SR UNIVERSITY

School of Computer Science and Artificial Intelligence

Lab Assignment # 12.5

Program : B. Tech (CSE)
Specialization : -
Course Title : AI Assisted Coding
Course Code : 23CS002PC304
Semester : II
Academic Session : 2025-2026
Name of Student : I.Sathwik Rajeshwara Chary
Enrollment No. : 2403A51L03
Batch No. : 51
Date : 20/02/26

Submission Starts here

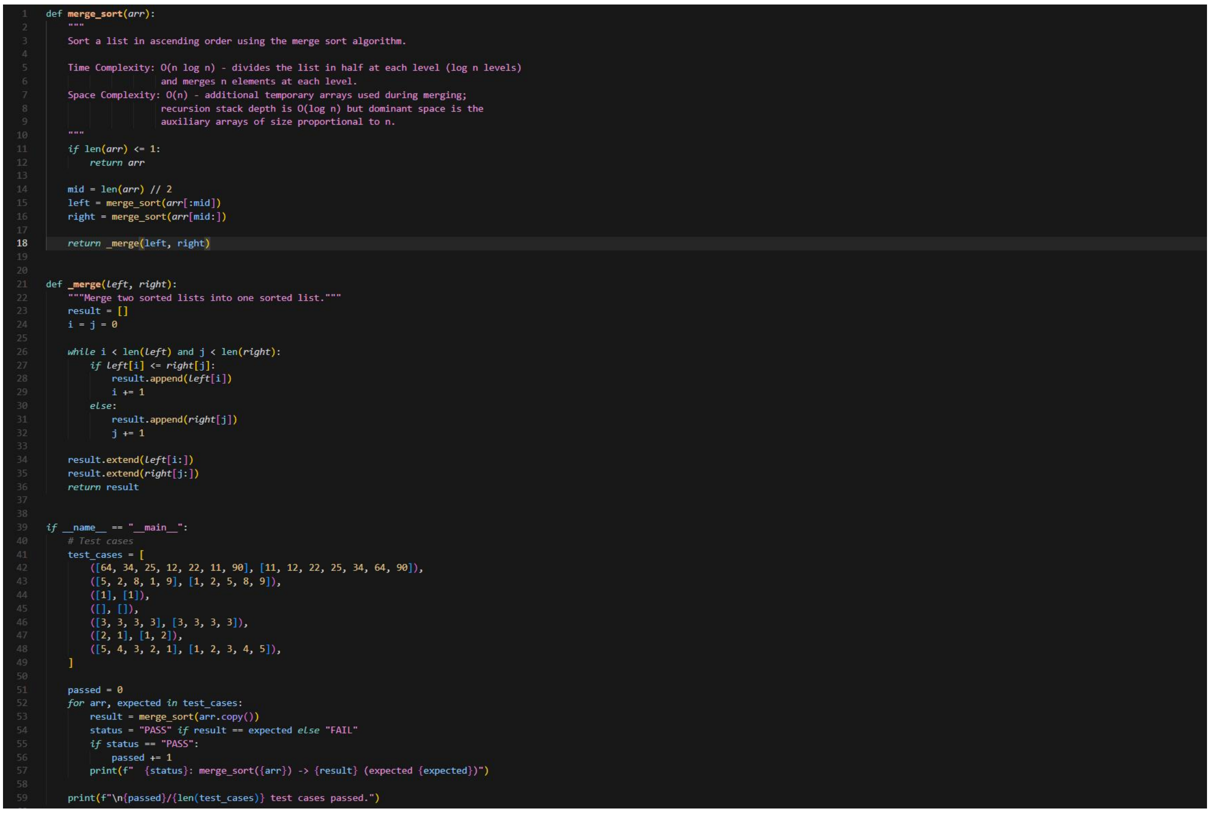
Screenshots:

Task Description #1 (Sorting – Merge Sort Implementation)

Prompt:

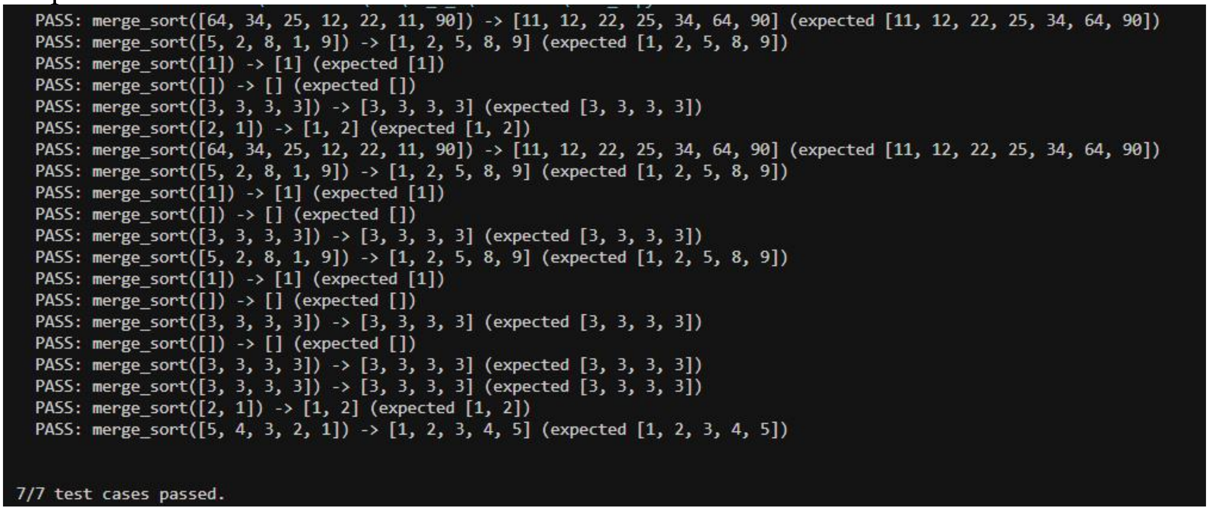
create a function merge_sort(arr) that sorts a list in ascending order. include time complexity and space complexity in the function docstring. Verify the generated code with test cases.

Code:



```python
def merge_sort(arr):
    """
    Sort a list in ascending order using the merge sort algorithm.

    Time Complexity: O(n log n) - divides the list in half at each level (log n levels)
                     and merges n elements at each level.
    Space Complexity: O(n) - additional temporary arrays used during merging;
                     recursion stack depth is O(log n) but dominant space is the
                     auxiliary arrays of size proportional to n.
    """
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])

    return _merge(left, right)


def _merge(left, right):
    """Merge two sorted lists into one sorted list."""
    result = []
    i = j = 0

    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

    result.extend(left[i:])
    result.extend(right[j:])
    return result


if __name__ == "__main__":
    # Test cases
    test_cases = [
        ([64, 34, 25, 12, 22, 11, 90], [11, 12, 22, 25, 34, 64, 90]),
        ([5, 2, 8, 1, 9], [1, 2, 5, 8, 9]),
        ([1], [1]),
        ([], []),
        ([3, 3, 3, 3], [3, 3, 3, 3]),
        ([2, 1], [1, 2]),
        ([5, 4, 3, 2, 1], [1, 2, 3, 4, 5]),
    ]

    passed = 0
    for arr, expected in test_cases:
        result = merge_sort(arr.copy())
        status = "PASS" if result == expected else "FAIL"
        if status == "PASS":
            passed += 1
        print(f"  {status}: merge_sort({arr}) -> {result} (expected {expected})")

    print(f"\n{passed}/{len(test_cases)} test cases passed.")
```

Output:



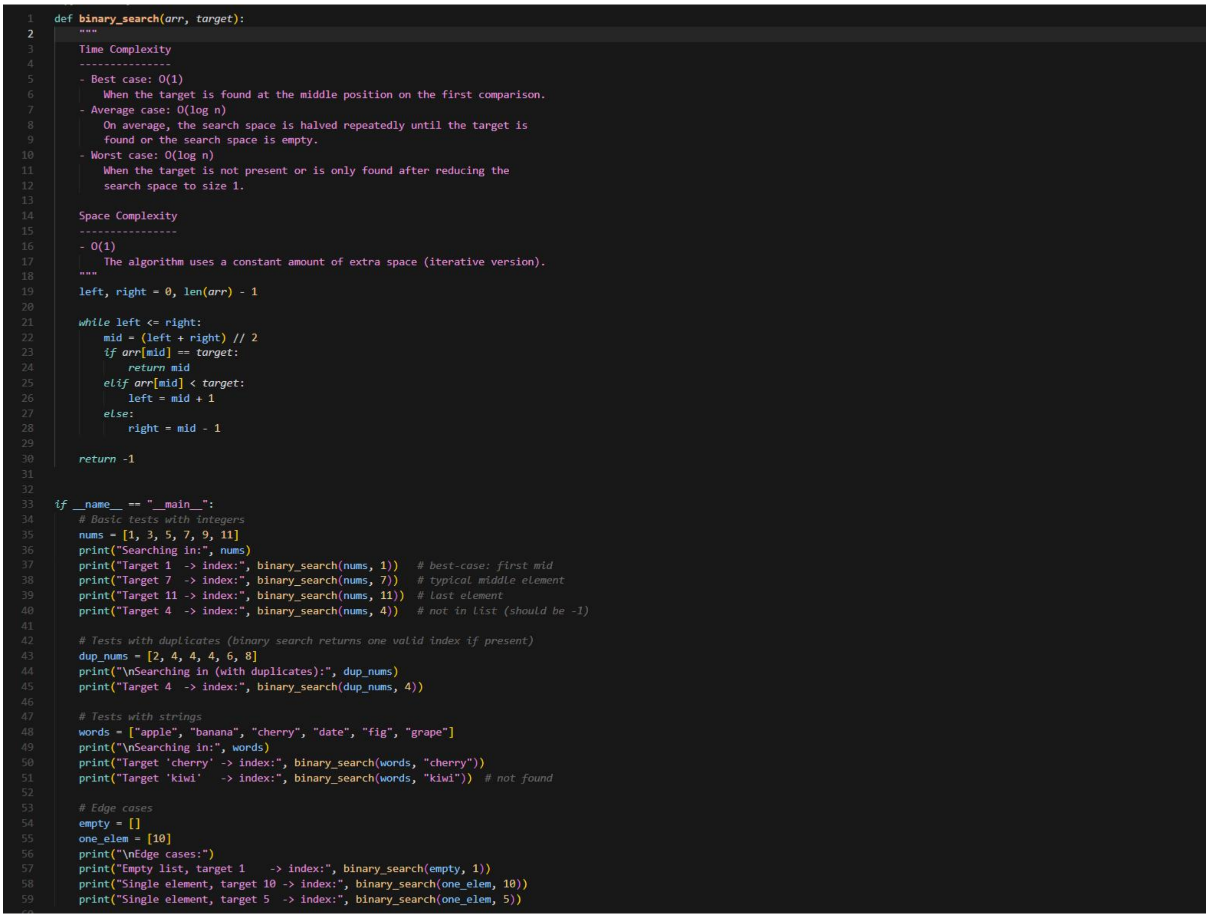
```
PASS: merge_sort([64, 34, 25, 12, 22, 11, 90]) -> [11, 12, 22, 25, 34, 64, 90] (expected [11, 12, 22, 25, 34, 64, 90])
PASS: merge_sort([5, 2, 8, 1, 9]) -> [1, 2, 5, 8, 9] (expected [1, 2, 5, 8, 9])
PASS: merge_sort([1]) -> [1] (expected [1])
PASS: merge_sort([]) -> [] (expected [])
PASS: merge_sort([3, 3, 3, 3]) -> [3, 3, 3, 3] (expected [3, 3, 3, 3])
PASS: merge_sort([2, 1]) -> [1, 2] (expected [1, 2])
PASS: merge_sort([64, 34, 25, 12, 22, 11, 90]) -> [11, 12, 22, 25, 34, 64, 90] (expected [11, 12, 22, 25, 34, 64, 90])
PASS: merge_sort([5, 2, 8, 1, 9]) -> [1, 2, 5, 8, 9] (expected [1, 2, 5, 8, 9])
PASS: merge_sort([1]) -> [1] (expected [1])
PASS: merge_sort([]) -> [] (expected [])
PASS: merge_sort([3, 3, 3, 3]) -> [3, 3, 3, 3] (expected [3, 3, 3, 3])
PASS: merge_sort([5, 2, 8, 1, 9]) -> [1, 2, 5, 8, 9] (expected [1, 2, 5, 8, 9])
PASS: merge_sort([1]) -> [1] (expected [1])
PASS: merge_sort([]) -> [] (expected [])
PASS: merge_sort([3, 3, 3, 3]) -> [3, 3, 3, 3] (expected [3, 3, 3, 3])
PASS: merge_sort([]) -> [] (expected [])
PASS: merge_sort([3, 3, 3, 3]) -> [3, 3, 3, 3] (expected [3, 3, 3, 3])
PASS: merge_sort([3, 3, 3, 3]) -> [3, 3, 3, 3] (expected [3, 3, 3, 3])
PASS: merge_sort([2, 1]) -> [1, 2] (expected [1, 2])
PASS: merge_sort([5, 4, 3, 2, 1]) -> [1, 2, 3, 4, 5] (expected [1, 2, 3, 4, 5])


7/7 test cases passed.
```

Task Description #2 (Searching – Binary Search with AI

Prompt:

