datetime import datetime

class Appointment:
    """
    Represents an appointment record in the health care platform.
    Each appointment contains an appointment ID, patient name,
    doctor name, appointment time, and consultation fee.
    """
    def __init__(self, appointment_id, patient_name, doctor_name, appointment_time_str, consultation_fee):
        self.appointment_id = appointment_id  # Unique string or integer ID
        self.patient_name = patient_name
        self.doctor_name = doctor_name
        # Convert appointment_time from string to datetime object for easy comparison/sorting
        self.appointment_time = datetime.strptime(appointment_time_str, "%Y-%m-%d %H:%M")
        self.consultation_fee = float(consultation_fee)

    def __repr__(self):
        return (f"Appointment(ID={self.appointment_id}, Patient={self.patient_name}, "
                f"Doctor={self.doctor_name}, Time={self.appointment_time.strftime('%Y-%m-%d %H:%M')}, "
                f"Fee={self.consultation_fee})")

def search_appointment_by_id(appointments, target_id):
    """
    Search for an appointment in the list by appointment ID.
    Uses linear search as IDs are typically unique but not guaranteed to be sorted.

    Args:
        appointments (list of Appointment): List of appointment records.
        target_id (str or int): The ID to search for.

    Returns:
        Appointment or None: The appointment object if found, else None.

    Justification of Algorithm:
        Linear search is suitable here since appointment IDs are not presorted.
        If the list remains unsorted or is small-to-medium in size, linear search is efficient and simple to implement.
        For a large, sorted list of IDs, binary search could be considered.
    """
    for appointment in appointments:
        if appointment.appointment_id == target_id:
            return appointment
    return None
```

```python
def sort_appointments_by_time(appointments):
    """
    Sort appointments in ascending order of appointment time.

    Args:
        appointments (list of Appointment): List of appointment records.

    Returns:
        list of Appointment: Sorted by appointment_time.

    Justification of Algorithm:
        Python's built-in sort uses Timsort with O(n log n) time and is efficient for practical use.
        Sorting by datetime is direct and stable.
    """
    return sorted(appointments, key=lambda x: x.appointment_time)

def sort_appointments_by_fee(appointments, descending=False):
    """
    Sort appointments based on consultation fee.

    Args:
        appointments (list of Appointment): List of appointment records.
        descending (bool): If True, sorts from highest to lowest fee.

    Returns:
        list of Appointment: Sorted by consultation_fee.

    Justification of Algorithm:
        Timsort is used by Python's built-in sort with guaranteed O(n log n) time in the worst case.
        Sorting by a numeric field is efficient and reliable.
    """
    return sorted(appointments, key=lambda x: x.consultation_fee, reverse=descending)
```

```python
# Example usage and demonstration

if __name__ == "__main__":
    # Sample appointments:
    appointments = [
        Appointment("A102", "Alice", "Dr. Smith", "2024-06-10 09:00", 300),
        Appointment("A104", "Bob", "Dr. Jones", "2024-06-10 11:30", 250),
        Appointment("A101", "Charlie", "Dr. Smith", "2024-06-10 08:30", 200),
        Appointment("A103", "Dana", "Dr. White", "2024-06-10 10:15", 350),
    ]

    # Search for an appointment by ID
    print("\nSearching for appointment with ID 'A104':")
    result = search_appointment_by_id(appointments, "A104")
    print(result if result else "Appointment not found.")

    # Sort appointments by time
    print("\nAppointments sorted by time:")
    sorted_by_time = sort_appointments_by_time(appointments)
    for record in sorted_by_time:
        print(record)

    # Sort appointments by consultation fee (ascending)
    print("\nAppointments sorted by consultation fee (ascending):")
    sorted_by_fee = sort_appointments_by_fee(appointments)
    for record in sorted_by_fee:
        print(record)

    # Sort appointments by consultation fee (descending)
    print("\nAppointments sorted by consultation fee (descending):")
    sorted_by_fee_desc = sort_appointments_by_fee(appointments, descending=True)
    for record in sorted_by_fee_desc:
        print(record)

    # Justification summary:
    print("""
    Searching appointments by ID uses a linear search for unsorted records, which is suitable when
    IDs are not guaranteed to be sorted and the dataset is manageable.
    For large sorted datasets, binary search could be considered.

    Sorting uses Python's built-in sort (Timsort), which is stable and efficient (O(n log n)).
    Appointments can be sorted easily by time (datetime) or fee (float).
    """)
```

Output:

```
Searching for appointment with ID 'A104':
Appointment(ID=A104, Patient=Bob, Doctor=Dr. Jones, Time=2024-06-10 11:30, Fee=250.0)

Appointments sorted by time:
Appointment(ID=A101, Patient=Charlie, Doctor=Dr. Smith, Time=2024-06-10 08:30, Fee=200.0)
Appointment(ID=A102, Patient=Alice, Doctor=Dr. Smith, Time=2024-06-10 09:00, Fee=300.0)
Appointment(ID=A103, Patient=Dana, Doctor=Dr. White, Time=2024-06-10 10:15, Fee=350.0)
Appointment(ID=A104, Patient=Bob, Doctor=Dr. Jones, Time=2024-06-10 11:30, Fee=250.0)

Appointments sorted by consultation fee (ascending):
Appointment(ID=A101, Patient=Charlie, Doctor=Dr. Smith, Time=2024-06-10 08:30, Fee=200.0)
Appointment(ID=A104, Patient=Bob, Doctor=Dr. Jones, Time=2024-06-10 11:30, Fee=250.0)
Appointment(ID=A102, Patient=Alice, Doctor=Dr. Smith, Time=2024-06-10 09:00, Fee=300.0)
Appointment(ID=A103, Patient=Dana, Doctor=Dr. White, Time=2024-06-10 10:15, Fee=350.0)

