

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:

```

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, 98], [11, 12, 22, 25, 34, 64, 98]),
43         ([5, 8, 1, 9], [1, 2, 5, 8, 9]),
44         ([1], [1]),
45         ([1, 1]),
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"\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([2, 1]) -> [1, 2] (expected [1, 2])
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([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:

```

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 (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 '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:

```

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), Space: O(n) for auxiliary arrays.
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
27 @dataclass
28 class Appointment:
29     """Single appointment record."""
30     appointment_id: int
31     patient_name: str
32     doctor_name: str
33     appointment_time: str # e.g. "2025-02-20 10:30"
34     consultation_fee: float
35
36     def __repr__(self):
37         return (
38             f"Appointment(id={self.appointment_id}, patient={self.patient_name}), "
39             f"doctor={self.doctor_name}, time={self.appointment_time}, fee={self.consultation_fee})"
40
41
42
43 # 1. SEARCH BY APPOINTMENT ID (Binary Search)
44 #
45 # Precondition: appointments must be sorted by appointment_id (ascending).
46
47 def binary_search_by_id(
48     appointments: list[Appointment],
49     target_id: int,
50 ) -> Optional[Appointment]:
51
52     Search for an appointment by ID using Binary Search.
53     Returns the Appointment if found, else None.
54     ==
55     if not appointments:
56         return None
57
58     # Ensure we search in a list sorted by appointment_id
59     sorted_by_id = sorted(appointments, key=lambda a: a.appointment_id)
60     left, right = 0, len(sorted_by_id) - 1
61     while left <= right:
62         mid = (left + right) // 2
63         a = sorted_by_id[mid]
64         if a.appointment_id == target_id:
65             return a
66         if a.appointment_id < target_id:
67             left = mid + 1
68         else:
69             right = mid - 1
70     return None
71
72
73 # 2. SORT APPOINTMENTS (Merge Sort)
74 #
75 # Sort by a numeric or comparable key (e.g. time string, fee).
76 # Uses a key function: by_time or by_fee.
77
78 def merge_sort_appointments(
79     appointments: list[Appointment],
80     *,
81     by_time: bool = True,
82 ) -> list[Appointment]:
83     """
84     Sort appointments using Merge Sort.
85     by_time=True -> sort by appointment_time (string comparison).
86

```

```

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 simple 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    # -----
147    # Done
148    # -----
149    def main():
150        # Sample appointment records
151        appointments = [
152            Appointment(id=105, patient="Alice Chen", doctor="Dr. Smith", time='2025-02-20 14:00', fee=1500.0),
153            Appointment(id=102, patient="Bob Kim", doctor="Dr. Jones", time='2025-02-20 09:30', fee=2000.0),
154            Appointment(id=108, patient="Carol Lee", doctor="Dr. Smith", time='2025-02-20 11:00', fee=1500.0),
155            Appointment(id=101, "David Park", "Dr. Brown", "2025-02-20 08:00", 1200.0),
156            Appointment(id=110, "Eve Wong", "Dr. Jones", "2025-02-20 16:30", 2500.0),
157        ]
158
159        print("== Healthcare Appointment System ==\n")
160        print("Original list (unsorted):")
161        for a in appointments:
162            print(f" {a}")
163        print()
164
165        # 1. Search by appointment ID
166        print("== Search by Appointment ID (Binary Search) ==")
167        for target in [102, 103]:
168            result = binary_search_by_id(appointments, target)
169            print(f" ID {target}: {result if result else 'Not found'}")
170        print()
171
172        # 2. Sort by time
173        print("== Sorted by Appointment Time (Merge Sort) ==")
174        by_time = sort_appointments_by_time(appointments)
175        for a in by_time:
176            print(f" {a.appointment_time} | {a.patient_name} | fee={a.consultation_fee}")
177        print()
178
179        # 3. Sort by consultation fee
180        print("== Sorted by Consultation Fee (Merge Sort) ==")
181        by_fee = sort_appointments_by_fee(appointments)
182        for a in by_fee:
183            print(f" fee={a.consultation_fee} | {a.patient_name} | {a.appointment_time}")
184
185
186    if __name__ == "__main__":
187        main()
188

```

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:

```

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: O(n log n), stable sort. Stable means equal seat numbers
73             keep their relative order, which is good for consistent display.