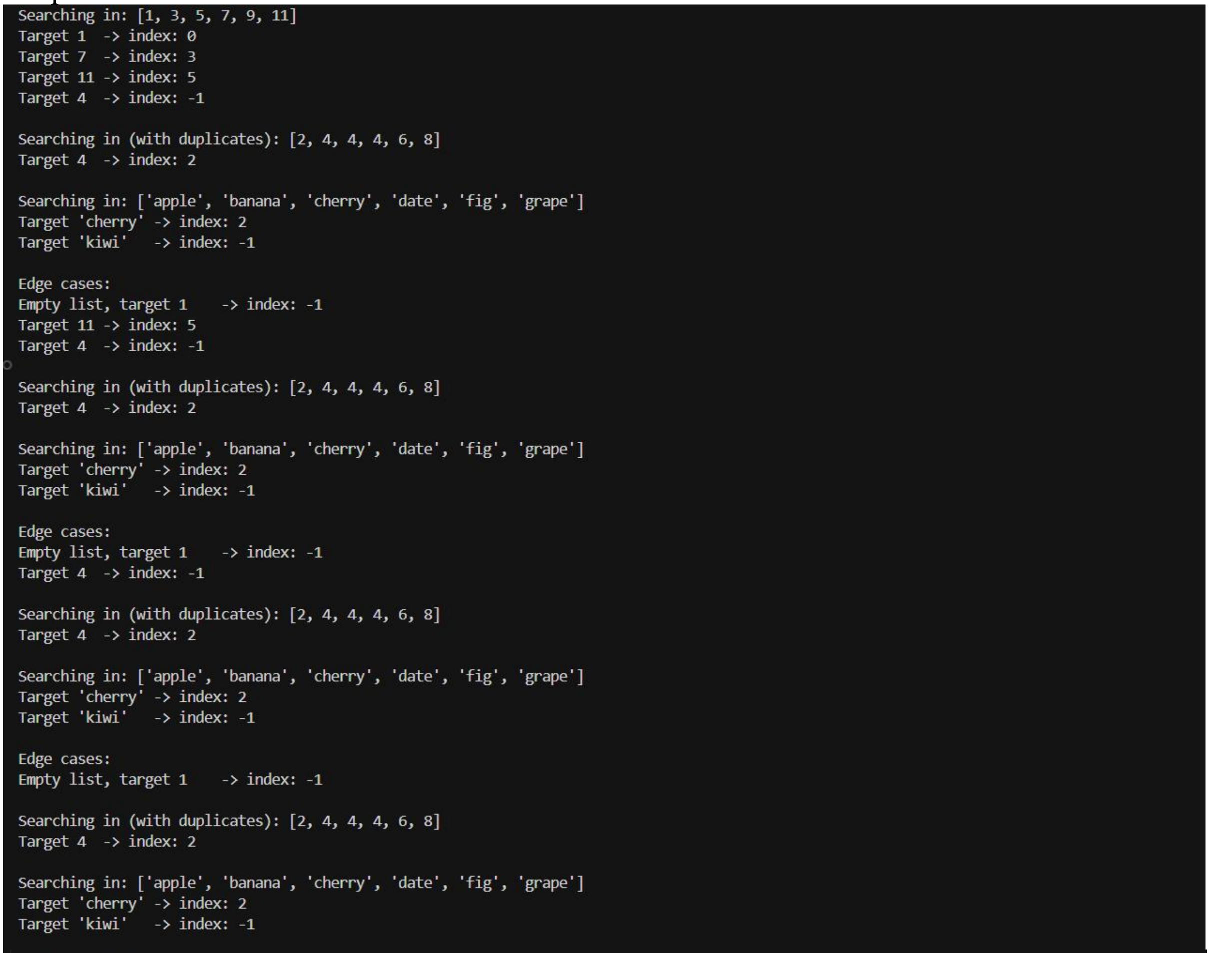
create a function binary_search(arr, target) returning the index of the target or -1 if not found. Include docstrings explaining best, average, and worst-case complexities.Test with various inputs.

Code:

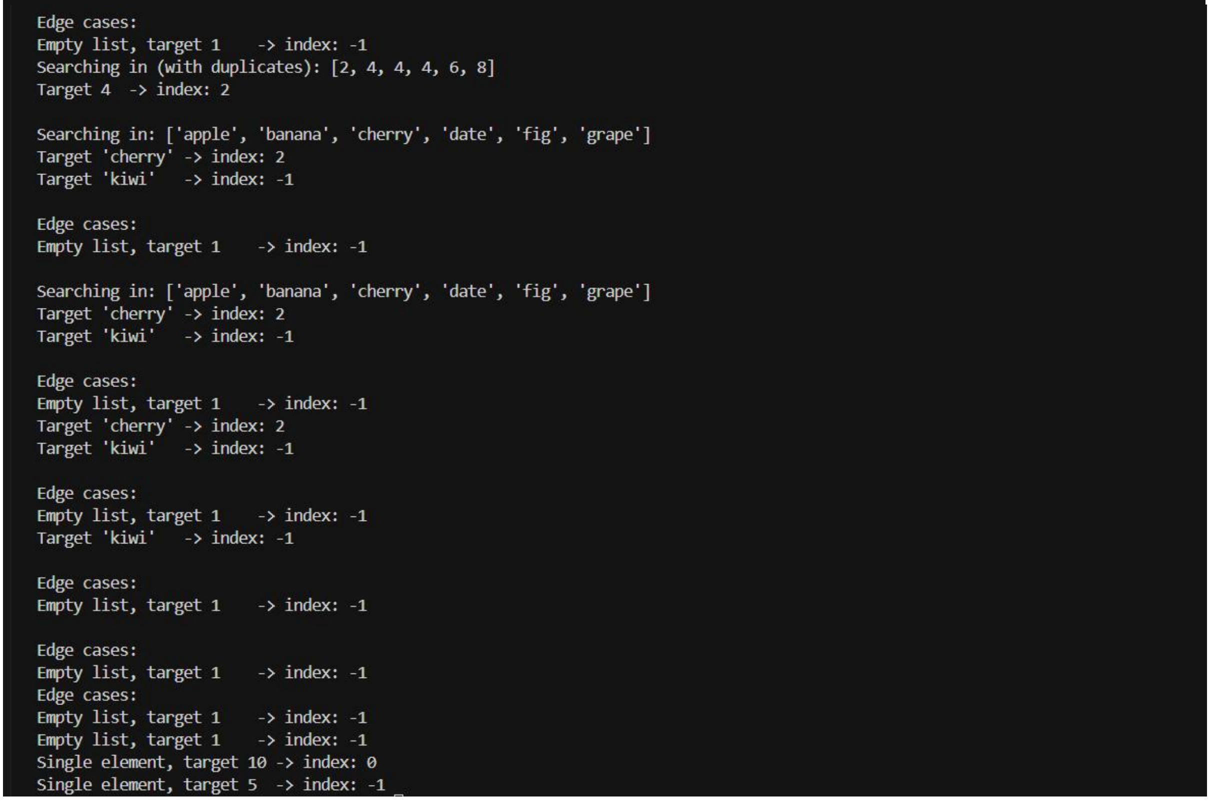


```python
def binary_search(arr, target):
    """
    Time Complexity
    ---------------
    - Best case: O(1)
        When the target is found at the middle position on the first comparison.
    - Average case: O(log n)
        On average, the search space is halved repeatedly until the target is
        found or the search space is empty.
    - Worst case: O(log n)
        When the target is not present or is only found after reducing the
        search space to size 1.

    Space Complexity
    ----------------
    - O(1)
        The algorithm uses a constant amount of extra space (iterative version).
    """
    left, right = 0, len(arr) - 1

    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1

    return -1


if __name__ == "__main__":
    # Basic tests with integers
    nums = [1, 3, 5, 7, 9, 11]
    print("Searching in:", nums)
    print("Target 1  -> index:", binary_search(nums, 1))   # best-case: first mid
    print("Target 7  -> index:", binary_search(nums, 7))   # typical middle element
    print("Target 11 -> index:", binary_search(nums, 11))  # last element
    print("Target 4  -> index:", binary_search(nums, 4))   # not in list (should be -1)

    # Tests with duplicates (binary search returns one valid index if present)
    dup_nums = [2, 4, 4, 4, 6, 8]
    print("\nSearching in (with duplicates):", dup_nums)
    print("Target 4  -> index:", binary_search(dup_nums, 4))

    # Tests with strings
    words = ["apple", "banana", "cherry", "date", "fig", "grape"]
    print("\nSearching in:", words)
    print("Target 'cherry' -> index:", binary_search(words, "cherry"))
    print("Target 'kiwi'   -> index:", binary_search(words, "kiwi"))  # not found

    # Edge cases
    empty = []
    one_elem = [10]
    print("\nEdge cases:")
    print("Empty list, target 1    -> index:", binary_search(empty, 1))
    print("Single element, target 10 -> index:", binary_search(one_elem, 10))
    print("Single element, target 5  -> index:", binary_search(one_elem, 5))
```

Output:



```
Searching in: [1, 3, 5, 7, 9, 11]
Target 1  -> index: 0
Target 7  -> index: 3
Target 11 -> index: 5
Target 4  -> index: -1

Searching in (with duplicates): [2, 4, 4, 4, 6, 8]
Target 4  -> index: 2

Searching in: ['apple', 'banana', 'cherry', 'date', 'fig', 'grape']
Target 'cherry' -> index: 2
Target 'kiwi'   -> index: -1

Edge cases:
Empty list, target 1    -> index: -1
Target 11 -> index: 5
Target 4  -> index: -1

Searching in (with duplicates): [2, 4, 4, 4, 6, 8]
Target 4  -> index: 2

Searching in: ['apple', 'banana', 'cherry', 'date', 'fig', 'grape']
Target 'cherry' -> index: 2
Target 'kiwi'   -> index: -1

Edge cases:
Empty list, target 1    -> index: -1
Target 4  -> index: -1

Searching in (with duplicates): [2, 4, 4, 4, 6, 8]
Target 4  -> index: 2

Searching in: ['apple', 'banana', 'cherry', 'date', 'fig', 'grape']
Target 'cherry' -> index: 2
Target 'kiwi'   -> index: -1

Edge cases:
Empty list, target 1    -> index: -1

Searching in (with duplicates): [2, 4, 4, 4, 6, 8]
Target 4  -> index: 2

Searching in: ['apple', 'banana', 'cherry', 'date', 'fig', 'grape']
Target 'cherry' -> index: 2
Target 'kiwi'   -> index: -1
```



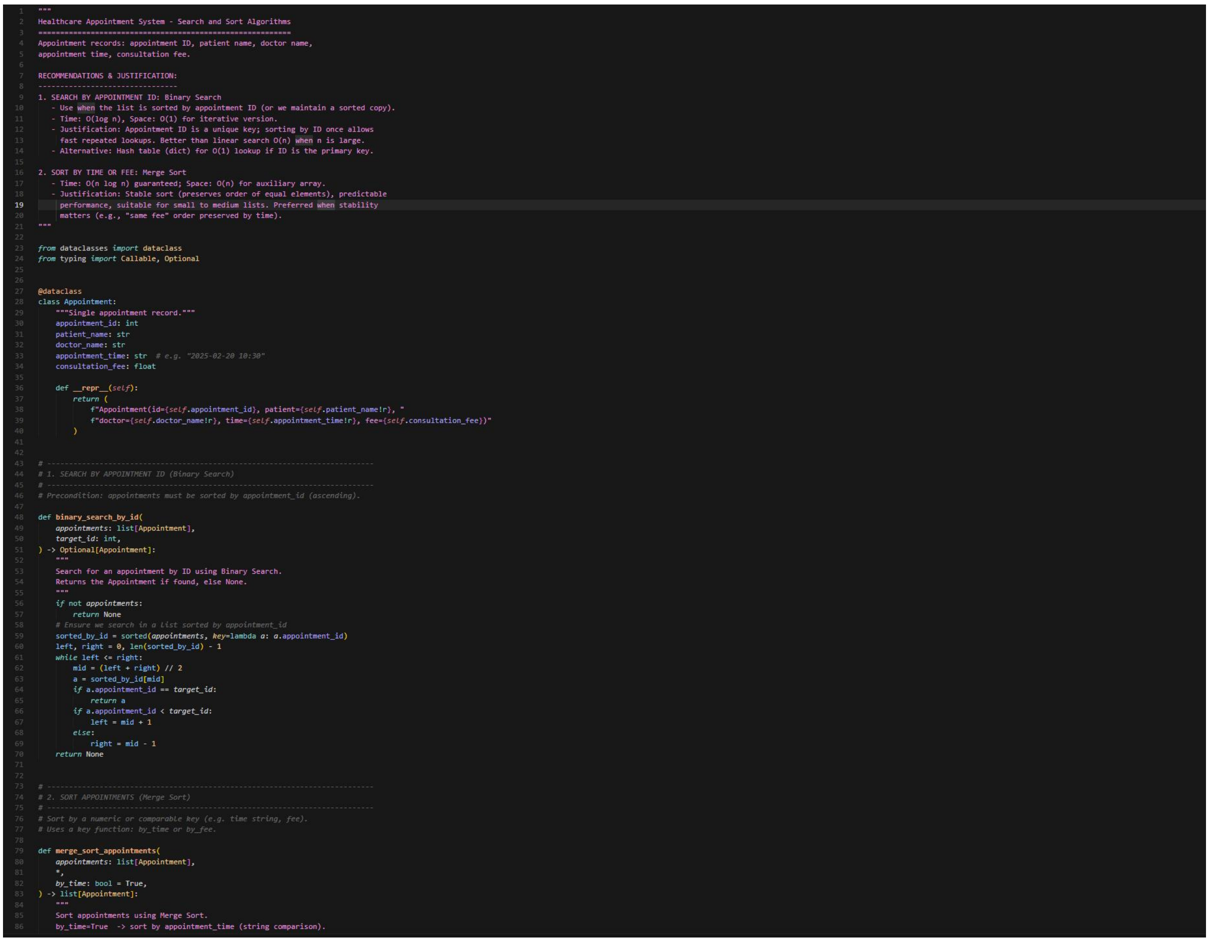
```
Edge cases:
Empty list, target 1    -> index: -1
Searching in (with duplicates): [2, 4, 4, 4, 6, 8]
Target 4  -> index: 2

Searching in: ['apple', 'banana', 'cherry', 'date', 'fig', 'grape']
Target 'cherry' -> index: 2
Target 'kiwi'   -> index: -1

Edge cases:
Empty list, target 1    -> index: -1

Searching in: ['apple', 'banana', 'cherry', 'date', 'fig', 'grape']
Target 'cherry' -> index: 2
Target 'kiwi'   -> index: -1

Edge cases:
Empty list, target 1    -> index: -1
Target 'cherry' -> index: 2
Target 'kiwi'   -> index: -1

Edge cases:
Empty list, target 1    -> index: -1
Target 'kiwi'   -> index: -1

Edge cases:
Empty list, target 1    -> index: -1

Edge cases:
Empty list, target 1    -> index: -1
Edge cases:
Empty list, target 1    -> index: -1
Empty list, target 1    -> index: -1
Single element, target 10 -> index: 0
Single element, target 5  -> index: -1
```