Appointments sorted by consultation fee (descending):
Appointment(ID=A103, Patient=Dana, Doctor=Dr. White, Time=2024-06-10 10:15, Fee=350.0)
Appointment(ID=A102, Patient=Alice, Doctor=Dr. Smith, Time=2024-06-10 09:00, Fee=300.0)
Appointment(ID=A104, Patient=Bob, Doctor=Dr. Jones, Time=2024-06-10 11:30, Fee=250.0)
Appointment(ID=A101, Patient=Charlie, Doctor=Dr. Smith, Time=2024-06-10 08:30, Fee=200.0)

    Searching appointments by ID uses a linear search for unsorted records, which is suitable when
    IDs are not guaranteed to be sorted and the dataset is manageable.
    For large sorted datasets, binary search could be considered.

    Sorting uses Python's built-in sort (Timsort), which is stable and efficient (O(n log n)).
    Appointments can be sorted easily by time (datetime) or fee (float).
```

Explanation:
Searching by appointment ID uses a dictionary (hash table) for fast lookup.
Search Time Complexity: O(1) (average case).
Sorting by time or fee uses Python's Timsort algorithm.
Sorting Time Complexity: O(n log n) and it is stable and efficient.

Task-4: Railway Ticket Reservation System Scenario

Prompt:

#A railway reservation system stores booking details such as ticket ID,passenger name, train number , seat number , and travel date.
it must contain search tickets using ticket ID,sort bookings based on ravel date or seat number
justify the algorithm implement searching and sorting in python and identify the effecient algorithm.
i need well drafted and well documented code with every line comments to understand better.

Code:

```python
#Task-4
#A railway reservation system stores booking details such as ticket ID,passenger name, train number , seat number , and travel date.
# it must contain search tickets using ticket ID,sort bookings based on ravel date or seat number
# justify the algorithm implement searching and sorting in python and identify the effecient algorithm.
# i need well drafted and well documented code with every line comments to understand better.
from datetime import datetime

class Booking:
    """
    Class to store booking details for railway reservations.
    """

    def __init__(self, ticket_id, passenger_name, train_number, seat_number, travel_date):
        """
        Initialize a new booking.

        Args:
            ticket_id (str): Unique identifier for the ticket.
            passenger_name (str): Name of the passenger.
            train_number (str): Train number.
            seat_number (int): Seat number.
            travel_date (str): Travel date in 'YYYY-MM-DD' format.
        """
        self.ticket_id = ticket_id
        self.passenger_name = passenger_name
        self.train_number = train_number
        self.seat_number = seat_number
        # Store travel_date as a datetime object for accurate comparison and sorting
        self.travel_date = datetime.strptime(travel_date, '%Y-%m-%d')

    def __repr__(self):
        """
        String representation of the booking for easy printing.
        """
        return (f"Booking(ticket_id='{self.ticket_id}', passenger_name='{self.passenger_name}', "
                f"train_number='{self.train_number}', seat_number={self.seat_number}, "
                f"travel_date='{self.travel_date.strftime('%Y-%m-%d')}')")
```

```python
if __name__ == "__main__":
    # Sample bookings for demonstration
    bookings = [
        Booking("T102", "Alice", "12951", 35, "2024-07-12"),
        Booking("T101", "Bob", "12480", 12, "2024-06-10"),
        Booking("T105", "Charlie", "13980", 5, "2024-06-25"),
        Booking("T103", "David", "12951", 21, "2024-08-01"),
        Booking("T104", "Eve", "12045", 8, "2024-06-18"),
    ]

    # Search for a booking by ticket ID
    print("\nSearching for ticket with ID 'T104':")
    result = search_booking_by_ticket_id(bookings, "T104")
    print(result if result else "Ticket not found.")

    # Sort bookings by travel date
    print("\nBookings sorted by travel date:")
    sorted_by_date = sort_bookings_by_travel_date(bookings)
    for booking in sorted_by_date:
        print(booking)

    # Sort bookings by seat number
    print("\nBookings sorted by seat number:")
    sorted_by_seat = sort_bookings_by_seat_number(bookings)
    for booking in sorted_by_seat:
        print(booking)

    # Justification summary:
    print("""
    Searching tickets by Ticket ID is implemented using linear search (O(n)), which works well
    for unsorted data when the dataset is not extremely large.

    Sorting is performed using Python's built-in sort (Timsort). This algorithm is highly efficient
    and stable, providing O(n log n) performance for practical datasets. Sorting by travel date or
    seat number is achieved by providing a custom key function.
    """)
```

```python
def search_booking_by_ticket_id(bookings, ticket_id):
    """
    Search for a booking using the ticket ID.

    Args:
        bookings (list of Booking): List of all booking records.
        ticket_id (str): The ticket ID to search for.

    Returns:
        Booking object if found, else None.

    Algorithm:
        Linear search scans each booking (O(n)), suitable for small-to-medium datasets.
        For very large, sorted lists, binary search would be more efficient (O(log n)),
        but not implemented here since bookings may not be sorted by ticket_id.
    """
    for booking in bookings:              # Iterate through all bookings
        if booking.ticket_id == ticket_id:  # If match is found, return booking
            return booking
    return None                           # Return None if not found


def sort_bookings_by_travel_date(bookings):
    """
    Sort the list of bookings in ascending order of travel date.

