

---

**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** : R.Sai Kowshik  
**Enrollment No.** : 2403A51L23  
**Batch No** : 51  
**Date** : 20/02/26

---

**Task-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:**

```
1 def merge_sort(arr):
2     """
3     Sort a list in ascending order using the merge sort algorithm.
4
5     Time Complexity: O(n log n) - divides the list in half at each level (log n levels)
6     and merges n elements at each level.
7     Space Complexity: O(n) - additional temporary arrays used during merging;
8     recursion stack depth is O(log n) but dominant space is the
9     auxiliary arrays of size proportional to n.
10    """
11    if len(arr) <= 1:
12        return arr
13
14    mid = len(arr) // 2
15    left = merge_sort(arr[:mid])
16    right = merge_sort(arr[mid:])
17
18    return _merge(left, right)
19
20
21 def _merge(left, right):
22     """Merge two sorted lists into one sorted list."""
23     result = []
24     i = j = 0
25
26     while i < len(left) and j < len(right):
27         if left[i] <= right[j]:
28             result.append(left[i])
29             i += 1
30         else:
31             result.append(right[j])
32             j += 1
33
34     result.extend(left[i:])
35     result.extend(right[j:])
36     return result
37
38
39 if __name__ == "__main__":
40     # Test cases
41     test_cases = [
42         ([64, 34, 25, 12, 22, 11, 90], [11, 12, 22, 25, 34, 64, 90]),
43         ([5, 2, 8, 1, 9], [1, 2, 5, 8, 9]),
44         ([1], [1]),
45         ([], []),
46         ([3, 3, 3, 3], [3, 3, 3, 3]),
47         ([2, 1], [1, 2]),
48         ([5, 4, 3, 2, 1], [1, 2, 3, 4, 5]),
49     ]
50
51     passed = 0
52     for arr, expected in test_cases:
53         result = merge_sort(arr.copy())
54         status = "PASS" if result == expected else "FAIL"
55         if status == "PASS":
56             passed += 1
57         print(f"({status}): merge_sort({arr}) -> {result} (expected {expected})")
58
59     print(f"({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([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-2 (Searching – Binary Search with AI Optimization):

**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:

```
1 def binary_search(arr, target):
2     """
3     Time Complexity
4     -----
5     - Best case: O(1)
6       When the target is found at the middle position on the first comparison.
7     - Average case: O(log n)
8       On average, the search space is halved repeatedly until the target is
9       found or the search space is empty.
10    - Worst case: O(log n)
11      When the target is not present or is only found after reducing the
12      search space to size 1.
13
14    Space Complexity
15    -----
16    - O(1)
17      The algorithm uses a constant amount of extra space (iterative version).
18    """
19    left, right = 0, len(arr) - 1
20
21    while left <= right:
22        mid = (left + right) // 2
23        if arr[mid] == target:
24            return mid
25        elif arr[mid] < target:
26            left = mid + 1
27        else:
28            right = mid - 1
29
30    return -1
31
32
33 if __name__ == "__main__":
34     # Basic tests with integers
35     nums = [1, 3, 5, 7, 9, 11]
36     print("Searching in:", nums)
37     print("Target 1 -> index:", binary_search(nums, 1)) # best-case: first mid
38     print("Target 7 -> index:", binary_search(nums, 7)) # typical middle element
39     print("Target 11 -> index:", binary_search(nums, 11)) # last element
40     print("Target 4 -> index:", binary_search(nums, 4)) # not in list (should be -1)
41
42     # Tests with duplicates (binary search returns one valid index if present)
43     dup_nums = [2, 4, 4, 4, 6, 8]
44     print("\nSearching in (with duplicates):", dup_nums)
45     print("Target 4 -> index:", binary_search(dup_nums, 4))
46
47     # Tests with strings
48     words = ["apple", "banana", "cherry", "date", "fig", "grape"]
49     print("\nSearching in:", words)
50     print("Target 'cherry' -> index:", binary_search(words, "cherry"))
51     print("Target 'kiwi' -> index:", binary_search(words, "kiwi")) # not found
52
53     # Edge cases
54     empty = []
55     one_elem = [10]
56     print("\nEdge cases:")
57     print("Empty list, target 1 -> index:", binary_search(empty, 1))
58     print("Single element, target 10 -> index:", binary_search(one_elem, 10))
59     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: ['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 'kiwi' -> 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 -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:

```
1 """
2 Healthcare Appointment System - Search and Sort Algorithms
3 =====
4 Appointment records: appointment ID, patient name, doctor name,
5 appointment time, consultation fee.
6
7 RECOMMENDATIONS & JUSTIFICATION:
8 =====
9 1. SEARCH BY APPOINTMENT ID: Binary Search
10 - Use when the list is sorted by appointment ID (or we maintain a sorted copy).
11 - Time: O(log n), Space: O(1) for iterative version.
12 - Justification: Appointment ID is a unique key; sorting by ID once allows
13   fast repeated lookups. Better than linear search O(n) when n is large.
14 - Alternative: Hash table (dict) for O(1) lookup if ID is the primary key.
15
16 2. SORT BY TIME OR FEE: Merge Sort
17 - Time: O(n log n) guaranteed; Space: O(n) for auxiliary array.
18 - Justification: Stable sort (preserves order of equal elements), predictable
19   performance, suitable for small to medium lists. Preferred when stability
20   matters (e.g., "same fee" order preserved by time).
21 """
22
23 from dataclasses import dataclass
24 from typing import Callable, Optional
25
26 @dataclass
27 class Appointment:
28     """Single appointment record."""
29     appointment_id: int
30     patient_name: str
31     doctor_name: str
32     appointment_time: str # e.g., "2025-02-20 10:30"
33     consultation_fee: float
34
35     def __repr__(self):
36         return (
37             f"Appointment(id={self.appointment_id}, patient='{self.patient_name}', "
38             f"doctor='{self.doctor_name}', time='{self.appointment_time}', fee={self.consultation_fee})"
39         )
40
41 # =====
42 # 1. SEARCH BY APPOINTMENT ID (Binary Search)
43 # =====
44 # Precondition: appointments must be sorted by appointment_id (ascending).
45
46 def binary_search_by_id(
47     appointments: list[Appointment],
48     target_id: int,
49     ) -> Optional[Appointment]:
50     """
51     Search for an appointment by ID using Binary Search.
52     Returns the Appointment if found, else None.
53     """
54     if not appointments:
55         return None
56     # Binary search is on a list sorted by appointment_id
57     sorted_by_id = sorted(appointments, key=lambda a: a.appointment_id)
58     left, right = 0, len(sorted_by_id) - 1
59     while left <= right:
60         mid = (left + right) // 2
61         a = sorted_by_id[mid]
62         if a.appointment_id == target_id:
63             return a
64         elif a.appointment_id < target_id:
65             left = mid + 1
66         else:
67             right = mid - 1
68     return None
69
70 # =====
71 # 2. SORT APPOINTMENTS (Merge Sort)
72 # =====
73 # Sort by a numeric or comparable key (e.g. time string, fee).
74 # Uses a key function: by_time or by_fee.
75
76 def merge_sort_appointments(
77     appointments: list[Appointment],
78     *,
79     by_time: bool = True,
80     ) -> list[Appointment]:
81     """
82     Sort appointments using Merge Sort.
83     by_time=True -> sort by appointment_time (string comparison).
84     """
```

```

87 by_time=False -> sort by consultation_fee (numeric).
88 Returns a new sorted list; does not modify the original.
89 Uses (key, appointment) pairs so the same merge logic works for any comparable key.
90 ---
91 if not appointments:
92     return []
93 key_fn: Callable[[Appointment], str | float] = (
94     lambda a: a.appointment_time if by_time else (lambda a: a.consultation_fee)
95 )
96 keyed = [(key_fn(a), a) for a in appointments]
97 n = len(keyed)
98 temp: list[Optional[tuple]] = [None] * n
99
100 def merge_keyed(left: int, mid: int, right: int) -> None:
101     i, j, k = left, mid + 1, left
102     while i <= mid and j <= right:
103         if keyed[i][0] <= keyed[j][0]:
104             temp[k] = keyed[i]
105             i += 1
106         else:
107             temp[k] = keyed[j]
108             j += 1
109         k += 1
110     while i <= mid:
111         temp[k] = keyed[i]
112         i += 1
113         k += 1
114     while j <= right:
115         temp[k] = keyed[j]
116         j += 1
117         k += 1
118     for idx in range(left, right + 1):
119         keyed[idx] = temp[idx]
120
121 def merge_sort_keyed(l: int, r: int) -> None:
122     if l >= r:
123         return
124     m = (l + r) // 2
125     merge_sort_keyed(l, m)
126     merge_sort_keyed(m + 1, r)
127     merge_keyed(l, m, r)
128
129 merge_sort_keyed(0, n - 1)
130 return [keyed[i][1] for i in range(n)]
131
132 # ---
133 # Alternative Merge sort with a single key (no tuple) for clarity
134 # ---
135
136 def sort_appointments_by_time(appointments: list[Appointment]) -> list[Appointment]:
137     """Sort appointments by appointment time using Merge Sort."""
138     return merge_sort_appointments(appointments, by_time=True)
139
140
141 def sort_appointments_by_fee(appointments: list[Appointment]) -> list[Appointment]:
142     """Sort appointments by consultation fee using Merge Sort."""
143     return merge_sort_appointments(appointments, by_time=False)
144
145 # ---
146 # Demo
147 # ---
148 def main():
149     # Sample appointment records
150     appointments = [
151         Appointment(105, "Alice Chen", "Dr. Smith", "2025-02-20 14:00", 1500.0),
152         Appointment(102, "Bob Kim", "Dr. Jones", "2025-02-20 09:30", 2000.0),
153         Appointment(108, "Carol Lee", "Dr. Smith", "2025-02-20 11:00", 1500.0),
154         Appointment(101, "David Park", "Dr. Brown", "2025-02-20 08:00", 1200.0),
155         Appointment(110, "Eve Wong", "Dr. Jones", "2025-02-20 16:30", 2500.0),
156     ]
157
158     print("=== Healthcare Appointment System ===\n")
159     print("Original list (unsorted):")
160     for a in appointments:
161         print(f"  {a}")
162     print()
163
164     # 1. Search by appointment ID
165     print("--- Search by Appointment ID (Binary Search) ---")
166     for target in [102, 109]:
167         result = binary_search_by_id(appointments, target)
168         print(f"  ID {target}: {result if result else 'Not found'}")
169     print()
170
171     # 2. Sort by time
172     print("--- Sorted by Appointment Time (Merge Sort) ---")
173     by_time = sort_appointments_by_time(appointments)
174     for a in by_time:
175         print(f"  {a}")
176     print()
177
178     # 3. Sort by consultation fee
179     print("--- Sorted by Consultation Fee (Merge Sort) ---")
180     by_fee = sort_appointments_by_fee(appointments)
181     for a in by_fee:
182         print(f"  {a}")
183
184 if __name__ == "__main__":
185     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-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:**