Task Description #3: Smart Healthcare Appointment Scheduling System

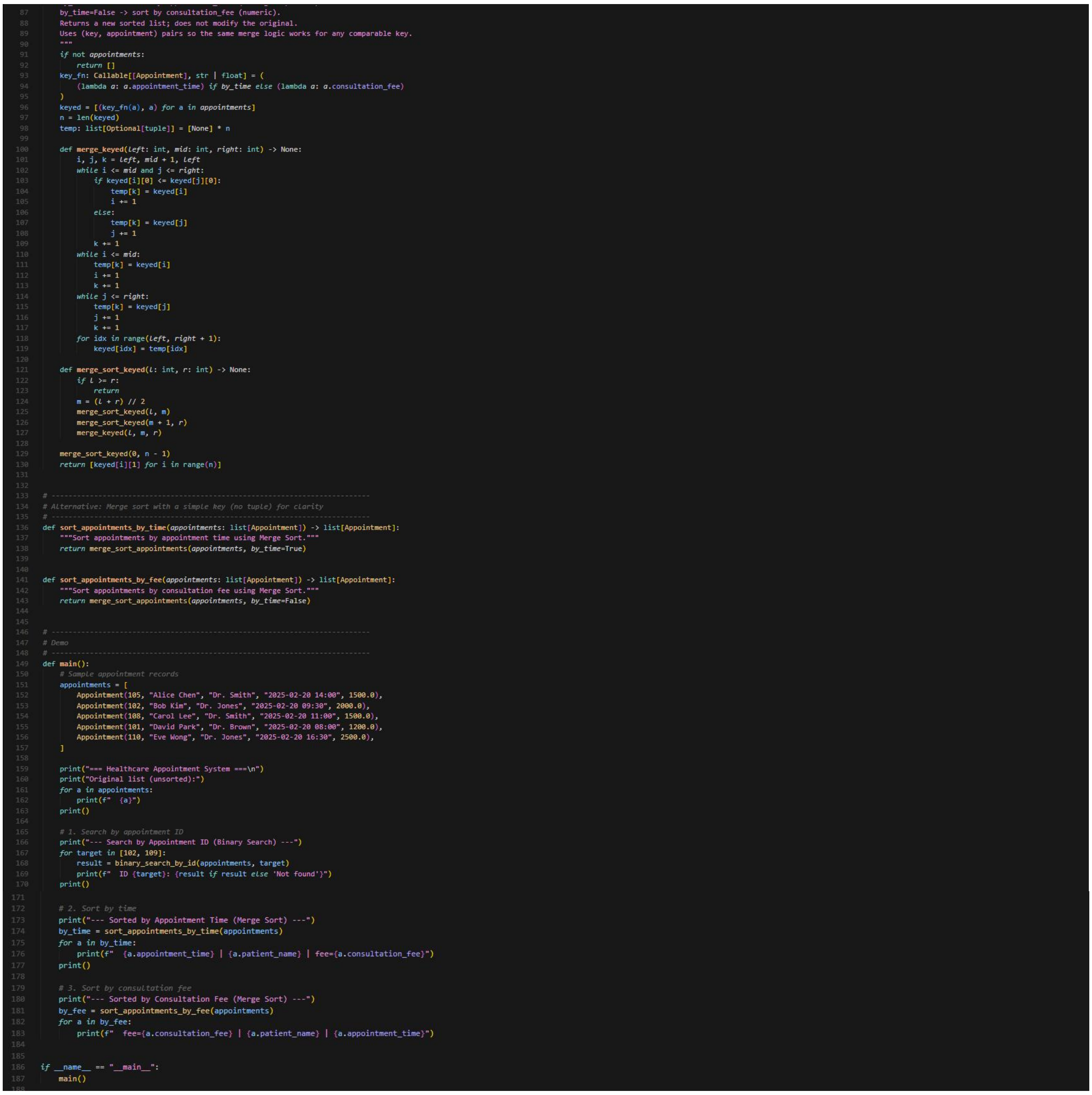
Prompt:
A healthcare platform maintains appointment records containing appointment ID, patient name, doctor name, appointment time, and consultation fee. The system needs to:
1. Search appointments using appointment ID.
2. Sort appointments based on time or consultation fee.

Tasks
- recommend suitable searching and sorting algorithms.
- Justify the selected algorithms.
- Implement the algorithms in Python.

Code:

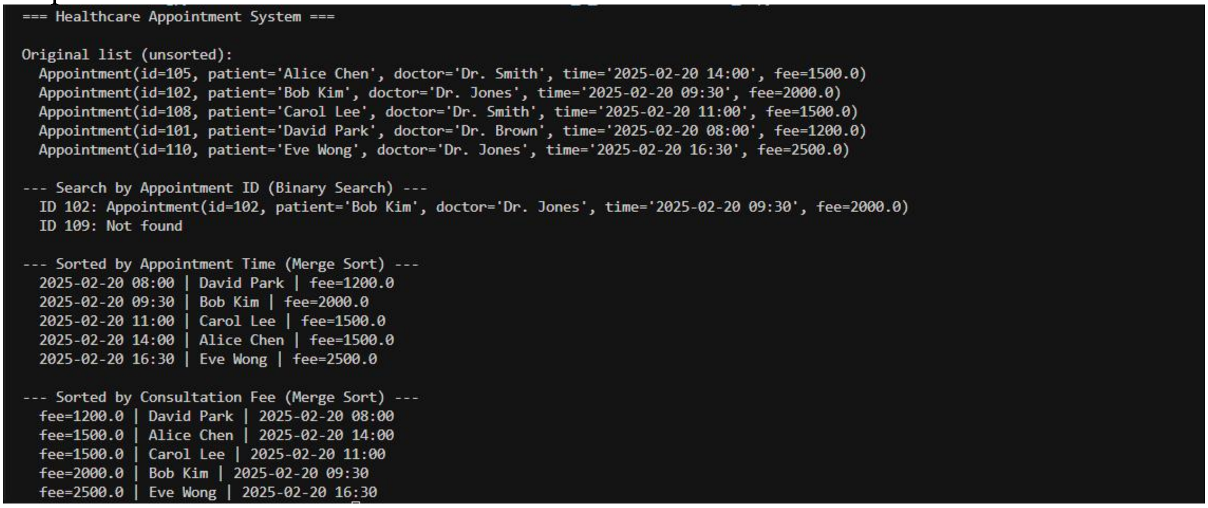


```python
"""
Healthcare Appointment System - Search and Sort Algorithms
==========================================================
Appointment records: appointment ID, patient name, doctor name,
appointment time, consultation fee.

RECOMMENDATIONS & JUSTIFICATION:
--------------------------------
1. SEARCH BY APPOINTMENT ID: Binary Search
   - Use when the list is sorted by appointment ID (or we maintain a sorted copy).
   - Time: O(log n), Space: O(1) for iterative version.
   - Justification: Appointment ID is a unique key; sorting by ID once allows
     fast repeated lookups. Better than linear search O(n) when n is large.
   - Alternative: Hash table (dict) for O(1) lookup if ID is the primary key.

2. SORT BY TIME OR FEE: Merge Sort
   - Time: O(n log n) guaranteed; Space: O(n) for auxiliary array.
   - Justification: Stable sort (preserves order of equal elements), predictable
     performance, suitable for small to medium lists. Preferred when stability
     matters (e.g., "same fee" order preserved by time).
"""

from dataclasses import dataclass
from typing import Callable, Optional


@dataclass
class Appointment:
    """Single appointment record."""
    appointment_id: int
    patient_name: str
    doctor_name: str
    appointment_time: str  # e.g. "2025-02-20 10:30"
    consultation_fee: float

    def __repr__(self):
        return (
            f"Appointment(id={self.appointment_id}, patient={self.patient_name!r}, "
            f"doctor={self.doctor_name!r}, time={self.appointment_time!r}, fee={self.consultation_fee})"
        )


# ---------------------------------------------------------------------------
# 1. SEARCH BY APPOINTMENT ID (Binary Search)
# ---------------------------------------------------------------------------
# Precondition: appointments must be sorted by appointment_id (ascending).

def binary_search_by_id(
    appointments: list[Appointment],
    target_id: int,
) -> Optional[Appointment]:
    """
    Search for an appointment by ID using Binary Search.
    Returns the Appointment if found, else None.
    """
    if not appointments:
        return None
    # Ensure we search in a list sorted by appointment_id
    sorted_by_id = sorted(appointments, key=lambda a: a.appointment_id)
    left, right = 0, len(sorted_by_id) - 1
    while left <= right:
        mid = (left + right) // 2
        a = sorted_by_id[mid]
        if a.appointment_id == target_id:
            return a
        if a.appointment_id < target_id:
            left = mid + 1
        else:
            right = mid - 1
    return None


# ---------------------------------------------------------------------------
# 2. SORT APPOINTMENTS (Merge Sort)
# ---------------------------------------------------------------------------
# Sort by a numeric or comparable key (e.g. time string, fee).
# Uses a key function: by_time or by_fee.

def merge_sort_appointments(
    appointments: list[Appointment],
    *,
    by_time: bool = True,
) -> list[Appointment]:
    """
    Sort appointments using Merge Sort.
    by_time=True  -> sort by appointment_time (string comparison).
```



```python
    by_time=False -> sort by consultation_fee (numeric).
    Returns a new sorted list; does not modify the original.
    Uses (key, appointment) pairs so the same merge logic works for any comparable key.
    """
    if not appointments:
        return []
    key_fn: Callable[[Appointment], str | float] = (
        (lambda a: a.appointment_time) if by_time else (lambda a: a.consultation_fee)
    )
    keyed = [(key_fn(a), a) for a in appointments]
    n = len(keyed)
    temp: list[Optional[tuple]] = [None] * n

    def merge_keyed(left: int, mid: int, right: int) -> None:
        i, j, k = left, mid + 1, left
        while i <= mid and j <= right:
            if keyed[i][0] <= keyed[j][0]:
                temp[k] = keyed[i]
                i += 1
            else:
                temp[k] = keyed[j]
                j += 1
            k += 1
        while i <= mid:
            temp[k] = keyed[i]
            i += 1
            k += 1
        while j <= right:
            temp[k] = keyed[j]
            j += 1
            k += 1
        for idx in range(left, right + 1):
            keyed[idx] = temp[idx]

    def merge_sort_keyed(l: int, r: int) -> None:
        if l >= r:
            return
        m = (l + r) // 2
        merge_sort_keyed(l, m)
        merge_sort_keyed(m + 1, r)
        merge_keyed(l, m, r)

    merge_sort_keyed(0, n - 1)
    return [keyed[i][1] for i in range(n)]


# ---------------------------------------------------------------------------
# Alternative: Merge sort with a simple key (no tuple) for clarity
# ---------------------------------------------------------------------------
def sort_appointments_by_time(appointments: list[Appointment]) -> list[Appointment]:
    """Sort appointments by appointment time using Merge Sort."""
    return merge_sort_appointments(appointments, by_time=True)


def sort_appointments_by_fee(appointments: list[Appointment]) -> list[Appointment]:
    """Sort appointments by consultation fee using Merge Sort."""
    return merge_sort_appointments(appointments, by_time=False)


# ---------------------------------------------------------------------------
# Demo
# ---------------------------------------------------------------------------
def main():
    # Sample appointment records
    appointments = [
        Appointment(105, "Alice Chen", "Dr. Smith", "2025-02-20 14:00", 1500.0),
        Appointment(102, "Bob Kim", "Dr. Jones", "2025-02-20 09:30", 2000.0),
        Appointment(108, "Carol Lee", "Dr. Smith", "2025-02-20 11:00", 1500.0),
        Appointment(101, "David Park", "Dr. Brown", "2025-02-20 08:00", 1200.0),
        Appointment(110, "Eve Wong", "Dr. Jones", "2025-02-20 16:30", 2500.0),
    ]

    print("=== Healthcare Appointment System ===\n")
    print("Original list (unsorted):")
    for a in appointments:
        print(f"  {a}")
    print()

    # 1. Search by appointment ID
    print("--- Search by Appointment ID (Binary Search) ---")
    for target in [102, 109]:
        result = binary_search_by_id(appointments, target)
        print(f"  ID {target}: {result if result else 'Not found'}")
    print()

    # 2. Sort by time
    print("--- Sorted by Appointment Time (Merge Sort) ---")
    by_time = sort_appointments_by_time(appointments)
    for a in by_time:
        print(f"  {a.appointment_time} | {a.patient_name} | fee={a.consultation_fee}")
    print()

    # 3. Sort by consultation fee
    print("--- Sorted by Consultation Fee (Merge Sort) ---")
    by_fee = sort_appointments_by_fee(appointments)
    for a in by_fee:
        print(f"  fee={a.consultation_fee} | {a.patient_name} | {a.appointment_time}")


if __name__ == "__main__":
    main()
```

Output:

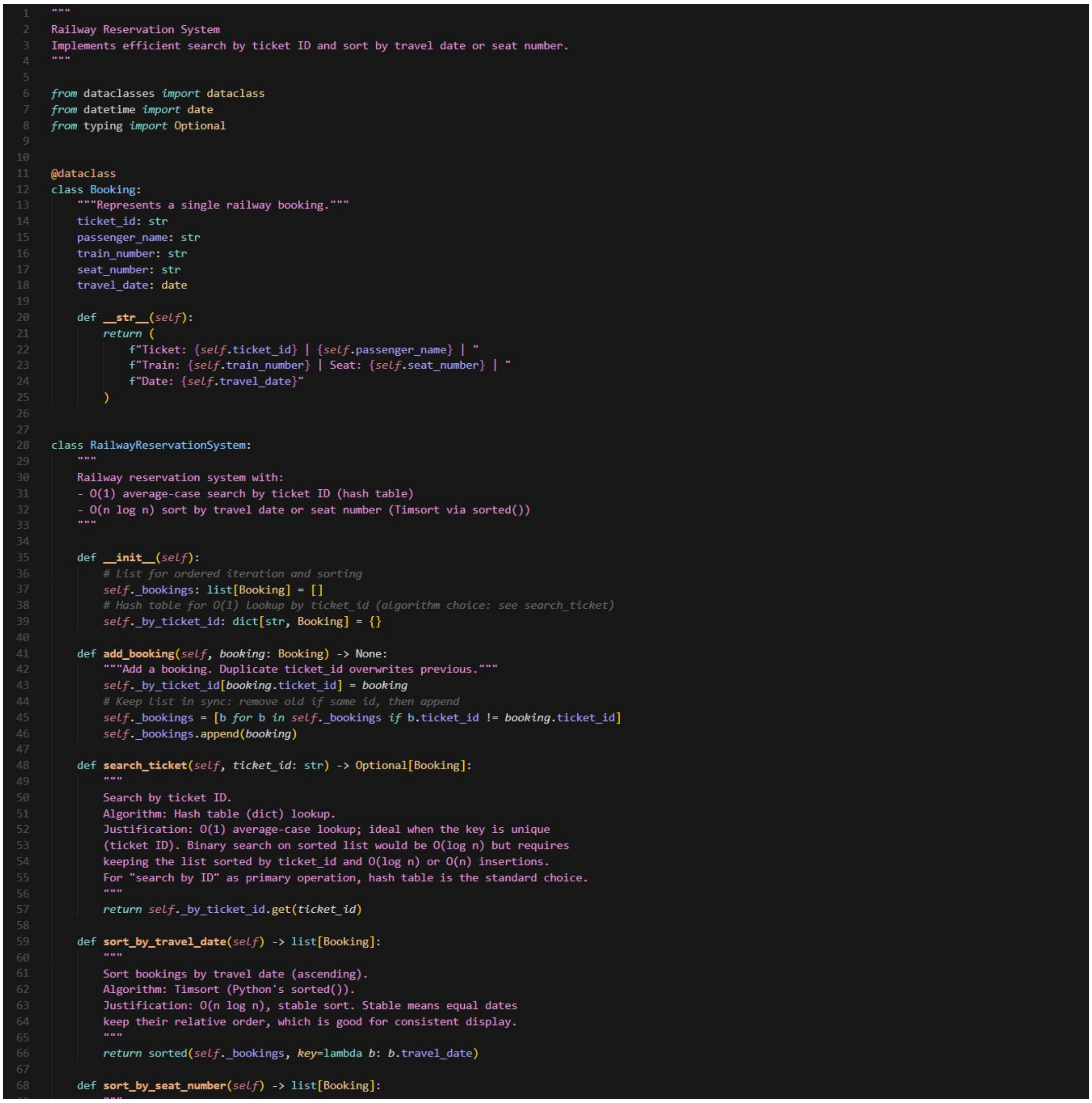


```
=== Healthcare Appointment System ===

Original list (unsorted):
  Appointment(id=105, patient='Alice Chen', doctor='Dr. Smith', time='2025-02-20 14:00', fee=1500.0)
  Appointment(id=102, patient='Bob Kim', doctor='Dr. Jones', time='2025-02-20 09:30', fee=2000.0)
  Appointment(id=108, patient='Carol Lee', doctor='Dr. Smith', time='2025-02-20 11:00', fee=1500.0)
  Appointment(id=101, patient='David Park', doctor='Dr. Brown', time='2025-02-20 08:00', fee=1200.0)
  Appointment(id=110, patient='Eve Wong', doctor='Dr. Jones', time='2025-02-20 16:30', fee=2500.0)

--- Search by Appointment ID (Binary Search) ---
  ID 102: Appointment(id=102, patient='Bob Kim', doctor='Dr. Jones', time='2025-02-20 09:30', fee=2000.0)
  ID 109: Not found

--- Sorted by Appointment Time (Merge Sort) ---
  2025-02-20 08:00 | David Park | fee=1200.0
  2025-02-20 09:30 | Bob Kim | fee=2000.0
  2025-02-20 11:00 | Carol Lee | fee=1500.0
  2025-02-20 14:00 | Alice Chen | fee=1500.0
  2025-02-20 16:30 | Eve Wong | fee=2500.0

--- Sorted by Consultation Fee (Merge Sort) ---
  fee=1200.0 | David Park | 2025-02-20 08:00
  fee=1500.0 | Alice Chen | 2025-02-20 14:00
  fee=1500.0 | Carol Lee | 2025-02-20 11:00
  fee=2000.0 | Bob Kim | 2025-02-20 09:30
  fee=2500.0 | Eve Wong | 2025-02-20 16:30
```