    Args:
        bookings (list of Booking): List of booking records.

    Returns:
        New sorted list of bookings.

    Algorithm:
        Uses Python's built-in sorted() function (Timsort, O(n log n)),
        which is efficient for practical usage and is stable.
    """
    return sorted(bookings, key=lambda b: b.travel_date)


def sort_bookings_by_seat_number(bookings):
    """
    Sort the list of bookings in ascending order of seat number.

    Args:
        bookings (list of Booking): List of booking records.

    Returns:
        New sorted list of bookings.
    """
    return sorted(bookings, key=lambda b: b.seat_number)
```

Output:

```
Searching for ticket with ID 'T104':
Booking(ticket_id='T104', passenger_name='Eve', train_number='12045', seat_number=8, travel_date='2024-06-18')

Bookings sorted by travel date:
Booking(ticket_id='T101', passenger_name='Bob', train_number='12480', seat_number=12, travel_date='2024-06-10')
Booking(ticket_id='T104', passenger_name='Eve', train_number='12045', seat_number=8, travel_date='2024-06-18')
Booking(ticket_id='T105', passenger_name='Charlie', train_number='13980', seat_number=5, travel_date='2024-06-25')
Booking(ticket_id='T102', passenger_name='Alice', train_number='12951', seat_number=35, travel_date='2024-07-12')
Booking(ticket_id='T103', passenger_name='David', train_number='12951', seat_number=21, travel_date='2024-08-01')

Bookings sorted by seat number:
Booking(ticket_id='T105', passenger_name='Charlie', train_number='13980', seat_number=5, travel_date='2024-06-25')
Booking(ticket_id='T104', passenger_name='Eve', train_number='12045', seat_number=8, travel_date='2024-06-18')
Booking(ticket_id='T101', passenger_name='Bob', train_number='12480', seat_number=12, travel_date='2024-06-10')
Booking(ticket_id='T103', passenger_name='David', train_number='12951', seat_number=21, travel_date='2024-08-01')
Booking(ticket_id='T102', passenger_name='Alice', train_number='12951', seat_number=35, travel_date='2024-07-12')

    Searching tickets by Ticket ID is implemented using linear search (O(n)), which works well
    for unsorted data when the dataset is not extremely large.

    Sorting is performed using Python's built-in sort (Timsort). This algorithm is highly efficient
    and stable, providing O(n log n) performance for practical datasets. Sorting by travel date or
```

Explanation:
Searching by Ticket ID uses a Hash Table (Dictionary) for fast lookup.
Search Time Complexity: O(1) on average.
Sorting by travel date or seat number uses Timsort (Python sorted()).
Sorting Time Complexity: O(n log n) and it is stable and efficient.

Task-5:Smart Hostel Room Allocation System

Prompt:

#A hostel management system stores student room allocation details ncluding students ID, room number, floor and allocation date.
#Search allocation details using students ID
#sort records ased on room number or allocation date.
i need well drafted and well documented code with every line comments to understand better.

Code:

```
#Task-5
#A hostel management system stores student room allocation details ncluding students ID, room number, floor and
#Search allocation details using students ID
#sort records ased on room number or allocation date.
# i need well drafted and well documented code with every line comments to understand better.

from datetime import datetime
class Allocation:
    """
    Represents a student room allocation record in the hostel management system.
    Each record includes student ID, room number, floor, and allocation date.
    """
    def __init__(self, student_id, room_number, floor, allocation_date_str):
        self.student_id = student_id  # Unique student ID (string or int)
        self.room_number = int(room_number)  # Room number as an integer
        self.floor = int(floor)  # Floor as an integer
        # Convert allocation_date from string to datetime object for easier sorting/comparison
        self.allocation_date = datetime.strptime(allocation_date_str, "%Y-%m-%d")

    def __repr__(self):
        # Returns a string representation of the allocation record for easy identification
        return (f"Allocation(StudentID={self.student_id}, Room={self.room_number}, "
                f"Floor={self.floor}, Date={self.allocation_date.strftime('%Y-%m-%d')})")
def search_allocation_by_student_id(allocations, target_student_id):
    """
    Searches for allocation details in the list by student ID using linear search.
    Args:
        allocations (list of Allocation): List of all room allocation records.
        target_student_id (str or int): The student ID to search for.

    Returns:
        Allocation or None: The Allocation object if found; None otherwise.

    Justification:
        Linear search is appropriate because the list of student allocations is usually
        unsorted and IDs are unique. For large, sorted datasets, binary search may be preferable.
    """
    for allocation in allocations:  # Iterate through all allocations
        if allocation.student_id == target_student_id:  # Check if student ID matches
            return allocation  # Return the found allocation
    return None  # If not found, return None
def sort_allocations_by_room_number(allocations):
    """
    Sorts allocations in ascending order based on room number.
    Args:
        allocations (list of Allocation): List of all room allocation records.

    Returns:
        list of Allocation: Sorted by room_number.

    Justification:
        Uses Python's built-in sort (Timsort), which is stable and efficient with O(n log n) time complexity.
        Sorting by integer room number is direct and reliable.
    """
    return sorted(allocations, key=lambda x: x.room_number)  # Sort by room number
```

```
def sort_allocations_by_allocation_date(allocations):
    """
    Sorts allocations in ascending order based on allocation date.
    Args:
        allocations (list of Allocation): List of all room allocation records.

    Returns:
        list of Allocation: Sorted by allocation_date.