```

1  """
2  Railway Reservation System
3  Implements efficient search by ticket ID and sort by travel date or seat number.
4  """
5
6  from dataclasses import dataclass
7  from datetime import date
8  from typing import Optional
9
10
11 @dataclass
12 class Booking:
13     """Represents a single railway booking."""
14     ticket_id: str
15     passenger_name: str
16     train_number: str
17     seat_number: str
18     travel_date: date
19
20     def __str__(self):
21         return (
22             f"Ticket: {self.ticket_id} | {self.passenger_name} | "
23             f"Train: {self.train_number} | Seat: {self.seat_number} | "
24             f"Date: {self.travel_date}"
25         )
26
27
28 class RailwayReservationSystem:
29     """
30     Railway reservation system with:
31     - O(1) average-case search by ticket ID (hash table)
32     - O(n log n) sort by travel date or seat number (Timsort via sorted())
33     """
34
35     def __init__(self):
36         # List for ordered iteration and sorting
37         self._bookings: list[Booking] = []
38         # Hash table for O(1) lookup by ticket_id (algorithm choice: see search_ticket)
39         self._by_ticket_id: dict[str, Booking] = {}
40
41     def add_booking(self, booking: Booking) -> None:
42         """Add a booking. Duplicate ticket_id overwrites previous."""
43         self._by_ticket_id[booking.ticket_id] = booking
44         # Keep list in sync: remove old if same id, then append
45         self._bookings = [b for b in self._bookings if b.ticket_id != booking.ticket_id]
46         self._bookings.append(booking)
47
48     def search_ticket(self, ticket_id: str) -> Optional[Booking]:
49         """
50         Search by ticket ID.
51         Algorithm: Hash table (dict) lookup.
52         Justification: O(1) average-case lookup; ideal when the key is unique
53         (ticket ID). Binary search on sorted list would be O(log n) but requires
54         keeping the list sorted by ticket_id and O(log n) or O(n) insertions.
55         For "search by ID" as primary operation, hash table is the standard choice.
56         """
57         return self._by_ticket_id.get(ticket_id)
58
59     def sort_by_travel_date(self) -> list[Booking]:
60         """
61         Sort bookings by travel date (ascending).
62         Algorithm: Timsort (Python's sorted()).
63         Justification: O(n log n), stable sort. Stable means equal dates
64         keep their relative order, which is good for consistent display.
65         """
66         return sorted(self._bookings, key=lambda b: b.travel_date)
67
68     def sort_by_seat_number(self) -> list[Booking]:

```