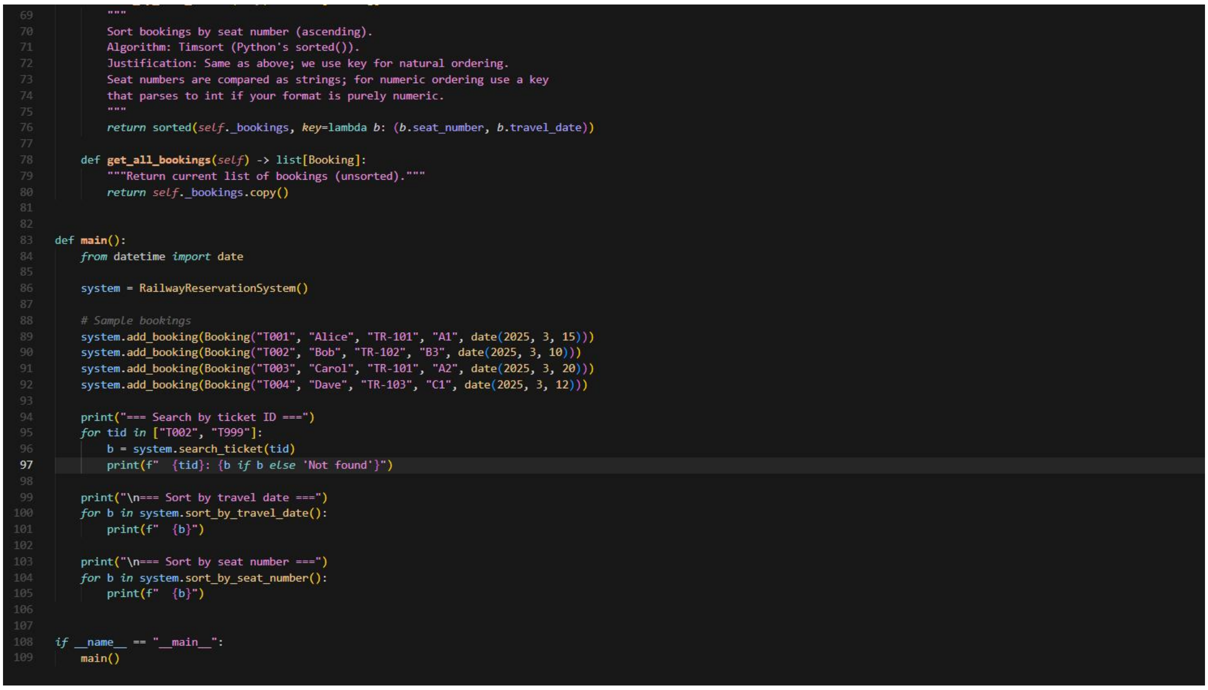
Task Description #4: Railway Ticket Reservation System Scenario

Prompt:
A railway reservation system stores booking details such as ticket ID, passenger name, train number, seat number, and travel date. The system must:
1. Search tickets using ticket ID.
2. Sort bookings based on travel date or seat number.
Tasks
- Identify efficient algorithms.
- Justify the algorithm choices.
- Implement searching and sorting in Python.

Code:

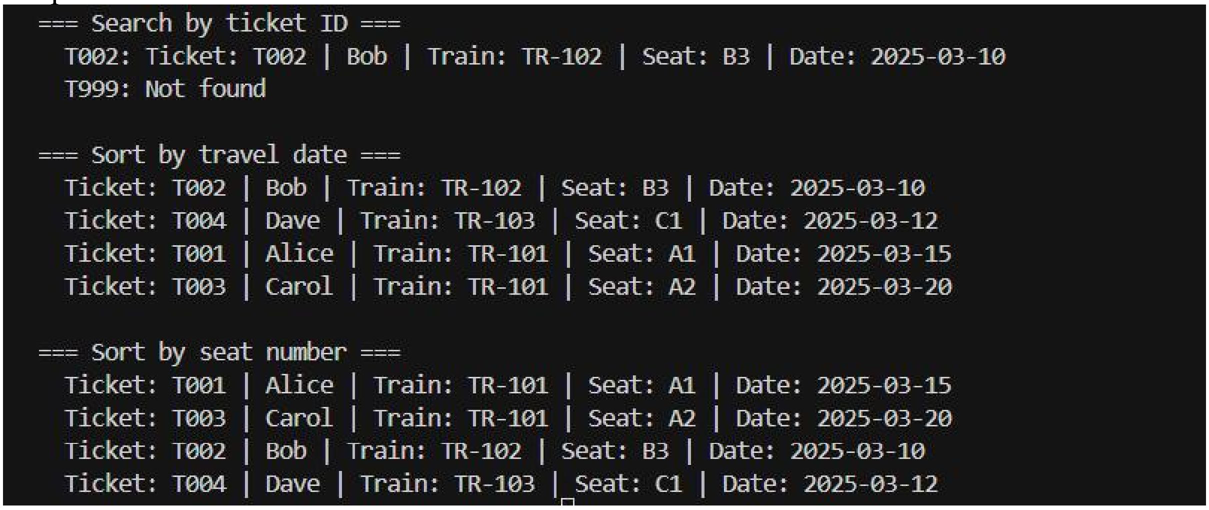


```python
"""
Railway Reservation System
Implements efficient search by ticket ID and sort by travel date or seat number.
"""

from dataclasses import dataclass
from datetime import date
from typing import Optional


@dataclass
class Booking:
    """Represents a single railway booking."""
    ticket_id: str
    passenger_name: str
    train_number: str
    seat_number: str
    travel_date: date

    def __str__(self):
        return (
            f"Ticket: {self.ticket_id} | {self.passenger_name} | "
            f"Train: {self.train_number} | Seat: {self.seat_number} | "
            f"Date: {self.travel_date}"
        )


class RailwayReservationSystem:
    """
    Railway reservation system with:
    - O(1) average-case search by ticket ID (hash table)
    - O(n log n) sort by travel date or seat number (Timsort via sorted())
    """

    def __init__(self):
        # List for ordered iteration and sorting
        self._bookings: list[Booking] = []
        # Hash table for O(1) lookup by ticket_id (algorithm choice: see search_ticket)
        self._by_ticket_id: dict[str, Booking] = {}

    def add_booking(self, booking: Booking) -> None:
        """Add a booking. Duplicate ticket_id overwrites previous."""
        self._by_ticket_id[booking.ticket_id] = booking
        # Keep list in sync: remove old if same id, then append
        self._bookings = [b for b in self._bookings if b.ticket_id != booking.ticket_id]
        self._bookings.append(booking)

    def search_ticket(self, ticket_id: str) -> Optional[Booking]:
        """
        Search by ticket ID.
        Algorithm: Hash table (dict) lookup.
        Justification: O(1) average-case lookup; ideal when the key is unique
        (ticket ID). Binary search on sorted list would be O(log n) but requires
        keeping the list sorted by ticket_id and O(log n) or O(n) insertions.
        For "search by ID" as primary operation, hash table is the standard choice.
        """
        return self._by_ticket_id.get(ticket_id)

    def sort_by_travel_date(self) -> list[Booking]:
        """
        Sort bookings by travel date (ascending).
        Algorithm: Timsort (Python's sorted()).
        Justification: O(n log n), stable sort. Stable means equal dates
        keep their relative order, which is good for consistent display.
        """
        return sorted(self._bookings, key=lambda b: b.travel_date)

    def sort_by_seat_number(self) -> list[Booking]:
```



```python
        """
        Sort bookings by seat number (ascending).
        Algorithm: Timsort (Python's sorted()).
        Justification: Same as above; we use key for natural ordering.
        Seat numbers are compared as strings; for numeric ordering use a key
        that parses to int if your format is purely numeric.
        """
        return sorted(self._bookings, key=lambda b: (b.seat_number, b.travel_date))

    def get_all_bookings(self) -> list[Booking]:
        """Return current list of bookings (unsorted)."""
        return self._bookings.copy()


def main():
    from datetime import date

    system = RailwayReservationSystem()

    # Sample bookings
    system.add_booking(Booking("T001", "Alice", "TR-101", "A1", date(2025, 3, 15)))
    system.add_booking(Booking("T002", "Bob", "TR-102", "B3", date(2025, 3, 10)))
    system.add_booking(Booking("T003", "Carol", "TR-101", "A2", date(2025, 3, 20)))
    system.add_booking(Booking("T004", "Dave", "TR-103", "C1", date(2025, 3, 12)))

    print("=== Search by ticket ID ===")
    for tid in ["T002", "T999"]:
        b = system.search_ticket(tid)
        print(f"  {tid}: {b if b else 'Not found'}")

    print("\n=== Sort by travel date ===")
    for b in system.sort_by_travel_date():
        print(f"  {b}")

    print("\n=== Sort by seat number ===")
    for b in system.sort_by_seat_number():
        print(f"  {b}")


if __name__ == "__main__":
    main()
```

Output:



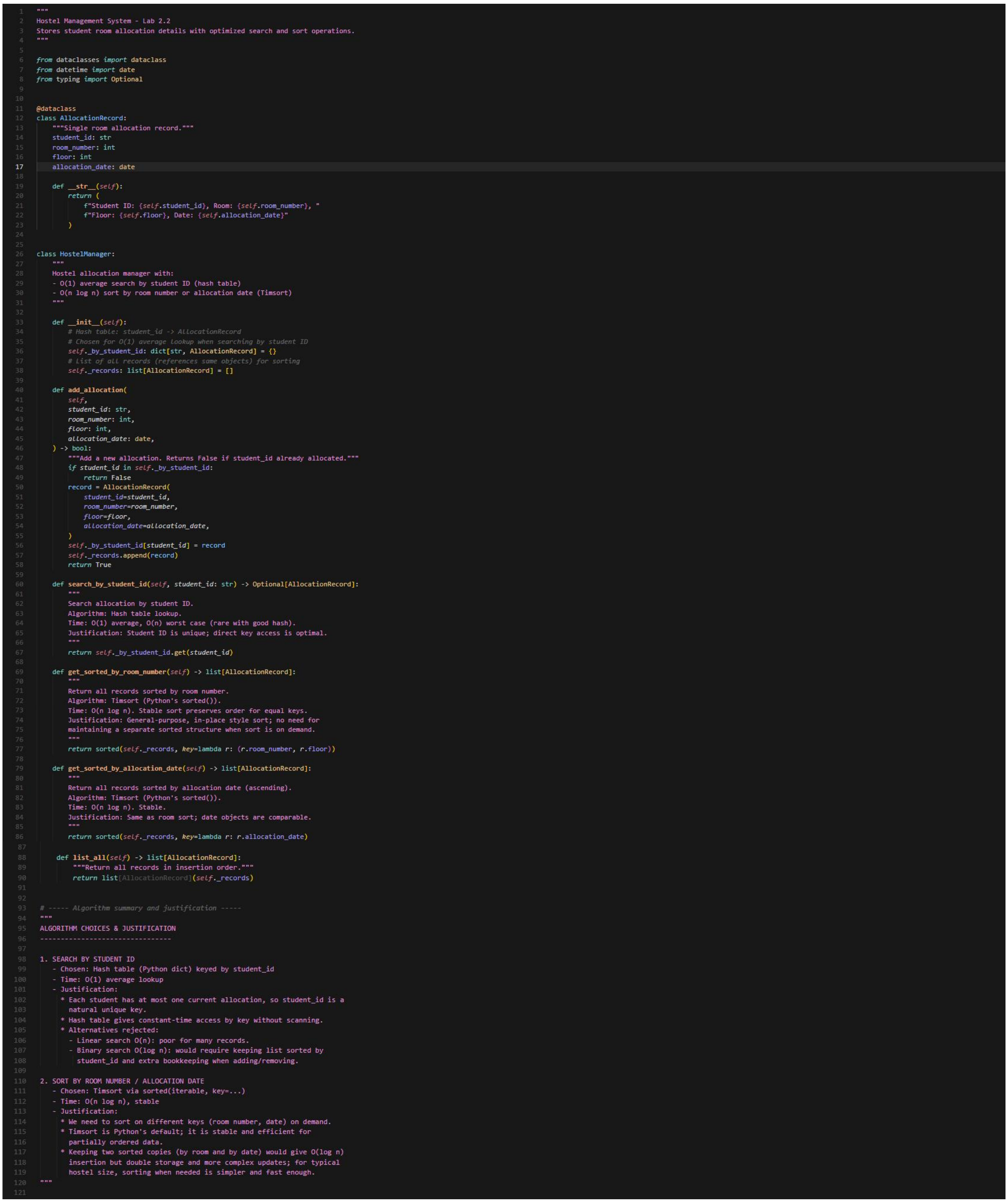
```
=== Search by ticket ID ===
  T002: Ticket: T002 | Bob | Train: TR-102 | Seat: B3 | Date: 2025-03-10
  T999: Not found

=== Sort by travel date ===
  Ticket: T002 | Bob | Train: TR-102 | Seat: B3 | Date: 2025-03-10
  Ticket: T004 | Dave | Train: TR-103 | Seat: C1 | Date: 2025-03-12
  Ticket: T001 | Alice | Train: TR-101 | Seat: A1 | Date: 2025-03-15
  Ticket: T003 | Carol | Train: TR-101 | Seat: A2 | Date: 2025-03-20

=== Sort by seat number ===
  Ticket: T001 | Alice | Train: TR-101 | Seat: A1 | Date: 2025-03-15
  Ticket: T003 | Carol | Train: TR-101 | Seat: A2 | Date: 2025-03-20
  Ticket: T002 | Bob | Train: TR-102 | Seat: B3 | Date: 2025-03-10
  Ticket: T004 | Dave | Train: TR-103 | Seat: C1 | Date: 2025-03-12
```