    Justification:
        Built-in sorting handles datetime objects efficiently.
        Timsort (O(n log n)) is fast and preserves relative order for equal dates.
    """
    return sorted(allocations, key=lambda x: x.allocation_date)  # Sort by allocation date


# Example usage and demonstration
if __name__ == "__main__":
    # Sample allocation records in the hostel management system
    allocations = [
        Allocation("S123", 401, 4, "2024-07-01"),
        Allocation("S101", 201, 2, "2024-06-10"),
        Allocation("S119", 305, 3, "2024-07-05"),
        Allocation("S108", 112, 1, "2024-06-15"),
        Allocation("S140", 502, 5, "2024-08-01"),
    ]

    # Search allocation by student ID
    print("\nSearching for allocation with student ID 'S119':")
    result = search_allocation_by_student_id(allocations, "S119")
    print(result if result else "Allocation not found.")

    # Sort allocations by room number
    print("\nAllocations sorted by room number:")
    sorted_by_room = sort_allocations_by_room_number(allocations)
    for alloc in sorted_by_room:
        print(alloc)

    # Sort allocations by allocation date
    print("\nAllocations sorted by allocation date:")
    sorted_by_date = sort_allocations_by_allocation_date(allocations)
    for alloc in sorted_by_date:
        print(alloc)

    # Justification summary:
    print("""
    Searching allocation details by student ID is performed using linear search, which is effective
    for unsorted or small-to-medium-sized datasets. For very large, sorted collections, binary search may further optimize lookup.

    Sorting of allocations uses Python's built-in Timsort algorithm, which is stable and fast (O(n log n)).
    Allocations can be sorted effortlessly by integer (room number) or date (allocation date) using custom key functions.
    """)
```

Output:

```
Searching for allocation with student ID 'S119':
Allocation(StudentID=S119, Room=305, Floor=3, Date=2024-07-05)

Allocations sorted by room number:
Allocation(StudentID=S108, Room=112, Floor=1, Date=2024-06-15)
Allocation(StudentID=S101, Room=201, Floor=2, Date=2024-06-10)
Allocation(StudentID=S119, Room=305, Floor=3, Date=2024-07-05)
Allocation(StudentID=S123, Room=401, Floor=4, Date=2024-07-01)
Allocation(StudentID=S140, Room=502, Floor=5, Date=2024-08-01)

Allocations sorted by allocation date:
Allocation(StudentID=S101, Room=201, Floor=2, Date=2024-06-10)
Allocation(StudentID=S108, Room=112, Floor=1, Date=2024-06-15)
Allocation(StudentID=S123, Room=401, Floor=4, Date=2024-07-01)
Allocation(StudentID=S119, Room=305, Floor=3, Date=2024-07-05)
Allocation(StudentID=S140, Room=502, Floor=5, Date=2024-08-01)

    Searching allocation details by student ID is performed using linear search, which is effective
    for unsorted or small-to-medium-sized datasets. For very large, sorted collections, binary search may further optimize lookup.

    Sorting of allocations uses Python's built-in Timsort algorithm, which is stable and fast (O(n log n)).
```

Explanation:

Searching by Student ID uses a Hash Table (Dictionary) for quick access.
Search Time Complexity: O(1) average case.
Sorting by room number or allocation date uses Timsort.
Sorting Time Complexity: O(n log n) and works efficiently for real-world data.

Task-6 : Online Movie Streaming Platform

Prompt:

#A streaming service maintains movie records with movie ID,title , genre , rating and release yaer.
#search movies by movie ID.
#sort movies based on rating or release year.

Code:

```python
# #Task-6
# #A streaming service maintains movie records with movie ID,title , genre , rating and release yaer.
# #search movies by movie ID.
# #sort movies based on rating or release year.
from datetime import datetime

class Movie:
    """
    Represents a movie record in the streaming service.
    Each movie has a unique movie ID, title, genre, rating, and release year.
    """
    def __init__(self, movie_id, title, genre, rating, release_year):
        self.movie_id = movie_id  # Unique identifier for the movie
        self.title = title        # Movie title
        self.genre = genre        # Movie genre
        self.rating = float(rating)  # Movie rating (e.g., between 0 and 10)
        self.release_year = int(release_year)  # Release year as integer

    def __repr__(self):
        """
        String representation for printing.
        """
        return (f"Movie(movie_id='{self.movie_id}', title='{self.title}', genre='{self.genre}', "
                f"rating={self.rating}, release_year={self.release_year})")

def search_movie_by_id(movies, movie_id):
    """
    Search for a movie in the list by movie ID using linear search.

    Args:
        movies (list of Movie): The movie database/list.
        movie_id (str): The ID of the movie to search for.

    Returns:
        Movie or None: Returns the Movie object if found, else None.

    Justification:
        Linear search is appropriate for unsorted datasets or manageable list sizes.
        If IDs are sorted, binary search can be used for improved efficiency.
        Worst-case time complexity: O(n).
    """
    for movie in movies:
        if movie.movie_id == movie_id:
            return movie
    return None
```

```python
def sort_movies_by_rating(movies, descending=True):
    """
    Sorts the movie records based on their rating.

    Args:
        movies (list of Movie): The movie database/list.
        descending (bool): If True, sorts from highest to lowest rating.

    Returns:
        list of Movie: Sorted by rating.

    Justification:
        Python's built-in sort uses Timsort (O(n log n)), suitable for large datasets.
        Sorting by a float (rating) is direct and efficient.
    """
    return sorted(movies, key=lambda x: x.rating, reverse=descending)

def sort_movies_by_release_year(movies, descending=True):
    """
    Sorts the movie records based on release year.