74         """
75         return sorted(self._bookings, key=lambda b: b.seat_number)
76
77     def __len__(self) -> int:
78         return len(self._bookings)
79
80     def __iter__(self) -> Iterator[Booking]:
81         return iter(self._bookings)
82
83     def __repr__(self) -> str:
84         return f"RailwayReservationSystem({len(self)} bookings)"
85
86     def __str__(self) -> str:
87         return f"Railway Reservation System with {len(self)} bookings."
88
89     def __eq__(self, other: object) -> bool:
90         if isinstance(other, RailwayReservationSystem):
91             return self._bookings == other._bookings
92         return False
93
94     def __ne__(self, other: object) -> bool:
95         if isinstance(other, RailwayReservationSystem):
96             return self._bookings != other._bookings
97         return True
98
99     def __hash__(self) -> int:
100        return hash(tuple(self._bookings))
101
102    def __getitem__(self, index: int) -> Booking:
103        return self._bookings[index]
104
105    def __setitem__(self, index: int, value: Booking):
106        self._bookings[index] = value
107
108    def __delitem__(self, index: int):
109        del self._bookings[index]
110
111    def __contains__(self, ticket_id: str) -> bool:
112        return ticket_id in self._by_ticket_id
113
114    def __del__(self):
115        del self._bookings
116        del self._by_ticket_id
117
118    def __getstate__(self) -> dict:
119        return {"bookings": self._bookings}
120
121    def __setstate__(self, state: dict):
122        self._bookings = state["bookings"]
123
124    def __getattribute__(self, name: str):
125        if name in ["_bookings", "_by_ticket_id"]:
126            return super().__getattribute__(name)
127        return self.__dict__.get(name)
128
129    def __setattr__(self, name: str, value):
130        if name in ["_bookings", "_by_ticket_id"]:
131            return super().__setattr__(name, value)
132        self.__dict__[name] = value
133
134    def __delattr__(self, name: str):
135        if name in ["_bookings", "_by_ticket_id"]:
136            return super().__delattr__(name)
137        del self.__dict__[name]
138
139    def __dir__(self) -> list:
140        return dir(self.__dict__)
141
142    def __bool__(self) -> bool:
143        return bool(self._bookings)
144
145    def __len__(self) -> int:
146        return len(self._bookings)
147
148    def __iter__(self) -> Iterator[Booking]:
149        return iter(self._bookings)
150
151    def __repr__(self) -> str:
152        return f"RailwayReservationSystem({len(self)} bookings)"
153
154    def __str__(self) -> str:
155        return f"Railway Reservation System with {len(self)} bookings."
156
157    def __eq__(self, other: object) -> bool:
158        if isinstance(other, RailwayReservationSystem):
159            return self._bookings == other._bookings
160        return False
161
162    def __ne__(self, other: object) -> bool:
163        if isinstance(other, RailwayReservationSystem):
164            return self._bookings != other._bookings
165        return True
166
167    def __hash__(self) -> int:
168        return hash(tuple(self._bookings))
169
170    def __getitem__(self, index: int) -> Booking:
171        return self._bookings[index]
172
173    def __setitem__(self, index: int, value: Booking):
174        self._bookings[index] = value
175
176    def __delitem__(self, index: int):
177        del self._bookings[index]
178
179    def __contains__(self, ticket_id: str) -> bool:
180        return ticket_id in self._by_ticket_id
181
182    def __del__(self):
183        del self._bookings
184        del self._by_ticket_id
185
186    def __getstate__(self) -> dict:
187        return {"bookings": self._bookings}
188
189    def __setstate__(self, state: dict):
190        self._bookings = state["bookings"]
191
192    def __getattribute__(self, name: str):
193        if name in ["_bookings", "_by_ticket_id"]:
194            return super().__getattribute__(name)
195        return self.__dict__.get(name)
196
197    def __setattr__(self, name: str, value):
198        if name in ["_bookings", "_by_ticket_id"]:
199            return super().__setattr__(name, value)
200        self.__dict__[name] = value
201
202    def __delattr__(self, name: str):
203        if name in ["_bookings", "_by_ticket_id"]:
204            return super().__delattr__(name)
205        del self.__dict__[name]
206
207    def __dir__(self) -> list:
208        return dir(self.__dict__)
209
210    def __bool__(self) -> bool:
211        return bool(self._bookings)
212
213    def __len__(self) -> int:
214        return len(self._bookings)
215
216    def __iter__(self) -> Iterator[Booking]:
217        return iter(self._bookings)
218
219    def __repr__(self) -> str:
220        return f"RailwayReservationSystem({len(self)} bookings)"
221
222    def __str__(self) -> str:
223        return f"Railway Reservation System with {len(self)} bookings."