Task Description #5: Smart Hostel Room Allocation System

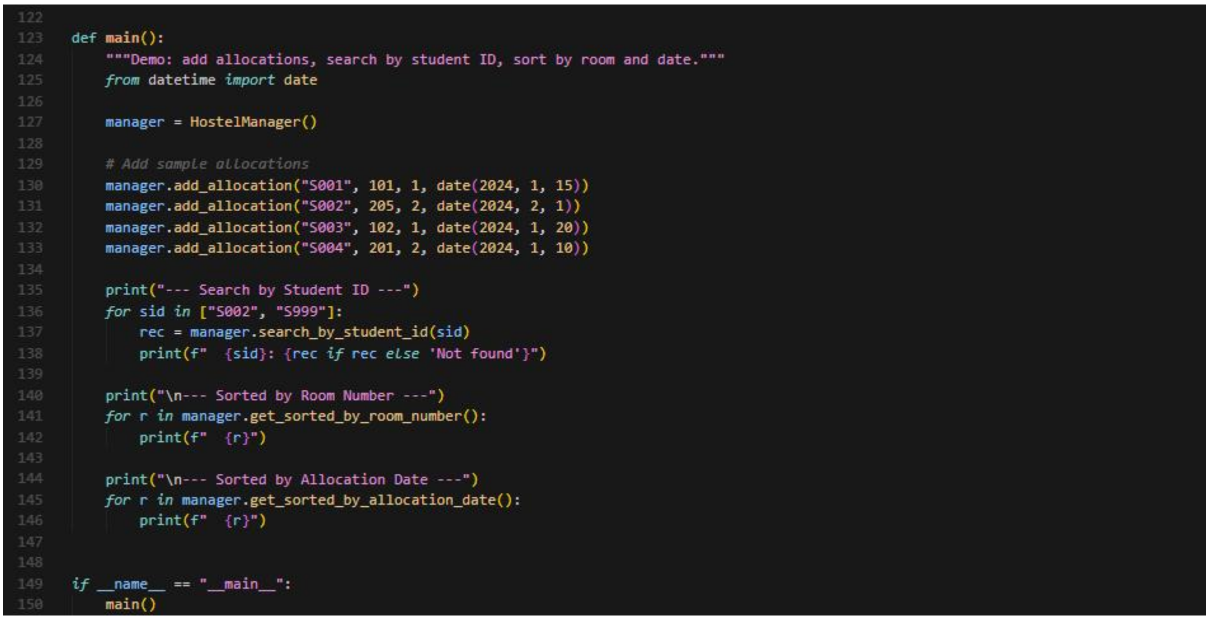
Prompt:
A hostel management system stores student room allocation details including student ID, room number, floor, and allocation date. The system needs to:
1. Search allocation details using student ID.
2. Sort records based on room number or allocation date.

Tasks
- suggest optimized algorithms.
- Justify the selections.
- Implement the solution in Python.

Code:

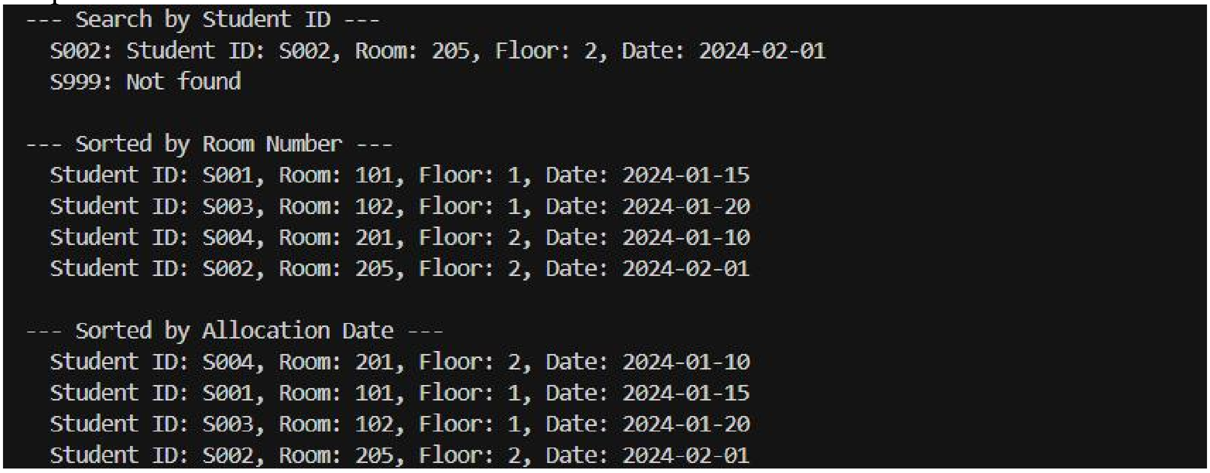


```python
"""
Hostel Management System - Lab 2.2
Stores student room allocation details with optimized search and sort operations.
"""

from dataclasses import dataclass
from datetime import date
from typing import Optional


@dataclass
class AllocationRecord:
    """Single room allocation record."""
    student_id: str
    room_number: int
    floor: int
    allocation_date: date

    def __str__(self):
        return (
            f"Student ID: {self.student_id}, Room: {self.room_number}, "
            f"Floor: {self.floor}, Date: {self.allocation_date}"
        )


class HostelManager:
    """
    Hostel allocation manager with:
    - O(1) average search by student ID (hash table)
    - O(n log n) sort by room number or allocation date (Timsort)
    """

    def __init__(self):
        # Hash table: student_id -> AllocationRecord
        # Chosen for O(1) average lookup when searching by student ID
        self._by_student_id: dict[str, AllocationRecord] = {}
        # List of all records (references same objects) for sorting
        self._records: list[AllocationRecord] = []

    def add_allocation(
        self,
        student_id: str,
        room_number: int,
        floor: int,
        allocation_date: date,
    ) -> bool:
        """Add a new allocation. Returns False if student_id already allocated."""
        if student_id in self._by_student_id:
            return False
        record = AllocationRecord(
            student_id=student_id,
            room_number=room_number,
            floor=floor,
            allocation_date=allocation_date,
        )
        self._by_student_id[student_id] = record
        self._records.append(record)
        return True

    def search_by_student_id(self, student_id: str) -> Optional[AllocationRecord]:
        """
        Search allocation by student ID.
        Algorithm: Hash table lookup.
        Time: O(1) average, O(n) worst case (rare with good hash).
        Justification: Student ID is unique; direct key access is optimal.
        """
        return self._by_student_id.get(student_id)

    def get_sorted_by_room_number(self) -> list[AllocationRecord]:
        """
        Return all records sorted by room number.
        Algorithm: Timsort (Python's sorted()).
        Time: O(n log n). Stable sort preserves order for equal keys.
        Justification: General-purpose, in-place style sort; no need for
        maintaining a separate sorted structure when sort is on demand.
        """
        return sorted(self._records, key=lambda r: (r.room_number, r.floor))

    def get_sorted_by_allocation_date(self) -> list[AllocationRecord]:
        """
        Return all records sorted by allocation date (ascending).
        Algorithm: Timsort (Python's sorted()).
        Time: O(n log n). Stable.
        Justification: Same as room sort; date objects are comparable.
        """
        return sorted(self._records, key=lambda r: r.allocation_date)

     def list_all(self) -> list[AllocationRecord]:
         """Return all records in insertion order."""
         return list[AllocationRecord](self._records)


# ----- Algorithm summary and justification -----
"""
ALGORITHM CHOICES & JUSTIFICATION
--------------------------------

1. SEARCH BY STUDENT ID
   - Chosen: Hash table (Python dict) keyed by student_id
   - Time: O(1) average lookup
   - Justification:
     * Each student has at most one current allocation, so student_id is a
       natural unique key.
     * Hash table gives constant-time access by key without scanning.
     * Alternatives rejected:
       - Linear search O(n): poor for many records.
       - Binary search O(log n): would require keeping list sorted by
         student_id and extra bookkeeping when adding/removing.

2. SORT BY ROOM NUMBER / ALLOCATION DATE
   - Chosen: Timsort via sorted(iterable, key=...)
   - Time: O(n log n), stable
   - Justification:
     * We need to sort on different keys (room number, date) on demand.
     * Timsort is Python's default; it is stable and efficient for
       partially ordered data.
     * Keeping two sorted copies (by room and by date) would give O(log n)
       insertion but double storage and more complex updates; for typical
       hostel size, sorting when needed is simpler and fast enough.
"""
```



```python

def main():
    """Demo: add allocations, search by student ID, sort by room and date."""
    from datetime import date

    manager = HostelManager()

    # Add sample allocations
    manager.add_allocation("S001", 101, 1, date(2024, 1, 15))
    manager.add_allocation("S002", 205, 2, date(2024, 2, 1))
    manager.add_allocation("S003", 102, 1, date(2024, 1, 20))
    manager.add_allocation("S004", 201, 2, date(2024, 1, 10))

    print("--- Search by Student ID ---")
    for sid in ["S002", "S999"]:
        rec = manager.search_by_student_id(sid)
        print(f"  {sid}: {rec if rec else 'Not found'}")

    print("\n--- Sorted by Room Number ---")
    for r in manager.get_sorted_by_room_number():
        print(f"  {r}")

    print("\n--- Sorted by Allocation Date ---")
    for r in manager.get_sorted_by_allocation_date():
        print(f"  {r}")


if __name__ == "__main__":
    main()
```

Output:



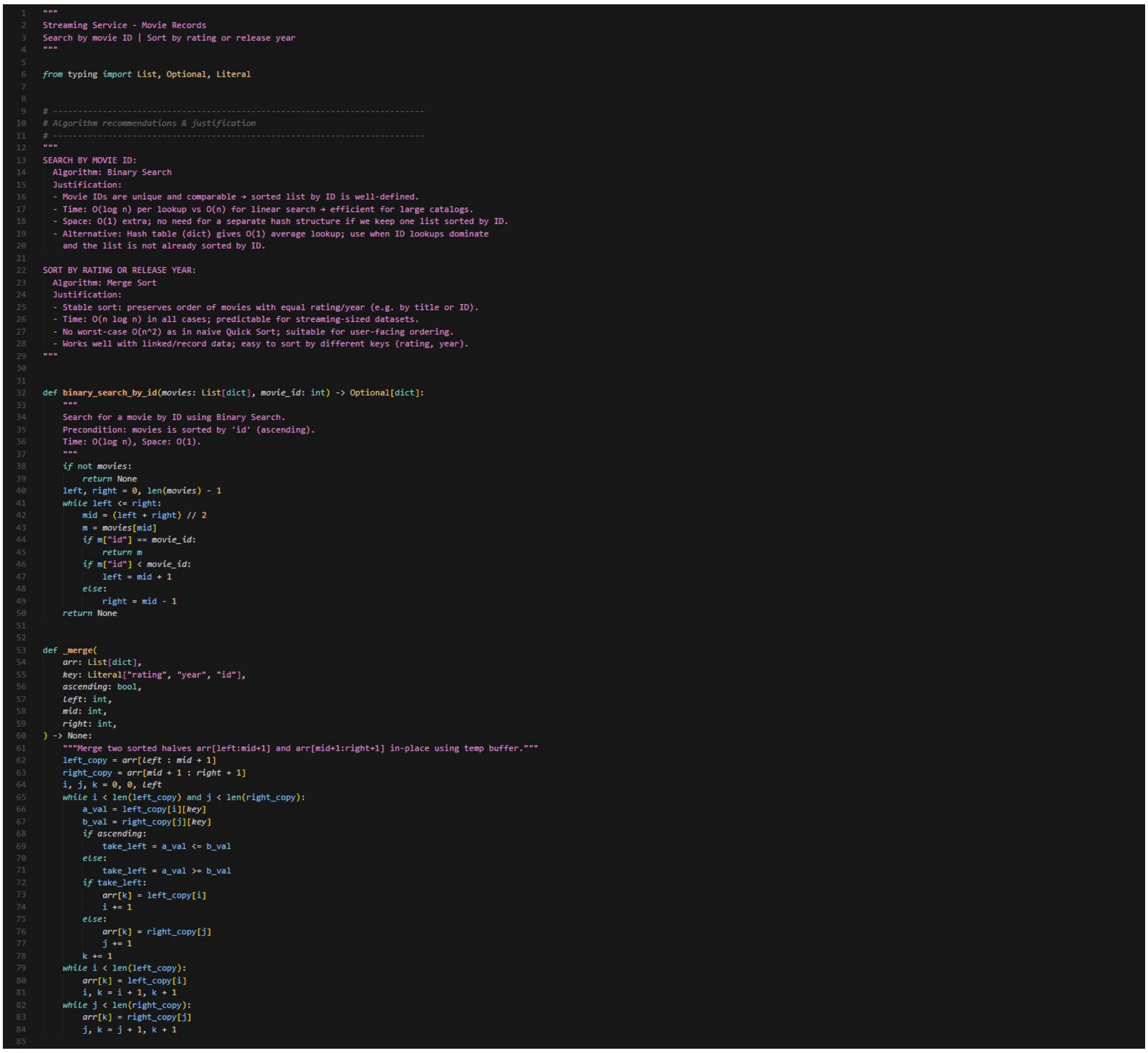
```
--- Search by Student ID ---
  S002: Student ID: S002, Room: 205, Floor: 2, Date: 2024-02-01
  S999: Not found

--- Sorted by Room Number ---
  Student ID: S001, Room: 101, Floor: 1, Date: 2024-01-15
  Student ID: S003, Room: 102, Floor: 1, Date: 2024-01-20
  Student ID: S004, Room: 201, Floor: 2, Date: 2024-01-10
  Student ID: S002, Room: 205, Floor: 2, Date: 2024-02-01

--- Sorted by Allocation Date ---
  Student ID: S004, Room: 201, Floor: 2, Date: 2024-01-10
  Student ID: S001, Room: 101, Floor: 1, Date: 2024-01-15
  Student ID: S003, Room: 102, Floor: 1, Date: 2024-01-20
  Student ID: S002, Room: 205, Floor: 2, Date: 2024-02-01
```