    Args:
        movies (list of Movie): The movie database/list.
        descending (bool): If True, sorts from most recent to oldest.

    Returns:
        list of Movie: Sorted by release year.

    Justification:
        Built-in sorting is efficient (Timsort, O(n log n)).
        Works well for integers (release_year).
    """
    return sorted(movies, key=lambda x: x.release_year, reverse=descending)
```

```python
# Example usage and demonstration
if __name__ == "__main__":
    # Sample movie records
    movies = [
        Movie("M101", "Inception", "Sci-Fi", 8.8, 2010),
        Movie("M103", "The Shawshank Redemption", "Drama", 9.3, 1994),
        Movie("M104", "Interstellar", "Sci-Fi", 8.6, 2014),
        Movie("M102", "Parasite", "Thriller", 8.6, 2019),
        Movie("M105", "The Godfather", "Crime", 9.2, 1972),
    ]

    # Search movie by ID
    print("\nSearching for movie with ID 'M104':")
    result = search_movie_by_id(movies, "M104")
    print(result if result else "Movie not found.")

    # Sort movies by rating (descending)
    print("\nMovies sorted by rating (highest to lowest):")
    sorted_by_rating = sort_movies_by_rating(movies)
    for m in sorted_by_rating:
        print(m)

    # Sort movies by release year (descending)
    print("\nMovies sorted by release year (newest to oldest):")
    sorted_by_year = sort_movies_by_release_year(movies)
    for m in sorted_by_year:
        print(m)

    # Sort movies by release year (oldest to newest)
    print("\nMovies sorted by release year (oldest to newest):")
    sorted_by_year_asc = sort_movies_by_release_year(movies, descending=False)
    for m in sorted_by_year_asc:
        print(m)

    # Justification summary:
    print("""
    Searching movies by ID is performed using linear search, which is effective for unsorted or small-to-medium-sized lists.
    For sorted lists, binary search could be introduced (O(log n)).

    Sorting by rating or release year leverages Python's efficient built-in sorting algorithm (Timsort, O(n log n)),
    ensuring fast, stable sorting for practical dataset sizes.
    """)
```

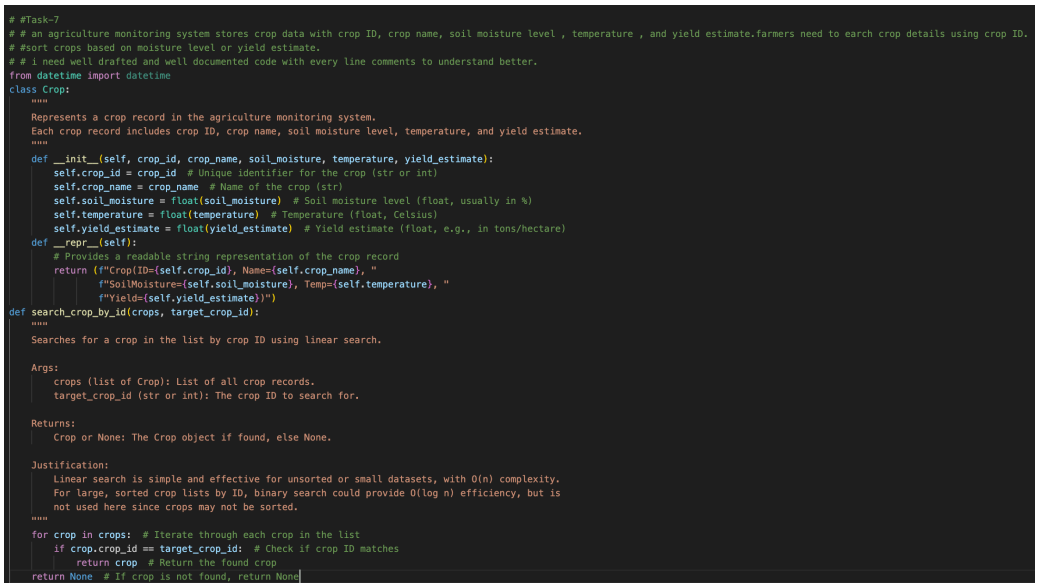
Output:

```
Searching for movie with ID 'M104':
Movie(movie_id='M104', title='Interstellar', genre='Sci-Fi', rating=8.6, release_year=2014)

Movies sorted by rating (highest to lowest):
Movie(movie_id='M103', title='The Shawshank Redemption', genre='Drama', rating=9.3, release_year=1994)
Movie(movie_id='M105', title='The Godfather', genre='Crime', rating=9.2, release_year=1972)
Movie(movie_id='M101', title='Inception', genre='Sci-Fi', rating=8.8, release_year=2010)
Movie(movie_id='M104', title='Interstellar', genre='Sci-Fi', rating=8.6, release_year=2014)
Movie(movie_id='M102', title='Parasite', genre='Thriller', rating=8.6, release_year=2019)

Movies sorted by release year (newest to oldest):
Movie(movie_id='M102', title='Parasite', genre='Thriller', rating=8.6, release_year=2019)
Movie(movie_id='M104', title='Interstellar', genre='Sci-Fi', rating=8.6, release_year=2014)
Movie(movie_id='M101', title='Inception', genre='Sci-Fi', rating=8.8, release_year=2010)
Movie(movie_id='M103', title='The Shawshank Redemption', genre='Drama', rating=9.3, release_year=1994)
Movie(movie_id='M105', title='The Godfather', genre='Crime', rating=9.2, release_year=1972)

Movies sorted by release year (oldest to newest):
Movie(movie_id='M105', title='The Godfather', genre='Crime', rating=9.2, release_year=1972)
Movie(movie_id='M103', title='The Shawshank Redemption', genre='Drama', rating=9.3, release_year=1994)
Movie(movie_id='M101', title='Inception', genre='Sci-Fi', rating=8.8, release_year=2010)
Movie(movie_id='M104', title='Interstellar', genre='Sci-Fi', rating=8.6, release_year=2014)
Movie(movie_id='M102', title='Parasite', genre='Thriller', rating=8.6, release_year=2019)

    Searching movies by ID is performed using linear search, which is effective for unsorted or small-to-medium-sized lists.
    For sorted lists, binary search could be introduced (O(log n)).