224
225    def __eq__(self, other: object) -> bool:
226        if isinstance(other, RailwayReservationSystem):
227            return self._bookings == other._bookings
228        return False
229
230    def __ne__(self, other: object) -> bool:
231        if isinstance(other, RailwayReservationSystem):
232            return self._bookings != other._bookings
233        return True
234
235    def __hash__(self) -> int:
236        return hash(tuple(self._bookings))
237
238    def __getitem__(self, index: int) -> Booking:
239        return self._bookings[index]
240
241    def __setitem__(self, index: int, value: Booking):
242        self._bookings[index] = value
243
244    def __delitem__(self, index: int):
245        del self._bookings[index]
246
247    def __contains__(self, ticket_id: str) -> bool:
248        return ticket_id in self._by_ticket_id
249
250    def __del__(self):
251        del self._bookings
252        del self._by_ticket_id
253
254    def __getstate__(self) -> dict:
255        return {"bookings": self._bookings}
256
257    def __setstate__(self, state: dict):
258        self._bookings = state["bookings"]
259
260    def __getattribute__(self, name: str):
261        if name in ["_bookings", "_by_ticket_id"]:
262            return super().__getattribute__(name)
263        return self.__dict__.get(name)
264
265    def __setattr__(self, name: str, value):
266        if name in ["_bookings", "_by_ticket_id"]:
267            return super().__setattr__(name, value)
268        self.__dict__[name] = value
269
270    def __delattr__(self, name: str):
271        if name in ["_bookings", "_by_ticket_id"]:
272            return super().__delattr__(name)
273        del self.__dict__[name]
274
275    def __dir__(self) -> list:
276        return dir(self.__dict__)
277
278    def __bool__(self) -> bool:
279        return bool(self._bookings)
280
281    def __len__(self) -> int:
282        return len(self._bookings)
283
284    def __iter__(self) -> Iterator[Booking]:
285        return iter(self._bookings)
286
287    def __repr__(self) -> str:
288        return f"RailwayReservationSystem({len(self)} bookings)"
289
290    def __str__(self) -> str:
291        return f"Railway Reservation System with {len(self)} bookings."
292
293    def __eq__(self, other: object) -> bool:
294        if isinstance(other, RailwayReservationSystem):
295            return self._bookings == other._bookings
296        return False
297
298    def __ne__(self, other: object) -> bool:
299        if isinstance(other, RailwayReservationSystem):
300            return self._bookings != other._bookings
301        return True
302
303    def __hash__(self) -> int:
304        return hash(tuple(self._bookings))
305
306    def __getitem__(self, index: int) -> Booking:
307        return self._bookings[index]
308
309    def __setitem__(self, index: int, value: Booking):
310        self._bookings[index] = value
311
312    def __delitem__(self, index: int):
313        del self._bookings[index]
314
315    def __contains__(self, ticket_id: str) -> bool:
316        return ticket_id in self._by_ticket_id
317
318    def __del__(self):
319        del self._bookings
320        del self._by_ticket_id
321
322    def __getstate__(self) -> dict:
323        return {"bookings": self._bookings}
324
325    def __setstate__(self, state: dict):
326        self._bookings = state["bookings"]
327
328    def __getattribute__(self, name: str):
329        if name in ["_bookings", "_by_ticket_id"]:
330            return super().__getattribute__(name)
331        return self.__dict__.get(name)
332
333    def __setattr__(self, name: str, value):
334        if name in ["_bookings", "_by_ticket_id"]:
335            return super().__setattr__(name, value)
336        self.__dict__[name] = value
337
338    def __delattr__(self, name: str):
339        if name in ["_bookings", "_by_ticket_id"]:
340            return super().__delattr__(name)
341        del self.__dict__[name]
342
343    def __dir__(self) -> list:
344        return dir(self.__dict__)
345
346    def __bool__(self) -> bool:
347        return bool(self._bookings)
348
349    def __len__(self) -> int:
350        return len(self._bookings)
351
352    def __iter__(self) -> Iterator[Booking]:
353        return iter(self._bookings)
354
355    def __repr__(self) -> str:
356        return f"RailwayReservationSystem({len(self)} bookings)"
357
358    def __str__(self) -> str:
359        return f"Railway Reservation System with {len(self)} bookings."