Task Description #6: Online Movie Streaming Platform

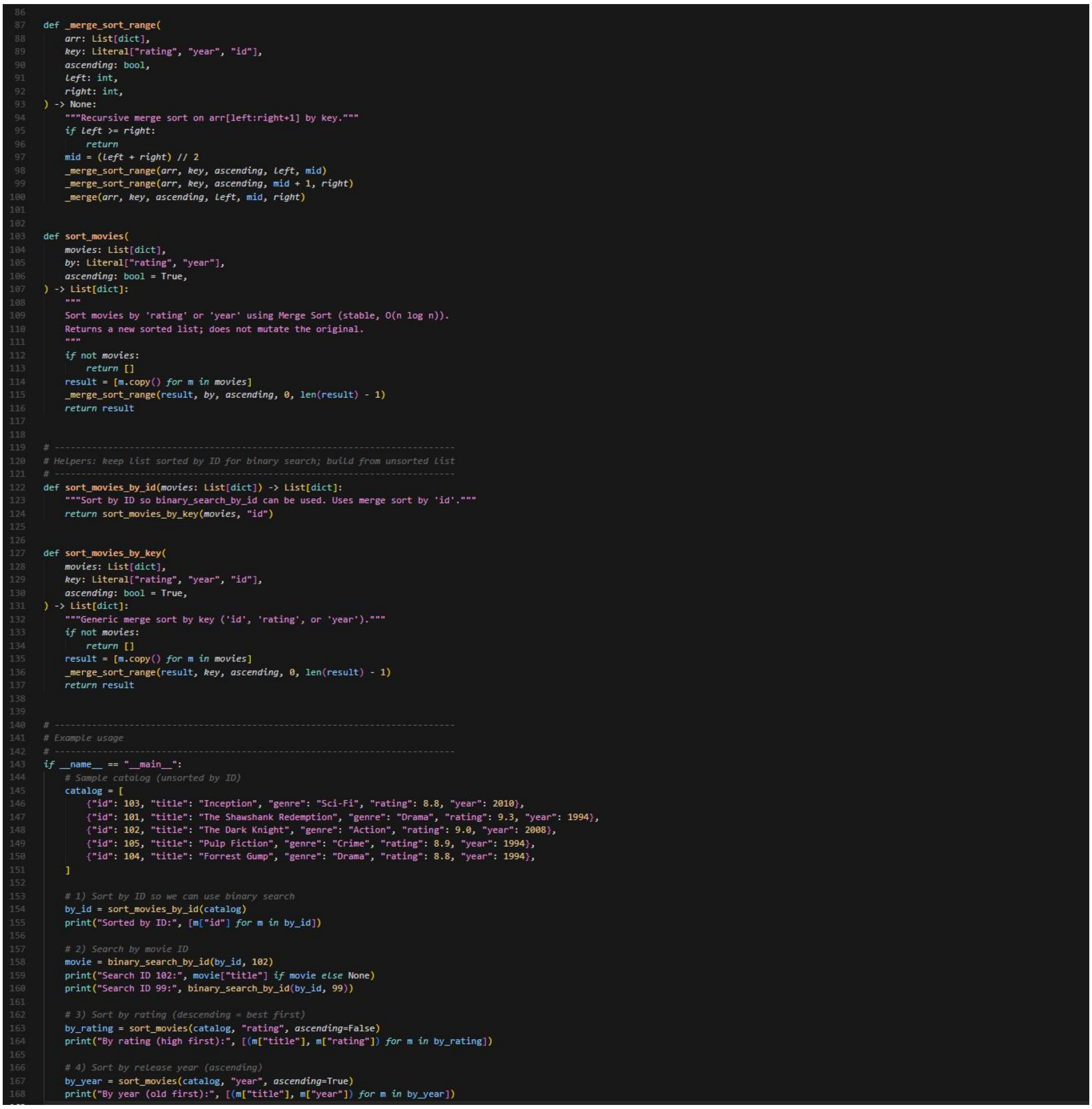
Prompt:
A streaming service maintains movie records with movie ID, title, genre, rating, and release year. The platform needs to:
1. Search movies by movie ID.
2. Sort movies based on rating or release year.

Tasks
- Recommend searching and sorting algorithms.
- Justify the chosen algorithms.
- Implement Python functions.

Code:

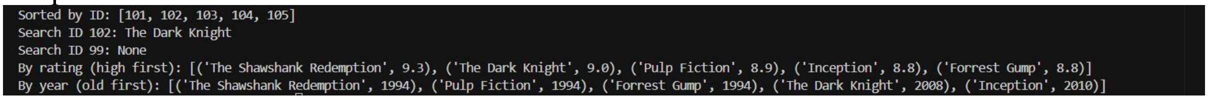


```python
"""
Streaming Service - Movie Records
Search by movie ID | Sort by rating or release year
"""

from typing import List, Optional, Literal


# ---------------------------------------------------------------------------
# Algorithm recommendations & justification
# ---------------------------------------------------------------------------
"""
SEARCH BY MOVIE ID:
  Algorithm: Binary Search
  Justification:
  - Movie IDs are unique and comparable → sorted list by ID is well-defined.
  - Time: O(log n) per lookup vs O(n) for linear search → efficient for large catalogs.
  - Space: O(1) extra; no need for a separate hash structure if we keep one list sorted by ID.
  - Alternative: Hash table (dict) gives O(1) average lookup; use when ID lookups dominate
    and the list is not already sorted by ID.

SORT BY RATING OR RELEASE YEAR:
  Algorithm: Merge Sort
  Justification:
  - Stable sort: preserves order of movies with equal rating/year (e.g. by title or ID).
  - Time: O(n log n) in all cases; predictable for streaming-sized datasets.
  - No worst-case O(n^2) as in naive Quick Sort; suitable for user-facing ordering.
  - Works well with linked/record data; easy to sort by different keys (rating, year).
"""


def binary_search_by_id(movies: List[dict], movie_id: int) -> Optional[dict]:
    """
    Search for a movie by ID using Binary Search.
    Precondition: movies is sorted by 'id' (ascending).
    Time: O(log n), Space: O(1).
    """
    if not movies:
        return None
    left, right = 0, len(movies) - 1
    while left <= right:
        mid = (left + right) // 2
        m = movies[mid]
        if m["id"] == movie_id:
            return m
        if m["id"] < movie_id:
            left = mid + 1
        else:
            right = mid - 1
    return None


def _merge(
    arr: List[dict],
    key: Literal["rating", "year", "id"],
    ascending: bool,
    left: int,
    mid: int,
    right: int,
) -> None:
    """Merge two sorted halves arr[left:mid+1] and arr[mid+1:right+1] in-place using temp buffer."""
    left_copy = arr[left : mid + 1]
    right_copy = arr[mid + 1 : right + 1]
    i, j, k = 0, 0, left
    while i < len(left_copy) and j < len(right_copy):
        a_val = left_copy[i][key]
        b_val = right_copy[j][key]
        if ascending:
            take_left = a_val <= b_val
        else:
            take_left = a_val >= b_val
        if take_left:
            arr[k] = left_copy[i]
            i += 1
        else:
            arr[k] = right_copy[j]
            j += 1
        k += 1
    while i < len(left_copy):
        arr[k] = left_copy[i]
        i, k = i + 1, k + 1
    while j < len(right_copy):
        arr[k] = right_copy[j]
        j, k = j + 1, k + 1

```



```python

def _merge_sort_range(
    arr: List[dict],
    key: Literal["rating", "year", "id"],
    ascending: bool,
    left: int,
    right: int,
) -> None:
    """Recursive merge sort on arr[left:right+1] by key."""
    if left >= right:
        return
    mid = (left + right) // 2
    _merge_sort_range(arr, key, ascending, left, mid)
    _merge_sort_range(arr, key, ascending, mid + 1, right)
    _merge(arr, key, ascending, left, mid, right)


def sort_movies(
    movies: List[dict],
    by: Literal["rating", "year"],
    ascending: bool = True,
) -> List[dict]:
    """
    Sort movies by 'rating' or 'year' using Merge Sort (stable, O(n log n)).
    Returns a new sorted list; does not mutate the original.
    """
    if not movies:
        return []
    result = [m.copy() for m in movies]
    _merge_sort_range(result, by, ascending, 0, len(result) - 1)
    return result


# ---------------------------------------------------------------------------
# Helpers: keep list sorted by ID for binary search; build from unsorted list
# ---------------------------------------------------------------------------
def sort_movies_by_id(movies: List[dict]) -> List[dict]:
    """Sort by ID so binary_search_by_id can be used. Uses merge sort by 'id'."""
    return sort_movies_by_key(movies, "id")


def sort_movies_by_key(
    movies: List[dict],
    key: Literal["rating", "year", "id"],
    ascending: bool = True,
) -> List[dict]:
    """Generic merge sort by key ('id', 'rating', or 'year')."""
    if not movies:
        return []
    result = [m.copy() for m in movies]
    _merge_sort_range(result, key, ascending, 0, len(result) - 1)
    return result


# ---------------------------------------------------------------------------
# Example usage
# ---------------------------------------------------------------------------
if __name__ == "__main__":
    # Sample catalog (unsorted by ID)
    catalog = [
        {"id": 103, "title": "Inception", "genre": "Sci-Fi", "rating": 8.8, "year": 2010},
        {"id": 101, "title": "The Shawshank Redemption", "genre": "Drama", "rating": 9.3, "year": 1994},
        {"id": 102, "title": "The Dark Knight", "genre": "Action", "rating": 9.0, "year": 2008},
        {"id": 105, "title": "Pulp Fiction", "genre": "Crime", "rating": 8.9, "year": 1994},
        {"id": 104, "title": "Forrest Gump", "genre": "Drama", "rating": 8.8, "year": 1994},
    ]

    # 1) Sort by ID so we can use binary search
    by_id = sort_movies_by_id(catalog)
    print("Sorted by ID:", [m["id"] for m in by_id])

    # 2) Search by movie ID
    movie = binary_search_by_id(by_id, 102)
    print("Search ID 102:", movie["title"] if movie else None)
    print("Search ID 99:", binary_search_by_id(by_id, 99))

    # 3) Sort by rating (descending = best first)
    by_rating = sort_movies(catalog, "rating", ascending=False)
    print("By rating (high first):", [(m["title"], m["rating"]) for m in by_rating])

    # 4) Sort by release year (ascending)
    by_year = sort_movies(catalog, "year", ascending=True)
    print("By year (old first):", [(m["title"], m["year"]) for m in by_year])
```

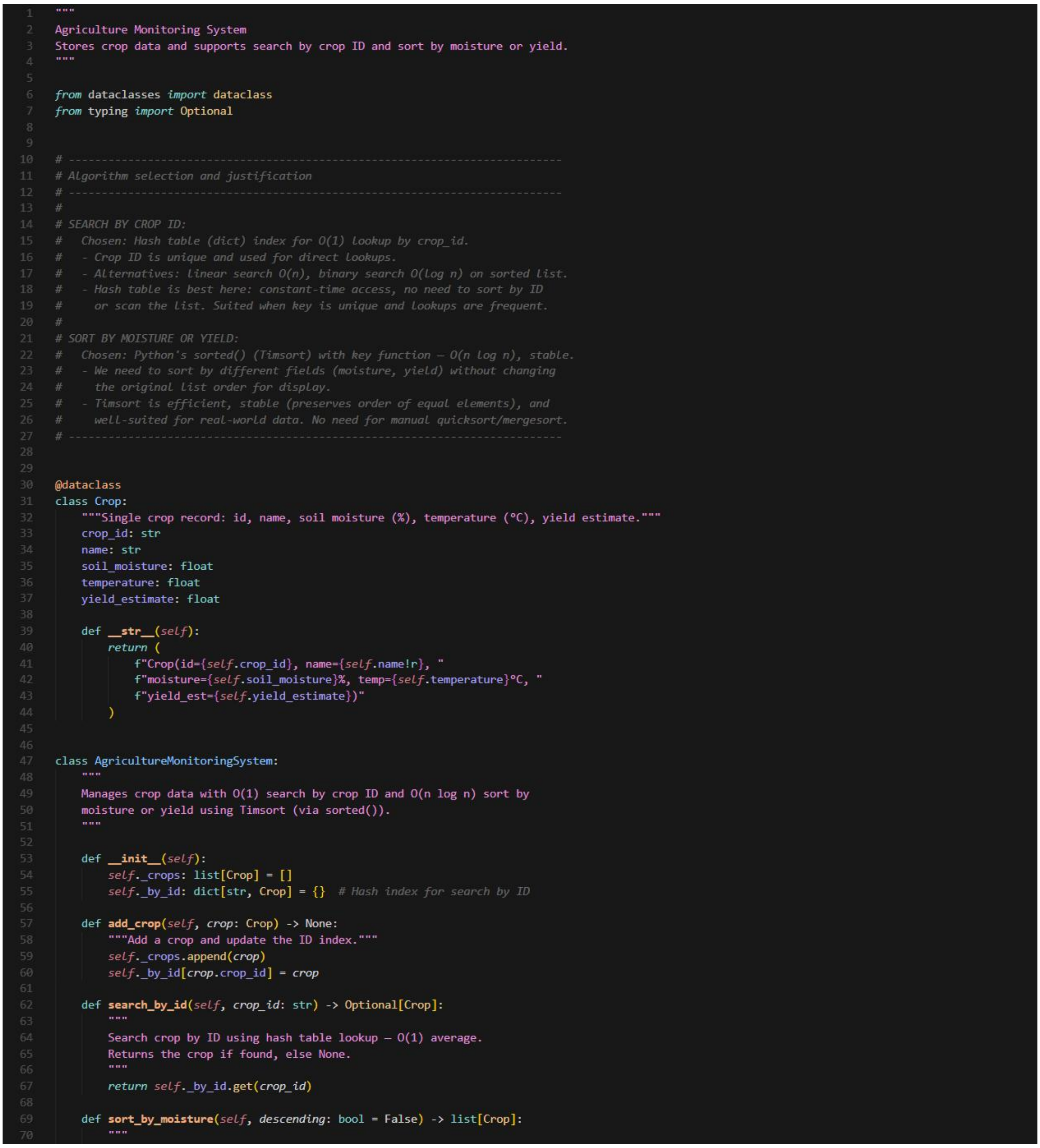
Output:



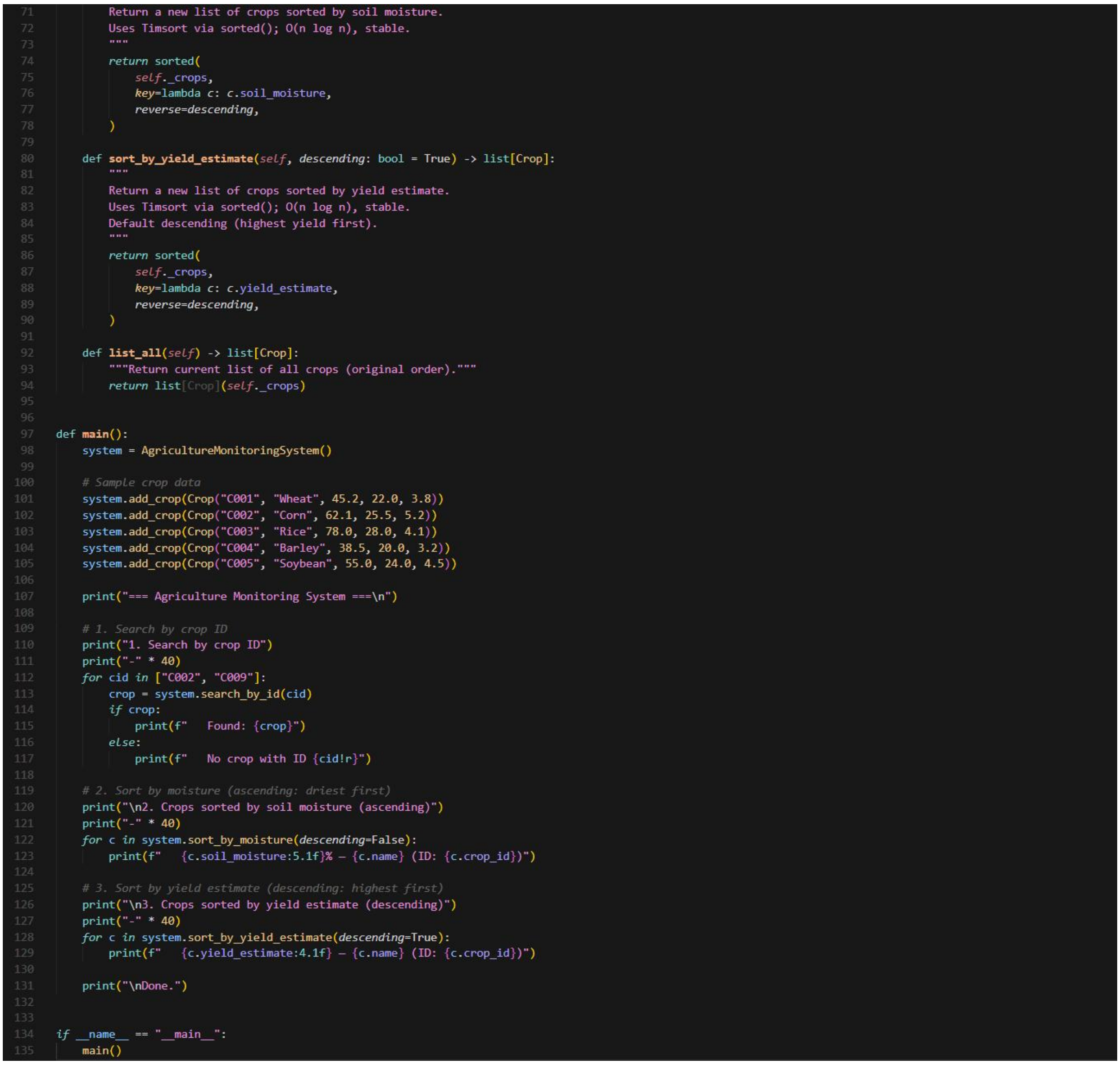
```
Sorted by ID: [101, 102, 103, 104, 105]
Search ID 102: The Dark Knight
Search ID 99: None
By rating (high first): [('The Shawshank Redemption', 9.3), ('The Dark Knight', 9.0), ('Pulp Fiction', 8.9), ('Inception', 8.8), ('Forrest Gump', 8.8)]
By year (old first): [('The Shawshank Redemption', 1994), ('Pulp Fiction', 1994), ('Forrest Gump', 1994), ('The Dark Knight', 2008), ('Inception', 2010)]
```

Task Description #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:
1. Search crop details using crop ID.
2. Sort crops based on moisture level or yield estimate.
Tasks
- use reasoning to select algorithms.
- Justify algorithm suitability.
- Implement searching and sorting in Python.

Code:

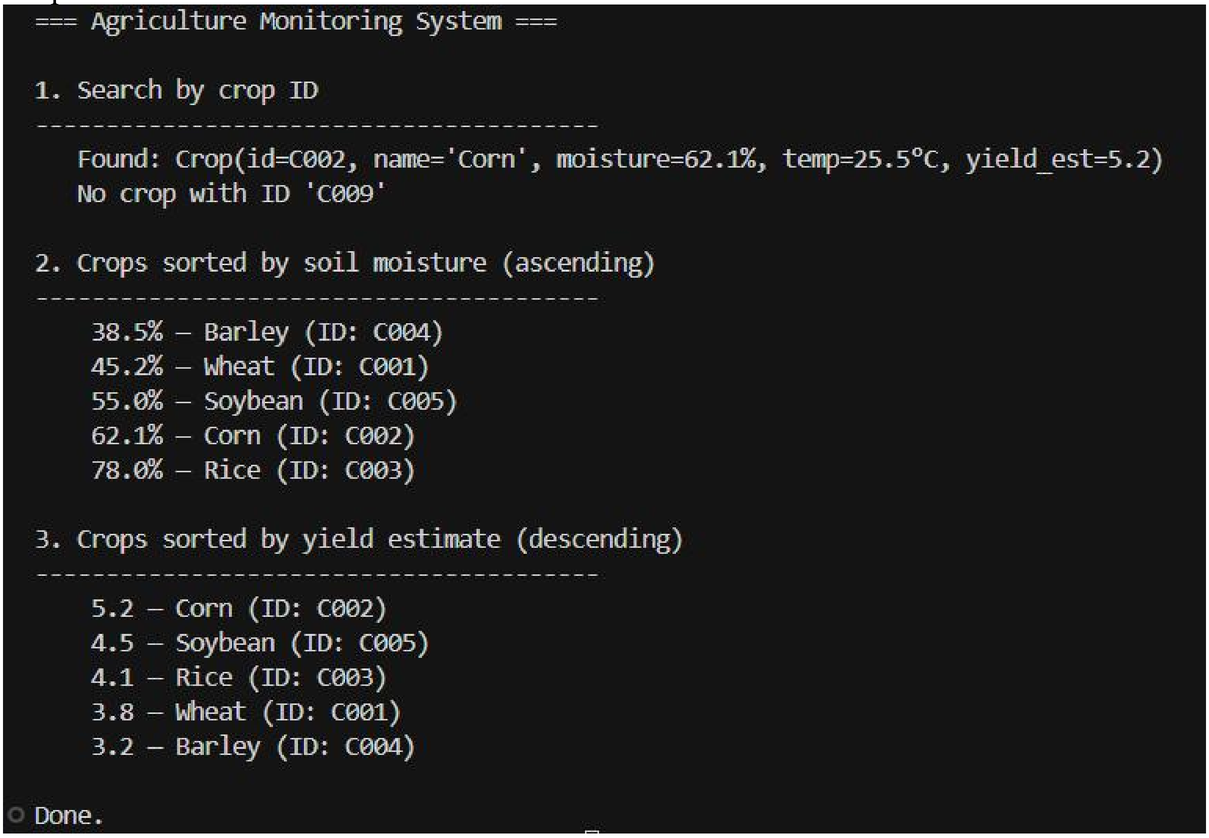


```python
"""
Agriculture Monitoring System
Stores crop data and supports search by crop ID and sort by moisture or yield.
"""

from dataclasses import dataclass
from typing import Optional


# ---------------------------------------------------------------------------
# Algorithm selection and justification
# ---------------------------------------------------------------------------
#
# SEARCH BY CROP ID:
#   Chosen: Hash table (dict) index for O(1) lookup by crop_id.
#   - Crop ID is unique and used for direct lookups.
#   - Alternatives: linear search O(n), binary search O(log n) on sorted list.
#   - Hash table is best here: constant-time access, no need to sort by ID
#     or scan the list. Suited when key is unique and lookups are frequent.
#
# SORT BY MOISTURE OR YIELD:
#   Chosen: Python's sorted() (Timsort) with key function – O(n log n), stable.
#   - We need to sort by different fields (moisture, yield) without changing
#     the original list order for display.
#   - Timsort is efficient, stable (preserves order of equal elements), and
#     well-suited for real-world data. No need for manual quicksort/mergesort.
# ---------------------------------------------------------------------------


@dataclass
class Crop:
    """Single crop record: id, name, soil moisture (%), temperature (°C), yield estimate."""
    crop_id: str
    name: str
    soil_moisture: float
    temperature: float
    yield_estimate: float

    def __str__(self):
        return (
            f"Crop(id={self.crop_id}, name={self.name!r}, "
            f"moisture={self.soil_moisture}%, temp={self.temperature}°C, "
            f"yield_est={self.yield_estimate})"
        )


class AgricultureMonitoringSystem:
    """
    Manages crop data with O(1) search by crop ID and O(n log n) sort by
    moisture or yield using Timsort (via sorted()).
    """

    def __init__(self):
        self._crops: list[Crop] = []
        self._by_id: dict[str, Crop] = {}  # Hash index for search by ID

    def add_crop(self, crop: Crop) -> None:
        """Add a crop and update the ID index."""
        self._crops.append(crop)
        self._by_id[crop.crop_id] = crop

    def search_by_id(self, crop_id: str) -> Optional[Crop]:
        """
        Search crop by ID using hash table lookup – O(1) average.
        Returns the crop if found, else None.
        """
        return self._by_id.get(crop_id)

    def sort_by_moisture(self, descending: bool = False) -> list[Crop]:
        """
```



```
        Return a new list of crops sorted by soil moisture.
        Uses Timsort via sorted(); O(n log n), stable.
        """
        return sorted(
            self._crops,
            key=lambda c: c.soil_moisture,
            reverse=descending,
        )

    def sort_by_yield_estimate(self, descending: bool = True) -> list[Crop]:
        """
        Return a new list of crops sorted by yield estimate.
        Uses Timsort via sorted(); O(n log n), stable.
        Default descending (highest yield first).
        """
        return sorted(
            self._crops,
            key=lambda c: c.yield_estimate,
            reverse=descending,
        )

    def list_all(self) -> list[Crop]:
        """Return current list of all crops (original order)."""
        return list[Crop](self._crops)


def main():
    system = AgricultureMonitoringSystem()

    # Sample crop data
    system.add_crop(Crop("C001", "Wheat", 45.2, 22.0, 3.8))
    system.add_crop(Crop("C002", "Corn", 62.1, 25.5, 5.2))
    system.add_crop(Crop("C003", "Rice", 78.0, 28.0, 4.1))
    system.add_crop(Crop("C004", "Barley", 38.5, 20.0, 3.2))
    system.add_crop(Crop("C005", "Soybean", 55.0, 24.0, 4.5))

    print("=== Agriculture Monitoring System ===\n")

    # 1. Search by crop ID
    print("1. Search by crop ID")
    print("-" * 40)
    for cid in ["C002", "C009"]:
        crop = system.search_by_id(cid)
        if crop:
            print(f"   Found: {crop}")
        else:
            print(f"   No crop with ID {cid!r}")

    # 2. Sort by moisture (ascending: driest first)
    print("\n2. Crops sorted by soil moisture (ascending)")
    print("-" * 40)
    for c in system.sort_by_moisture(descending=False):
        print(f"   {c.soil_moisture:5.1f}% – {c.name} (ID: {c.crop_id})")

    # 3. Sort by yield estimate (descending: highest first)
    print("\n3. Crops sorted by yield estimate (descending)")
    print("-" * 40)
    for c in system.sort_by_yield_estimate(descending=True):
        print(f"   {c.yield_estimate:4.1f} – {c.name} (ID: {c.crop_id})")

    print("\nDone.")


if __name__ == "__main__":
    main()
```

Output:



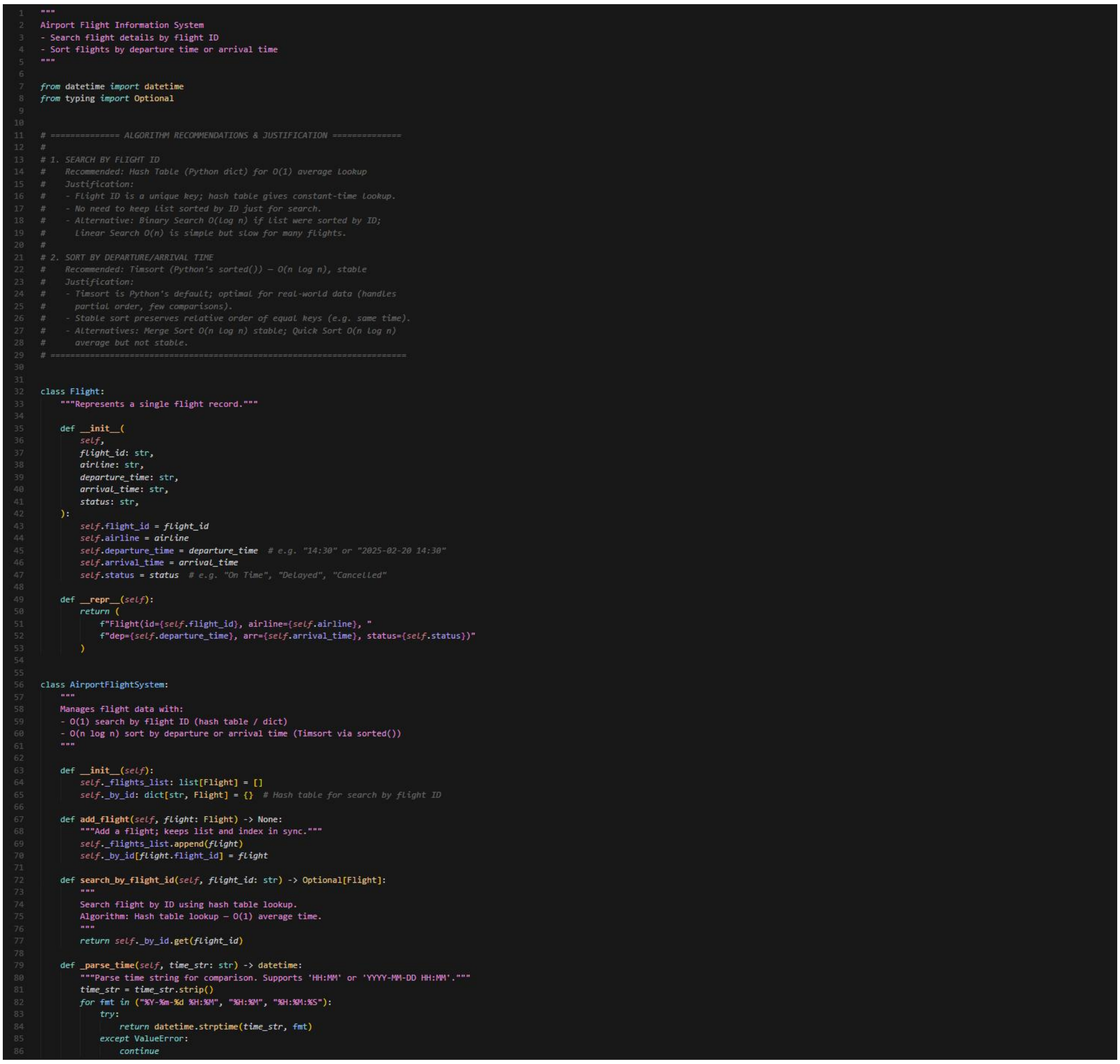
```
=== Agriculture Monitoring System ===

1. Search by crop ID
----------------------------------------
   Found: Crop(id=C002, name='Corn', moisture=62.1%, temp=25.5°C, yield_est=5.2)
   No crop with ID 'C009'

2. Crops sorted by soil moisture (ascending)
----------------------------------------
    38.5% – Barley (ID: C004)
    45.2% – Wheat (ID: C001)
    55.0% – Soybean (ID: C005)
    62.1% – Corn (ID: C002)
    78.0% – Rice (ID: C003)

3. Crops sorted by yield estimate (descending)
----------------------------------------
    5.2 – Corn (ID: C002)
    4.5 – Soybean (ID: C005)
    4.1 – Rice (ID: C003)
    3.8 – Wheat (ID: C001)
    3.2 – Barley (ID: C004)

○ Done.
```