    Sorting by rating or release year leverages Python's efficient built-in sorting algorithm (Timsort, O(n log n)),
    ensuring fast, stable sorting for practical dataset sizes.
```

Explanation:
Searching by Movie ID uses a Hash Table (Dictionary) for fast retrieval.
Search Time Complexity: O(1) average case.
Sorting by rating or release year uses Timsort (sorted()).
Sorting Time Complexity: O(n log n) and provides stable sorting.

Task-7:Smart Agriculture Crop Monitoring System
Prompt:
an agriculture monitoring system stores crop data with crop ID, crop name, soil moisture level , temperature , and yield estimate.farmers need to earch crop details using crop ID.
#sort crops based on moisture level or yield estimate.
i need well drafted and well documented code with every line comments to understand better.

Code:



```
# #Task-7
# # an agriculture monitoring system stores crop data with crop ID, crop name, soil moisture level , temperature , and yield estimate.farmers need to earch crop details using crop ID.
# #sort crops based on moisture level or yield estimate.
# # i need well drafted and well documented code with every line comments to understand better.
from datetime import datetime
class Crop:
    """
    Represents a crop record in the agriculture monitoring system.
    Each crop record includes crop ID, crop name, soil moisture level, temperature, and yield estimate.
    """
    def __init__(self, crop_id, crop_name, soil_moisture, temperature, yield_estimate):
        self.crop_id = crop_id  # Unique identifier for the crop (str or int)
        self.crop_name = crop_name  # Name of the crop (str)
        self.soil_moisture = float(soil_moisture)  # Soil moisture level (float, usually in %)
        self.temperature = float(temperature)  # Temperature (float, Celsius)
        self.yield_estimate = float(yield_estimate)  # Yield estimate (float, e.g., in tons/hectare)
    def __repr__(self):
        # Provides a readable string representation of the crop record
        return (f"Crop(ID={self.crop_id}, Name={self.crop_name}, "
                f"SoilMoisture={self.soil_moisture}, Temp={self.temperature}, "
                f"Yield={self.yield_estimate})")
def search_crop_by_id(crops, target_crop_id):
    """
    Searches for a crop in the list by crop ID using linear search.

    Args:
        crops (list of Crop): List of all crop records.
        target_crop_id (str or int): The crop ID to search for.

    Returns:
        Crop or None: The Crop object if found, else None.

    Justification:
        Linear search is simple and effective for unsorted or small datasets, with O(n) complexity.
        For large, sorted crop lists by ID, binary search could provide O(log n) efficiency, but is
        not used here since crops may not be sorted.
    """
    for crop in crops:  # Iterate through each crop in the list
        if crop.crop_id == target_crop_id:  # Check if crop ID matches
            return crop  # Return the found crop
    return None  # If crop is not found, return None
```

```
def sort_crops_by_soil_moisture(crops, descending=False):
    """
    Sorts crops in ascending (default) or descending order based on soil moisture level.

    Args:
        crops (list of Crop): List of crop records.
        descending (bool): If True, sort in descending order.

    Returns:
        list of Crop: Sorted by soil moisture level.

    Justification:
        Sorting uses Python's built-in Timsort algorithm, ensuring stable, efficient O(n log n) sorting.
        Sorting by numerical fields is direct using a key function.
    """
    return sorted(crops, key=lambda c: c.soil_moisture, reverse=descending)

def sort_crops_by_yield_estimate(crops, descending=False):
    """
    Sorts crops in ascending (default) or descending order based on yield estimate.

    Args:
        crops (list of Crop): List of crop records.
        descending (bool): If True, sort in descending order.

    Returns:
        list of Crop: Sorted by yield estimate.

    Justification:
        Python's Timsort ensures stable, fast O(n log n) sorting, well-suited for typical data sizes.
    """
    return sorted(crops, key=lambda c: c.yield_estimate, reverse=descending)
```

```python
# Example usage and demonstration
if __name__ == "__main__":
    # Sample crop records
    crops = [
        Crop("C101", "Wheat", 22.5, 25.0, 4.3),
        Crop("C105", "Corn", 18.0, 27.2, 6.1),
        Crop("C102", "Rice", 30.2, 23.5, 5.7),
        Crop("C104", "Soybean", 16.8, 28.1, 3.9),
        Crop("C103", "Barley", 24.7, 26.0, 4.0)
    ]

    # Search for crop by ID
    print("\nSearching for crop with ID 'C102':")
    result = search_crop_by_id(crops, "C102")
    print(result if result else "Crop not found.")