```

69     """
70     Sort bookings by seat number (ascending).
71     Algorithm: Timsort (Python's sorted()).
72     Justification: Same as above; we use key for natural ordering.
73     Seat numbers are compared as strings; for numeric ordering use a key
74     that parses to int if your format is purely numeric.
75     """
76     return sorted(self._bookings, key=lambda b: (b.seat_number, b.travel_date))
77
78     def get_all_bookings(self) -> list[Booking]:
79         """Return current list of bookings (unsorted)."""
80         return self._bookings.copy()
81
82
83 def main():
84     from datetime import date
85
86     system = RailwayReservationSystem()
87
88     # Sample bookings
89     system.add_booking(Booking("T001", "Alice", "TR-101", "A1", date(2025, 3, 15)))
90     system.add_booking(Booking("T002", "Bob", "TR-102", "B3", date(2025, 3, 10)))
91     system.add_booking(Booking("T003", "Carol", "TR-101", "A2", date(2025, 3, 20)))
92     system.add_booking(Booking("T004", "Dave", "TR-103", "C1", date(2025, 3, 12)))
93
94     print("=== Search by ticket ID ===")
95     for tid in ["T002", "T999"]:
96         b = system.search_ticket(tid)
97         print(f" {tid}: {b if b else 'Not found'}")
98
99     print("\n=== Sort by travel date ===")
100    for b in system.sort_by_travel_date():
101        print(f" {b}")
102
103    print("\n=== Sort by seat number ===")
104    for b in system.sort_by_seat_number():
105        print(f" {b}")
106
107
108 if __name__ == "__main__":
109     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-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:

```

1  """
2  Hostel Management System - Lab 2.2
3  Stores student room allocation details with optimized search and sort operations.
4  """
5
6  from dataclasses import dataclass
7  from datetime import date
8  from typing import Optional
9
10
11 @dataclass
12 class AllocationRecord:
13     """Single room allocation record."""
14     student_id: str
15     room_number: int
16     floor: int
17     allocation_date: date
18
19     def __str__(self):
20         return (
21             f"Student ID: {self.student_id}, Room: {self.room_number}, "
22             f"Floor: {self.floor}, Date: {self.allocation_date}"
23         )
24
25
26 class HostelManager:
27     """
28     Hostel allocation manager with:
29     - O(1) average search by student ID (hash table)
30     - O(n log n) sort by room number or allocation date (Timsort)
31     """
32
33     def __init__(self):
34         # Hash Table: student_id -> AllocationRecord
35         # Chosen for O(1) average lookup when searching by student ID
36         self._by_student_id: dict[str, AllocationRecord] = {}
37         # List of all records (references same objects) for sorting
38         self._records: list[AllocationRecord] = []
39
40     def add_allocation(
41         self,
42         student_id: str,
43         room_number: int,
44         floor: int,
45         allocation_date: date,
46     ) -> bool:
47         """Add a new allocation. Returns False if student_id already allocated.
48         If student_id in self._by_student_id:
49             return False
50         record = AllocationRecord(
51             student_id=student_id,
52             room_number=room_number,
53             floor=floor,
54             allocation_date=allocation_date,
55         )
56         self._by_student_id[student_id] = record
57         self._records.append(record)
58         return True
59
60     def search_by_student_id(self, student_id: str) -> Optional[AllocationRecord]:
61         """
62         Search allocation by student ID.
63         Algorithm: Hash table lookup.
64         Time: O(1) average, O(n) worst case (rare with good hash).
65         Justification: Student ID is unique; direct key access is optimal.
66         """
67         return self._by_student_id.get(student_id)
68
69     def get_sorted_by_room_number(self) -> list[AllocationRecord]:
70         """
71         Return all records sorted by room number.
72         Algorithm: Timsort (Python's sorted()).
73         Time: O(n log n). Stable sort preserves order for equal keys.
74         Justification: General-purpose, in-place style sort; no need for
75         maintaining a separate sorted structure when sort is on demand.
76         """
77         return sorted(self._records, key=lambda r: (r.room_number, r.floor))
78
79     def get_sorted_by_allocation_date(self) -> list[AllocationRecord]:
80         """
81         Return all records sorted by allocation date (ascending).
82         Algorithm: Timsort (Python's sorted()).
83         Time: O(n log n). Stable.
84         Justification: Same as room sort; date objects are comparable.
85         """
86         return sorted(self._records, key=lambda r: r.allocation_date)
87
88     def list_all(self) -> list[AllocationRecord]:
89         """Return all records in insertion order."""
90         return list[AllocationRecord](self._records)
91
92
93 # ----- Algorithm summary and justification -----
94 """
95 ALGORITHM CHOICES & JUSTIFICATION
96 """
97
98 1. SEARCH BY STUDENT ID
99 - Chosen: Hash table (Python dict) keyed by student_id
100 - Time: O(1) average lookup
101 - Justification:
102     * Each student has at most one current allocation, so student_id is a
103     natural unique key.
104     * Hash table gives constant-time access by key without scanning.
105     * Alternatives rejected:
106         - Linear search O(n): poor for many records.
107         - Binary search O(log n): would require keeping list sorted by
108           student_id and extra bookkeeping when adding/removing.
109
110 2. SORT BY ROOM NUMBER / ALLOCATION DATE
111 - Chosen: Timsort via sorted(Iterable, key=...)
112 - Time: O(n log n), stable
113 - Justification:
114     * We need to sort on different keys (room number, date) on demand.
115     * Timsort is Python's default; it is stable and efficient for
116     partially ordered data.
117     * Keeping two sorted copies (by room and by date) would give O(log n)
118     insertion but double storage and more complex updates; for typical
119     hostel size, sorting when needed is simpler and fast enough.
120
121 """