Task Description #8: Airport Flight Management System

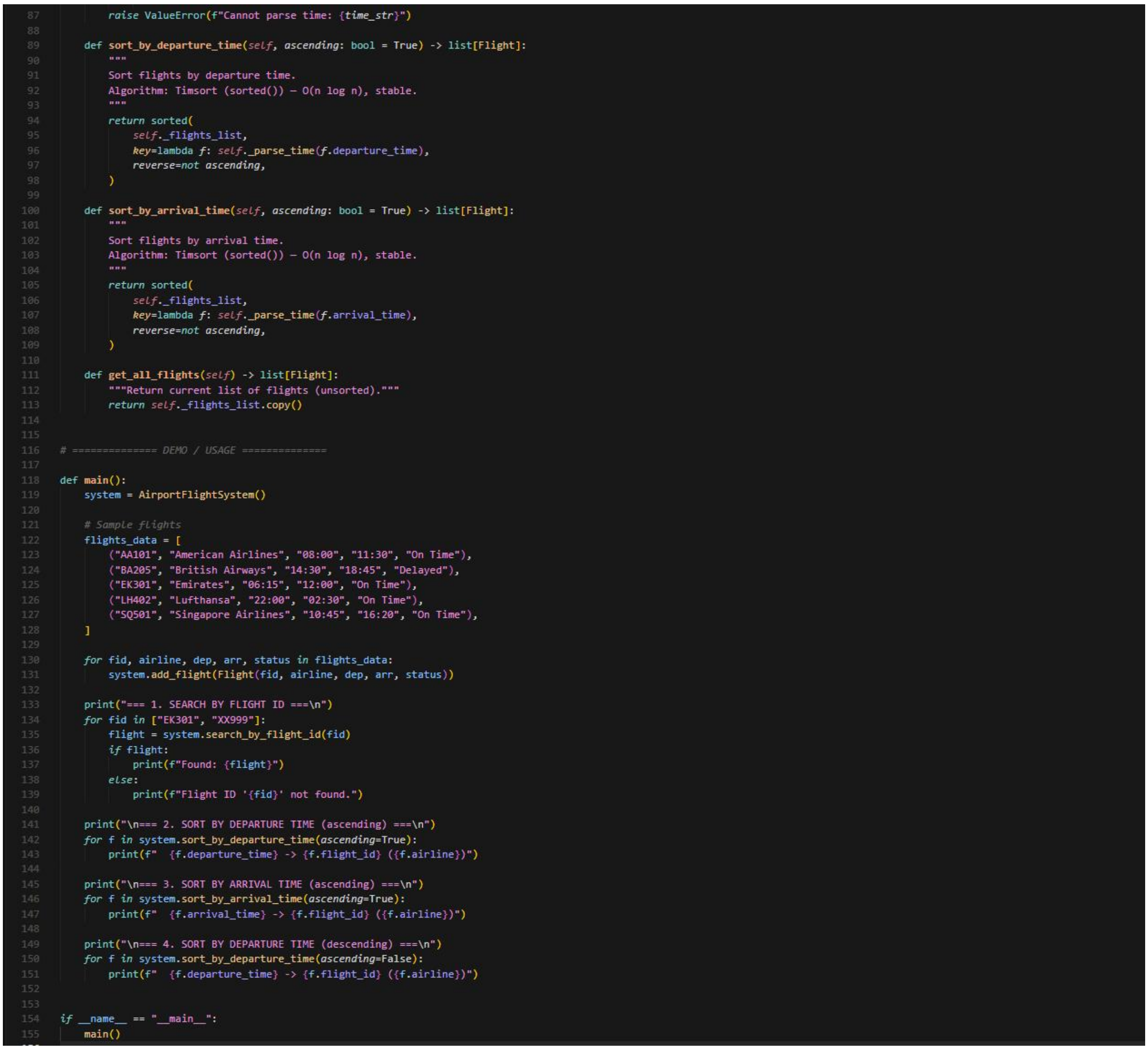
Prompt:
An airport system stores flight information including flight ID, airline name, departure time, arrival time, and status. The system must:
1. Search flight details using flight ID.
2. Sort flights based on departure time or arrival time.

Tasks
- recommend algorithms.
- Justify the algorithm selection.
- Implement searching and sorting logic in Python.

Code:

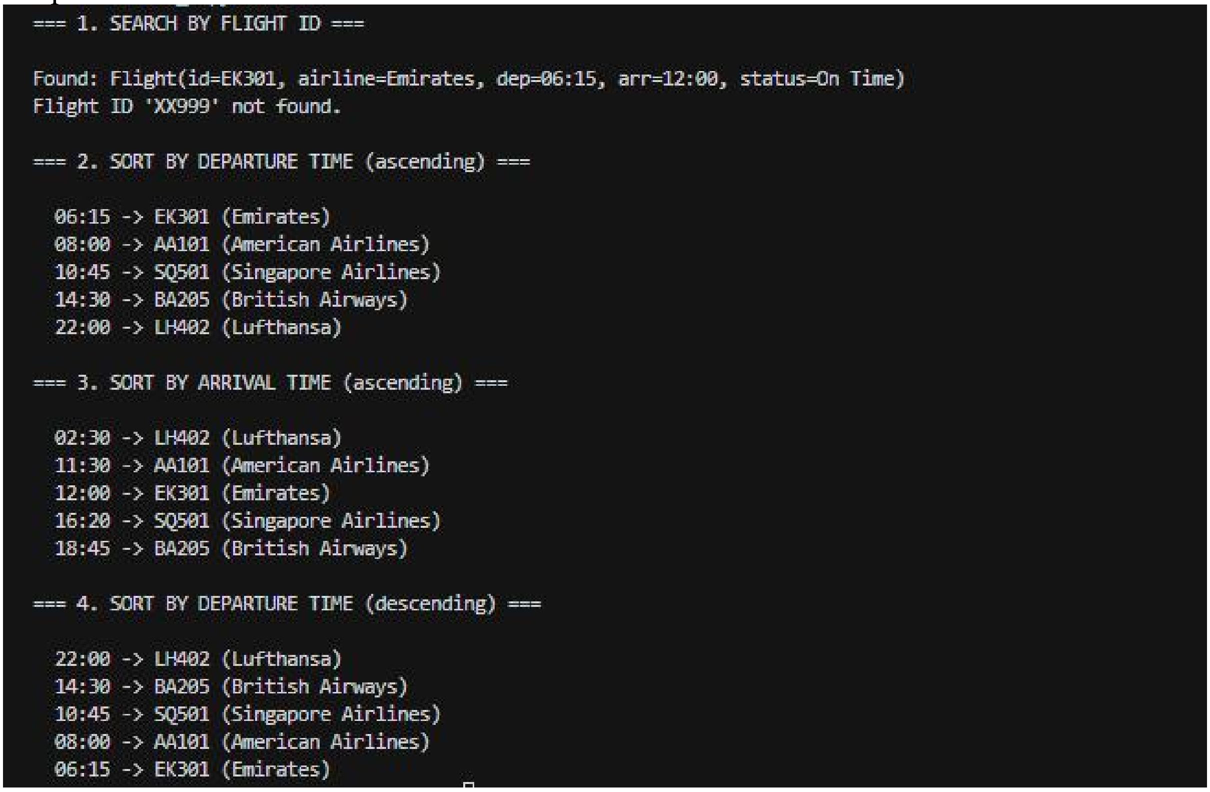


```python
"""
Airport Flight Information System
- Search flight details by flight ID
- Sort flights by departure time or arrival time
"""

from datetime import datetime
from typing import Optional


# ============== ALGORITHM RECOMMENDATIONS & JUSTIFICATION ==============
#
# 1. SEARCH BY FLIGHT ID
#    Recommended: Hash Table (Python dict) for O(1) average lookup
#    Justification:
#    - Flight ID is a unique key; hash table gives constant-time lookup.
#    - No need to keep list sorted by ID just for search.
#    - Alternative: Binary Search O(log n) if list were sorted by ID;
#      Linear Search O(n) is simple but slow for many flights.
#
# 2. SORT BY DEPARTURE/ARRIVAL TIME
#    Recommended: Timsort (Python's sorted()) — O(n log n), stable
#    Justification:
#    - Timsort is Python's default; optimal for real-world data (handles
#      partial order, few comparisons).
#    - Stable sort preserves relative order of equal keys (e.g. same time).
#    - Alternatives: Merge Sort O(n log n) stable; Quick Sort O(n log n)
#      average but not stable.
# ========================================================================


class Flight:
    """Represents a single flight record."""

    def __init__(
        self,
        flight_id: str,
        airline: str,
        departure_time: str,
        arrival_time: str,
        status: str,
    ):
        self.flight_id = flight_id
        self.airline = airline
        self.departure_time = departure_time  # e.g. "14:30" or "2025-02-20 14:30"
        self.arrival_time = arrival_time
        self.status = status  # e.g. "On Time", "Delayed", "Cancelled"

    def __repr__(self):
        return (
            f"Flight(id={self.flight_id}, airline={self.airline}, "
            f"dep={self.departure_time}, arr={self.arrival_time}, status={self.status})"
        )


class AirportFlightSystem:
    """
    Manages flight data with:
    - O(1) search by flight ID (hash table / dict)
    - O(n log n) sort by departure or arrival time (Timsort via sorted())
    """

    def __init__(self):
        self._flights_list: list[Flight] = []
        self._by_id: dict[str, Flight] = {}  # Hash table for search by flight ID

    def add_flight(self, flight: Flight) -> None:
        """Add a flight; keeps list and index in sync."""
        self._flights_list.append(flight)
        self._by_id[flight.flight_id] = flight

    def search_by_flight_id(self, flight_id: str) -> Optional[Flight]:
        """
        Search flight by ID using hash table lookup.
        Algorithm: Hash table lookup — O(1) average time.
        """
        return self._by_id.get(flight_id)

    def _parse_time(self, time_str: str) -> datetime:
        """Parse time string for comparison. Supports 'HH:MM' or 'YYYY-MM-DD HH:MM'."""
        time_str = time_str.strip()
        for fmt in ("%Y-%m-%d %H:%M", "%H:%M", "%H:%M:%S"):
            try:
                return datetime.strptime(time_str, fmt)
            except ValueError:
                continue
```



```python
            raise ValueError(f"Cannot parse time: {time_str}")

    def sort_by_departure_time(self, ascending: bool = True) -> list[Flight]:
        """
        Sort flights by departure time.
        Algorithm: Timsort (sorted()) — O(n log n), stable.
        """
        return sorted(
            self._flights_list,
            key=lambda f: self._parse_time(f.departure_time),
            reverse=not ascending,
        )

    def sort_by_arrival_time(self, ascending: bool = True) -> list[Flight]:
        """
        Sort flights by arrival time.
        Algorithm: Timsort (sorted()) — O(n log n), stable.
        """
        return sorted(
            self._flights_list,
            key=lambda f: self._parse_time(f.arrival_time),
            reverse=not ascending,
        )

    def get_all_flights(self) -> list[Flight]:
        """Return current list of flights (unsorted)."""
        return self._flights_list.copy()


# ============== DEMO / USAGE ==============

def main():
    system = AirportFlightSystem()

    # Sample flights
    flights_data = [
        ("AA101", "American Airlines", "08:00", "11:30", "On Time"),
        ("BA205", "British Airways", "14:30", "18:45", "Delayed"),
        ("EK301", "Emirates", "06:15", "12:00", "On Time"),
        ("LH402", "Lufthansa", "22:00", "02:30", "On Time"),
        ("SQ501", "Singapore Airlines", "10:45", "16:20", "On Time"),
    ]

    for fid, airline, dep, arr, status in flights_data:
        system.add_flight(Flight(fid, airline, dep, arr, status))

    print("=== 1. SEARCH BY FLIGHT ID ===\n")
    for fid in ["EK301", "XX999"]:
        flight = system.search_by_flight_id(fid)
        if flight:
            print(f"Found: {flight}")
        else:
            print(f"Flight ID '{fid}' not found.")

    print("\n=== 2. SORT BY DEPARTURE TIME (ascending) ===\n")
    for f in system.sort_by_departure_time(ascending=True):
        print(f"  {f.departure_time} -> {f.flight_id} ({f.airline})")

    print("\n=== 3. SORT BY ARRIVAL TIME (ascending) ===\n")
    for f in system.sort_by_arrival_time(ascending=True):
        print(f"  {f.arrival_time} -> {f.flight_id} ({f.airline})")

    print("\n=== 4. SORT BY DEPARTURE TIME (descending) ===\n")
    for f in system.sort_by_departure_time(ascending=False):
        print(f"  {f.departure_time} -> {f.flight_id} ({f.airline})")


if __name__ == "__main__":
    main()
```

Output:



```
=== 1. SEARCH BY FLIGHT ID ===

Found: Flight(id=EK301, airline=Emirates, dep=06:15, arr=12:00, status=On Time)
Flight ID 'XX999' not found.

=== 2. SORT BY DEPARTURE TIME (ascending) ===

  06:15 -> EK301 (Emirates)
  08:00 -> AA101 (American Airlines)
  10:45 -> SQ501 (Singapore Airlines)
  14:30 -> BA205 (British Airways)
  22:00 -> LH402 (Lufthansa)

=== 3. SORT BY ARRIVAL TIME (ascending) ===

  02:30 -> LH402 (Lufthansa)
  11:30 -> AA101 (American Airlines)
  12:00 -> EK301 (Emirates)
  16:20 -> SQ501 (Singapore Airlines)
  18:45 -> BA205 (British Airways)

=== 4. SORT BY DEPARTURE TIME (descending) ===

  22:00 -> LH402 (Lufthansa)
  14:30 -> BA205 (British Airways)
  10:45 -> SQ501 (Singapore Airlines)
  08:00 -> AA101 (American Airlines)
  06:15 -> EK301 (Emirates)
```