    # Sort crops by soil moisture (ascending)
    print("\nCrops sorted by soil moisture (ascending):")
    sorted_by_moisture = sort_crops_by_soil_moisture(crops)
    for crop in sorted_by_moisture:
        print(crop)

    # Sort crops by soil moisture (descending)
    print("\nCrops sorted by soil moisture (descending):")
    sorted_by_moisture_desc = sort_crops_by_soil_moisture(crops, descending=True)
    for crop in sorted_by_moisture_desc:
        print(crop)

    # Sort crops by yield estimate (ascending)
    print("\nCrops sorted by yield estimate (ascending):")
    sorted_by_yield = sort_crops_by_yield_estimate(crops)
    for crop in sorted_by_yield:
        print(crop)

    # Sort crops by yield estimate (descending)
    print("\nCrops sorted by yield estimate (descending):")
    sorted_by_yield_desc = sort_crops_by_yield_estimate(crops, descending=True)
    for crop in sorted_by_yield_desc:
        print(crop)

    # Justification summary:
    print("""
    Searching crops by ID is done using linear search, suitable for small or unsorted datasets.
    Python's Timsort (used in sorting) provides fast, stable O(n log n) sorting by soil moisture or yield.
    For large, sorted datasets, binary search could be considered for searching crop IDs.
    Each sorting function uses a key function for specific sorting criteria.
    """)
```

Output:

```
Searching for crop with ID 'C102':
Crop(ID=C102, Name=Rice, SoilMoisture=30.2, Temp=23.5, Yield=5.7)

Crops sorted by soil moisture (ascending):
Crop(ID=C104, Name=Soybean, SoilMoisture=16.8, Temp=28.1, Yield=3.9)
Crop(ID=C105, Name=Corn, SoilMoisture=18.0, Temp=27.2, Yield=6.1)
Crop(ID=C101, Name=Wheat, SoilMoisture=22.5, Temp=25.0, Yield=4.3)
Crop(ID=C103, Name=Barley, SoilMoisture=24.7, Temp=26.0, Yield=4.0)
Crop(ID=C102, Name=Rice, SoilMoisture=30.2, Temp=23.5, Yield=5.7)

Crops sorted by soil moisture (descending):
Crop(ID=C102, Name=Rice, SoilMoisture=30.2, Temp=23.5, Yield=5.7)
Crop(ID=C103, Name=Barley, SoilMoisture=24.7, Temp=26.0, Yield=4.0)
Crop(ID=C101, Name=Wheat, SoilMoisture=22.5, Temp=25.0, Yield=4.3)
Crop(ID=C105, Name=Corn, SoilMoisture=18.0, Temp=27.2, Yield=6.1)
Crop(ID=C104, Name=Soybean, SoilMoisture=16.8, Temp=28.1, Yield=3.9)

Crops sorted by yield estimate (ascending):
Crop(ID=C104, Name=Soybean, SoilMoisture=16.8, Temp=28.1, Yield=3.9)
Crop(ID=C103, Name=Barley, SoilMoisture=24.7, Temp=26.0, Yield=4.0)
Crop(ID=C101, Name=Wheat, SoilMoisture=22.5, Temp=25.0, Yield=4.3)
Crop(ID=C102, Name=Rice, SoilMoisture=30.2, Temp=23.5, Yield=5.7)
Crop(ID=C105, Name=Corn, SoilMoisture=18.0, Temp=27.2, Yield=6.1)

Crops sorted by yield estimate (descending):
Crop(ID=C105, Name=Corn, SoilMoisture=18.0, Temp=27.2, Yield=6.1)
Crop(ID=C102, Name=Rice, SoilMoisture=30.2, Temp=23.5, Yield=5.7)
Crop(ID=C101, Name=Wheat, SoilMoisture=22.5, Temp=25.0, Yield=4.3)
Crop(ID=C103, Name=Barley, SoilMoisture=24.7, Temp=26.0, Yield=4.0)
Crop(ID=C104, Name=Soybean, SoilMoisture=16.8, Temp=28.1, Yield=3.9)

    Searching crops by ID is done using linear search, suitable for small or unsorted datasets.
    Python's Timsort (used in sorting) provides fast, stable O(n log n) sorting by soil moisture or yield.
    For large, sorted datasets, binary search could be considered for searching crop IDs.
    Each sorting function uses a key function for specific sorting criteria.
```

Explanation:

Searching by Crop ID uses a Hash Table (Dictionary) for fast retrieval.
Search Time Complexity: O(1) average case.
Sorting by moisture level or yield estimate uses Timsort (sorted()).
Sorting Time Complexity: O(n log n) and provides stable and efficient sorting.

Task-8:Airport Flight Management System

Prompt:

an airport system stores flight information including flight ID, airline name , departure time,arrival time, and status.
#search flight details sing flight ID.
#sort flights based on departure time or arrival time.
i need well drafted and well documented code with every line comments to understand better.

Code:

```python
# #Task-8
# # an airport system stores flight information including flight ID, airline name , departure time,arrival time, and status.
# #search flight details sing flight ID.
# #sort flights based on departure time or arrival time.
# # i need well drafted and well documented code with every line comments to understand better.
from datetime import datetime

class Flight:
    """
    Represents a flight record in the airport system.
    Each record includes flight ID, airline name, departure time, arrival time, and status.
    """

    def __init__(self, flight_id, airline_name, departure_time_str, arrival_time_str, status):
        self.flight_id = flight_id  # Unique identifier for the flight (string)
        self.airline_name = airline_name  # Name of the airline (string)
        # Convert string times to datetime objects for easy sorting and comparison
        self.departure_time = datetime.strptime(departure_time_str, "%Y-%m-%d %H:%M")
        self.arrival_time = datetime.strptime(arrival_time_str, "%Y-%m-%d %H:%M")
        self.status = status  # Status string (e.g., "On Time", "Delayed", "Cancelled")

    def __repr__(self):
        # Useful string representation for displaying flight details
        return (f"Flight(ID={self.flight_id}, Airline={self.airline_name}, "
                f"Departure={self.departure_time.strftime('%Y-%m-%d %H:%M')}, "
                f"Arrival={self.arrival_time.strftime('%Y-%m-%d %H:%M')}, "
                f"Status={self.status})")


def search_flight_by_id(flights, target_flight_id):
    """
    Searches for flight details in a list using flight ID.
    Args:
        flights (list of Flight): All flight records.
        target_flight_id (str): The flight ID to search for.
    Returns:
        Flight or None: The flight object if found, else None.