```



```

122
123 def main():
124     """Demo: add allocations, search by student ID, sort by room and date."""
125     from datetime import date
126
127     manager = HostelManager()
128
129     # Add sample allocations
130     manager.add_allocation("S001", 101, 1, date(2024, 1, 15))
131     manager.add_allocation("S002", 205, 2, date(2024, 2, 1))
132     manager.add_allocation("S003", 102, 1, date(2024, 1, 20))
133     manager.add_allocation("S004", 201, 2, date(2024, 1, 10))
134
135     print("--- Search by Student ID ---")
136     for sid in ["S002", "S999"]:
137         rec = manager.search_by_student_id(sid)
138         print(f" {sid}: {rec if rec else 'Not found'}")
139
140     print("\n--- Sorted by Room Number ---")
141     for r in manager.get_sorted_by_room_number():
142         print(f" {r}")
143
144     print("\n--- Sorted by Allocation Date ---")
145     for r in manager.get_sorted_by_allocation_date():
146         print(f" {r}")
147
148
149 if __name__ == "__main__":
150     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-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:

```

1  """
2  Streaming Service - Movie Records
3  Search by movie ID | Sort by rating or release year
4  """
5
6  from typing import List, Optional, Literal
7
8
9  # -----
10 # Algorithm recommendations & justification
11 # -----
12 """
13 SEARCH BY MOVIE ID:
14 Algorithm: Binary Search
15 Justification:
16 - Movie IDs are unique and comparable → sorted list by ID is well-defined.
17 - Time:  $O(\log n)$  per lookup vs  $O(n)$  for linear search → efficient for large catalogs.
18 - Space:  $O(1)$  extra; no need for a separate hash structure if we keep one list sorted by ID.
19 - Alternative: Hash table (dict) gives  $O(1)$  average lookup; use when ID lookups dominate
    and the list is not already sorted by ID.
20
21 SORT BY RATING OR RELEASE YEAR:
22 Algorithm: Merge Sort
23 Justification:
24 - Stable sort: preserves order of movies with equal rating/year (e.g. by title or ID).
25 - Time:  $O(n \log n)$  in all cases; predictable for streaming-sized datasets.
26 - No worst-case  $O(n^2)$  as in naive Quick Sort; suitable for user-facing ordering.
27 - Works well with linked/record data; easy to sort by different keys (rating, year).
28 """
29
30
31
32 def binary_search_by_id(movies: List[dict], movie_id: int) -> Optional[dict]:
33     """
34     Search for a movie by ID using Binary Search.
35     Precondition: movies is sorted by 'id' (ascending).
36     Time:  $O(\log n)$ , Space:  $O(1)$ .
37     """
38     if not movies:
39         return None
40     left, right = 0, len(movies) - 1
41     while left <= right:
42         mid = (left + right) // 2
43         m = movies[mid]
44         if m["id"] == movie_id:
45             return m
46         if m["id"] < movie_id:
47             left = mid + 1
48         else:
49             right = mid - 1
50     return None
51
52
53 def _merge(
54     arr: List[dict],
55     key: Literal["rating", "year", "id"],
56     ascending: bool,
57     left: int,
58     mid: int,
59     right: int,
60 ) -> None:
61     """Merge two sorted halves arr[left:mid+1] and arr[mid+1:right+1] in-place using temp buffer."""
62     left_copy = arr[left : mid + 1]
63     right_copy = arr[mid + 1 : right + 1]
64     i, j, k = 0, 0, left
65     while i < len(left_copy) and j < len(right_copy):
66         a_val = left_copy[i][key]
67         b_val = right_copy[j][key]
68         if ascending:
69             take_left = a_val <= b_val
70         else:
71             take_left = a_val >= b_val
72         if take_left:
73             arr[k] = left_copy[i]
74             i += 1
75         else:
76             arr[k] = right_copy[j]
77             j += 1
78         k += 1
79     while i < len(left_copy):
80         arr[k] = left_copy[i]
81         i, k = i + 1, k + 1
82     while j < len(right_copy):
83         arr[k] = right_copy[j]
84         j, k = j + 1, k + 1
85

```

```

86 def _merge_sort_range(
87     arr: List[dict],
88     key: Literal["rating", "year", "id"],
89     ascending: bool,
90     left: int,
91     right: int,
92 ) -> None:
93     """Recursive merge sort on arr[left:right+1] by key."""
94     if left >= right:
95         return
96     mid = (left + right) // 2
97     _merge_sort_range(arr, key, ascending, left, mid)
98     _merge_sort_range(arr, key, ascending, mid + 1, right)
99     _merge(arr, key, ascending, left, mid, right)
100
101
102
103 def sort_movies(
104     movies: List[dict],
105     by: Literal["rating", "year"],
106     ascending: bool = True,
107 ) -> List[dict]:
108     """
109     Sort movies by 'rating' or 'year' using Merge Sort (stable, O(n log n)).
110     Returns a new sorted list; does not mutate the original.
111     """
112     if not movies:
113         return []
114     result = [m.copy() for m in movies]
115     _merge_sort_range(result, by, ascending, 0, len(result) - 1)
116     return result
117
118
119 # -----
120 # Helpers: keep list sorted by ID for binary search; build from unsorted list
121 # -----
122 def sort_movies_by_id(movies: List[dict]) -> List[dict]:
123     """Sort by ID so binary_search_by_id can be used. Uses merge sort by 'id'."""
124     return sort_movies_by_key(movies, "id")
125
126
127 def sort_movies_by_key(
128     movies: List[dict],
129     key: Literal["rating", "year", "id"],
130     ascending: bool = True,
131 ) -> List[dict]:
132     """Generic merge sort by key ('id', 'rating', or 'year')."""
133     if not movies:
134         return []
135     result = [m.copy() for m in movies]
136     _merge_sort_range(result, key, ascending, 0, len(result) - 1)
137     return result
138
139
140 # -----
141 # Example usage
142 # -----
143 if __name__ == "__main__":
144     # Sample catalog (unsorted by ID)
145     catalog = [
146         {"id": 103, "title": "Inception", "genre": "Sci-Fi", "rating": 8.8, "year": 2010},
147         {"id": 101, "title": "The Shawshank Redemption", "genre": "Drama", "rating": 9.3, "year": 1994},
148         {"id": 102, "title": "The Dark Knight", "genre": "Action", "rating": 9.0, "year": 2008},
149         {"id": 105, "title": "Pulp Fiction", "genre": "Crime", "rating": 8.9, "year": 1994},
150         {"id": 104, "title": "Forrest Gump", "genre": "Drama", "rating": 8.8, "year": 1994},
151     ]
152
153     # 1) Sort by ID so we can use binary search
154     by_id = sort_movies_by_id(catalog)
155     print("Sorted by ID:", [m["id"] for m in by_id])
156
157     # 2) Search by movie ID
158     movie = binary_search_by_id(by_id, 102)
159     print("Search ID 102:", movie["title"] if movie else None)
160     print("Search ID 99:", binary_search_by_id(by_id, 99))
161
162     # 3) Sort by rating (descending = best first)
163     by_rating = sort_movies(catalog, "rating", ascending=False)
164     print("By rating (high first):", [(m["title"], m["rating"]) for m in by_rating])
165
166     # 4) Sort by release year (ascending)
167     by_year = sort_movies(catalog, "year", ascending=True)
168     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-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:

```

1  """
2  Agriculture Monitoring System
3  Stores crop data and supports search by crop ID and sort by moisture or yield.
4  """
5
6  from dataclasses import dataclass
7  from typing import Optional
8
9
10 # -----
11 # Algorithm selection and justification
12 # -----
13 #
14 # SEARCH BY CROP ID:
15 # Chosen: Hash table (dict) index for O(1) Lookup by crop_id.
16 # - Crop ID is unique and used for direct Lookups.
17 # - Alternatives: Linear search O(n), binary search O(log n) on sorted list.
18 # - Hash table is best here: constant-time access, no need to sort by ID
19 #   or scan the List. Suited when key is unique and Lookups are frequent.
20 #
21 # SORT BY MOISTURE OR YIELD:
22 # Chosen: Python's sorted() (Timsort) with key function - O(n log n), stable.
23 # - We need to sort by different fields (moisture, yield) without changing
24 #   the original List order for display.
25 # - Timsort is efficient, stable (preserves order of equal elements), and
26 #   well-suited for real-world data. No need for manual quicksort/mergesort.
27 # -----
28
29
30 @dataclass
31 class Crop:
32     """Single crop record: id, name, soil moisture (%), temperature (°C), yield estimate."""
33     crop_id: str
34     name: str
35     soil_moisture: float
36     temperature: float
37     yield_estimate: float
38
39     def __str__(self):
40         return (
41             f"Crop(id={self.crop_id}, name={self.name!r}, "
42             f"moisture={self.soil_moisture}%, temp={self.temperature}°C, "
43             f"yield_est={self.yield_estimate})"
44         )
45
46
47 class AgricultureMonitoringSystem:
48     """
49     Manages crop data with O(1) search by crop ID and O(n log n) sort by
50     moisture or yield using Timsort (via sorted()).
51     """
52
53     def __init__(self):
54         self._crops: list[Crop] = []
55         self._by_id: dict[str, Crop] = {} # Hash index for search by ID
56
57     def add_crop(self, crop: Crop) -> None:
58         """Add a crop and update the ID index."""
59         self._crops.append(crop)
60         self._by_id[crop.crop_id] = crop
61
62     def search_by_id(self, crop_id: str) -> Optional[Crop]:
63         """
64         Search crop by ID using hash table lookup - O(1) average.
65         Returns the crop if found, else None.
66         """
67         return self._by_id.get(crop_id)
68
69     def sort_by_moisture(self, descending: bool = False) -> list[Crop]:
70         """