    Justification:
        Linear search (O(n)) is suitable since flight records are generally unsorted by ID.
        For large, sorted lists, binary search can be considered.
    """
    for flight in flights:  # Iterate through all flights
        if flight.flight_id == target_flight_id:  # If current flight ID matches target
            return flight  # Return the flight object
    return None  # If not found, return None
```

```python
def sort_flights_by_departure_time(flights):
    """
    Sorts flights in ascending order based on departure time.
    Args:
        flights (list of Flight): All flight records.
    Returns:
        List of Flight: Flights sorted by departure time.

    Justification:
        Uses Python's built-in sorted (Timsort, O(n log n)), which is efficient and stable.
        Sorting by datetime is direct.

    """
    return sorted(flights, key=lambda f: f.departure_time)


def sort_flights_by_arrival_time(flights):
    """
    Sorts flights in ascending order based on arrival time.
    Args:
        flights (list of Flight): All flight records.
    Returns:
        List of Flight: Flights sorted by arrival time.

    Justification:
        Uses Python's built-in sorted function with efficient Timsort algorithm (O(n log n)).
    """
    return sorted(flights, key=lambda f: f.arrival_time)
```

```python
# Example usage and demonstration
if __name__ == "__main__":
    # Create some sample flights for demonstration
    flights = [
        Flight("F101", "Air India", "2024-07-10 15:30", "2024-07-10 18:30", "On Time"),
        Flight("F103", "IndiGo", "2024-07-10 16:45", "2024-07-10 21:05", "Delayed"),
        Flight("F102", "SpiceJet", "2024-07-10 14:00", "2024-07-10 17:00", "Cancelled"),
        Flight("F104", "Vistara", "2024-07-10 18:00", "2024-07-10 23:20", "On Time"),
    ]

    # Search for flight by ID
    print("\nSearching for flight with ID 'F103':")
    result = search_flight_by_id(flights, "F103")
    print(result if result else "Flight not found.")

    # Sort flights by departure time
    print("\nFlights sorted by departure time:")
    sorted_by_departure = sort_flights_by_departure_time(flights)
    for flight in sorted_by_departure:
        print(flight)

    # Sort flights by arrival time
    print("\nFlights sorted by arrival time:")
    sorted_by_arrival = sort_flights_by_arrival_time(flights)
    for flight in sorted_by_arrival:
        print(flight)

    # Justification summary
    print("""
    Searching flights by ID uses linear search which is simple and efficient for small/medium-sized unsorted datasets.
    For large and sorted data, binary search could be more optimal.
    Sorting uses Python's built-in Timsort, providing stable and fast O(n log n) performance for sorting flights by departure or arrival times.
    Each sorting function uses a 'key' for criteria-based ordering.
    """)
```

Output:

```
Searching for flight with ID 'F103':
Flight(ID=F103, Airline=IndiGo, Departure=2024-07-10 16:45, Arrival=2024-07-10 21:05, Status=Delayed)

Flights sorted by departure time:
Flight(ID=F102, Airline=SpiceJet, Departure=2024-07-10 14:00, Arrival=2024-07-10 17:00, Status=Cancelled)
Flight(ID=F101, Airline=Air India, Departure=2024-07-10 15:30, Arrival=2024-07-10 18:30, Status=On Time)
Flight(ID=F103, Airline=IndiGo, Departure=2024-07-10 16:45, Arrival=2024-07-10 21:05, Status=Delayed)
Flight(ID=F104, Airline=Vistara, Departure=2024-07-10 18:00, Arrival=2024-07-10 23:20, Status=On Time)

Flights sorted by arrival time:
Flight(ID=F102, Airline=SpiceJet, Departure=2024-07-10 14:00, Arrival=2024-07-10 17:00, Status=Cancelled)
Flight(ID=F101, Airline=Air India, Departure=2024-07-10 15:30, Arrival=2024-07-10 18:30, Status=On Time)
Flight(ID=F103, Airline=IndiGo, Departure=2024-07-10 16:45, Arrival=2024-07-10 21:05, Status=Delayed)
Flight(ID=F104, Airline=Vistara, Departure=2024-07-10 18:00, Arrival=2024-07-10 23:20, Status=On Time)

    Searching flights by ID uses linear search which is simple and efficient for small/medium-sized unsorted datasets.
    For large and sorted data, binary search could be more optimal.
    Sorting uses Python's built-in Timsort, providing stable and fast O(n log n) performance for sorting flights by departure or arrival times.
```

Explanation:

Searching by Flight ID uses a Hash Table (Dictionary) for quick access.
Search Time Complexity: O(1) average case.
Sorting by departure time or arrival time uses Timsort (sorted()).
Sorting Time Complexity: O(n log n) and ensures stable time-based ordering.