```

```

71     Return a new list of crops sorted by soil moisture.
72     Uses Timsort via sorted(); O(n log n), stable.
73     """
74     return sorted(
75         self._crops,
76         key=lambda c: c.soil_moisture,
77         reverse=descending,
78     )
79
80 def sort_by_yield_estimate(self, descending: bool = True) -> list[Crop]:
81     """
82     Return a new list of crops sorted by yield estimate.
83     Uses Timsort via sorted(); O(n log n), stable.
84     Default descending (highest yield first).
85     """
86     return sorted(
87         self._crops,
88         key=lambda c: c.yield_estimate,
89         reverse=descending,
90     )
91
92 def list_all(self) -> list[Crop]:
93     """Return current list of all crops (original order)."""
94     return list[Crop](self._crops)
95
96
97 def main():
98     system = AgricultureMonitoringSystem()
99
100     # Sample crop data
101     system.add_crop(Crop("C001", "Wheat", 45.2, 22.0, 3.8))
102     system.add_crop(Crop("C002", "Corn", 62.1, 25.5, 5.2))
103     system.add_crop(Crop("C003", "Rice", 78.0, 28.0, 4.1))
104     system.add_crop(Crop("C004", "Barley", 38.5, 20.0, 3.2))
105     system.add_crop(Crop("C005", "Soybean", 55.0, 24.0, 4.5))
106
107     print("=== Agriculture Monitoring System ===\n")
108
109     # 1. Search by crop ID
110     print("\n1. Search by crop ID")
111     print("-" * 40)
112     for cid in ["C002", "C009"]:
113         crop = system.search_by_id(cid)
114         if crop:
115             print(f"    Found: {crop}")
116         else:
117             print(f"    No crop with ID {cid!r}")
118
119     # 2. Sort by moisture (ascending: driest first)
120     print("\n2. Crops sorted by soil moisture (ascending)")
121     print("-" * 40)
122     for c in system.sort_by_moisture(descending=False):
123         print(f"    {c.soil_moisture:5.1f}% - {c.name} (ID: {c.crop_id})")
124
125     # 3. Sort by yield estimate (descending: highest first)
126     print("\n3. Crops sorted by yield estimate (descending)")
127     print("-" * 40)
128     for c in system.sort_by_yield_estimate(descending=True):
129         print(f"    {c.yield_estimate:4.1f} - {c.name} (ID: {c.crop_id})")
130
131     print("\nDone.")
132
133
134 if __name__ == "__main__":
135     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-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:**

```

1  """
2  Airport Flight Information System
3  - Search flight details by flight ID
4  - Sort flights by departure time or arrival time
5  """
6
7  from datetime import datetime
8  from typing import Optional
9
10
11 # ===== ALGORITHM RECOMMENDATIONS & JUSTIFICATION =====
12 #
13 # 1. SEARCH BY FLIGHT ID
14 # Recommended: Hash Table (Python dict) for  $O(1)$  average lookup
15 # Justification:
16 # - Flight ID is a unique key; hash table gives constant-time lookup.
17 # - No need to keep list sorted by ID just for search.
18 # - Alternatives: Binary Search  $O(\log n)$  if list were sorted by ID;
19 #   Linear Search  $O(n)$  is simple but slow for many flights.
20 #
21 # 2. SORT BY DEPARTURE/ARRIVAL TIME
22 # Recommended: Timsort (Python's sorted()) -  $O(n \log n)$ , stable
23 # Justification:
24 # - Timsort is Python's default; optimal for real-world data (handles
25 #   partial order, few comparisons).
26 # - Stable sort preserves relative order of equal keys (e.g. same time).
27 # - Alternatives: Merge Sort  $O(n \log n)$  stable; Quick Sort  $O(n \log n)$ 
28 #   average but not stable.
29 # =====
30
31
32 class Flight:
33     """Represents a single flight record."""
34
35     def __init__(
36         self,
37         flight_id: str,
38         airline: str,
39         departure_time: str,
40         arrival_time: str,
41         status: str,
42     ):
43         self.flight_id = flight_id
44         self.airline = airline
45         self.departure_time = departure_time # e.g. "14:30" or "2025-02-20 14:30"
46         self.arrival_time = arrival_time
47         self.status = status # e.g. "On Time", "Delayed", "Cancelled"
48
49     def __repr__(self):
50         return (
51             f"Flight(id={self.flight_id}, airline={self.airline}, "
52             f"dep={self.departure_time}, arr={self.arrival_time}, status={self.status})"
53         )
54
55
56 class AirportFlightSystem:
57     """
58     Manages flight data with:
59     -  $O(1)$  search by flight ID (hash table / dict)
60     -  $O(n \log n)$  sort by departure or arrival time (Timsort via sorted())
61     """
62
63     def __init__(self):
64         self.flights_list: list[Flight] = []
65         self._by_id: dict[str, Flight] = {} # Hash table for search by flight ID
66
67     def add_flight(self, flight: Flight) -> None:
68         """Add a flight; keeps list and index in sync."""
69         self.flights_list.append(flight)
70         self._by_id[flight.flight_id] = flight
71
72     def search_by_flight_id(self, flight_id: str) -> Optional[Flight]:
73         """
74         Search flight by ID using hash table lookup.
75         Algorithm: Hash table lookup -  $O(1)$  average time.
76         """
77         return self._by_id.get(flight_id)
78
79     def _parse_time(self, time_str: str) -> datetime:
80         """Parse time string for comparison. Supports 'HH:MM' or 'YYYY-MM-DD HH:MM'."""
81         time_str = time_str.strip()
82         for fmt in ("%Y-%m-%d %H:%M", "%H:%M", "%H:%M:%S"):
83             try:
84                 return datetime.strptime(time_str, fmt)
85             except ValueError:
86                 continue

```

```

87         raise ValueError(f"Cannot parse time: {time_str}")
88
89     def sort_by_departure_time(self, ascending: bool = True) -> list[Flight]:
90         """
91         Sort flights by departure time.
92         Algorithm: Timsort (sorted()) - O(n log n), stable.
93         """
94         return sorted(
95             self._flights_list,
96             key=lambda f: self._parse_time(f.departure_time),
97             reverse=not ascending,
98         )
99
100     def sort_by_arrival_time(self, ascending: bool = True) -> list[Flight]:
101         """
102         Sort flights by arrival time.
103         Algorithm: Timsort (sorted()) - O(n log n), stable.
104         """
105         return sorted(
106             self._flights_list,
107             key=lambda f: self._parse_time(f.arrival_time),
108             reverse=not ascending,
109         )
110
111     def get_all_flights(self) -> list[Flight]:
112         """Return current list of flights (unsorted)."""
113         return self._flights_list.copy()
114
115
116 # ===== DEMO / USAGE =====
117
118 def main():
119     system = AirportFlightSystem()
120
121     # Sample flights
122     flights_data = [
123         ("AA101", "American Airlines", "08:00", "11:30", "On Time"),
124         ("BA205", "British Airways", "14:30", "18:45", "Delayed"),
125         ("EK301", "Emirates", "06:15", "12:00", "On Time"),
126         ("LH402", "Lufthansa", "22:00", "02:30", "On Time"),
127         ("SQ501", "Singapore Airlines", "10:45", "16:20", "On Time"),
128     ]
129
130     for fid, airline, dep, arr, status in flights_data:
131         system.add_flight(Flight(fid, airline, dep, arr, status))
132
133     print("\n== 1. SEARCH BY FLIGHT ID ==\n")
134     for fid in ["EK301", "XX999"]:
135         flight = system.search_by_flight_id(fid)
136         if flight:
137             print(f"Found: {flight}")
138         else:
139             print(f"Flight ID '{fid}' not found.")
140
141     print("\n== 2. SORT BY DEPARTURE TIME (ascending) ==\n")
142     for f in system.sort_by_departure_time(ascending=True):
143         print(f" {f.departure_time} -> {f.flight_id} ({f.airline})")
144
145     print("\n== 3. SORT BY ARRIVAL TIME (ascending) ==\n")
146     for f in system.sort_by_arrival_time(ascending=True):
147         print(f" {f.arrival_time} -> {f.flight_id} ({f.airline})")
148
149     print("\n== 4. SORT BY DEPARTURE TIME (descending) ==\n")
150     for f in system.sort_by_departure_time(ascending=False):
151         print(f" {f.departure_time} -> {f.flight_id} ({f.airline})")
152
153
154 if __name__ == "__main__":
155     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)
```