360
361    def __eq__(self, other: object) -> bool:
362        if isinstance(other, RailwayReservationSystem):
363            return self._bookings == other._bookings
364        return False
365
366    def __ne__(self, other: object) -> bool:
367        if isinstance(other, RailwayReservationSystem):
368            return self._bookings != other._bookings
369        return True
370
371    def __hash__(self) -> int:
372        return hash(tuple(self._bookings))
373
374    def __getitem__(self, index: int) -> Booking:
375        return self._bookings[index]
376
377    def __setitem__(self, index: int, value: Booking):
378        self._bookings[index] = value
379
380    def __delitem__(self, index: int):
381        del self._bookings[index]
382
383    def __contains__(self, ticket_id: str) -> bool:
384        return ticket_id in self._by_ticket_id
385
386    def __del__(self):
387        del self._bookings
388        del self._by_ticket_id
389
390    def __getstate__(self) -> dict:
391        return {"bookings": self._bookings}
392
393    def __setstate__(self, state: dict):
394        self._bookings = state["bookings"]
395
396    def __getattribute__(self, name: str):
397        if name in ["_bookings", "_by_ticket_id"]:
398            return super().__getattribute__(name)
399        return self.__dict__.get(name)
400
401    def __setattr__(self, name: str, value):
402        if name in ["_bookings", "_by_ticket_id"]:
403            return super().__setattr__(name, value)
404        self.__dict__[name] = value
405
406    def __delattr__(self, name: str):
407        if name in ["_bookings", "_by_ticket_id"]:
408            return super().__delattr__(name)
409        del self.__dict__[name]
410
411    def __dir__(self) -> list:
412        return dir(self.__dict__)
413
414    def __bool__(self) -> bool:
415        return bool(self._bookings)
416
417    def __len__(self) -> int:
418        return len(self._bookings)
419
420    def __iter__(self) -> Iterator[Booking]:
421        return iter(self._bookings)
422
423    def __repr__(self) -> str:
424        return f"RailwayReservationSystem({len(self)} bookings)"
425
426    def __str__(self) -> str:
427        return f"Railway Reservation System with {len(self)} bookings."
428
429    def __eq__(self, other: object) -> bool:
430        if isinstance(other, RailwayReservationSystem):
431            return self._bookings == other._bookings
432        return False
433
434    def __ne__(self, other: object) -> bool:
435        if isinstance(other, RailwayReservationSystem):
436            return self._bookings != other._bookings
437        return True
438
439    def __hash__(self) -> int:
440        return hash(tuple(self._bookings))
441
442    def __getitem__(self, index: int) -> Booking:
443        return self._bookings[index]
444
445    def __setitem__(self, index: int, value: Booking):
446        self._bookings[index] = value
447
448    def __delitem__(self, index: int):
449        del self._bookings[index]
450
451    def __contains__(self, ticket_id: str) -> bool:
452        return ticket_id in self._by_ticket_id
453
454    def __del__(self):
455        del self._bookings
456        del self._by_ticket_id
457
458    def __getstate__(self) -> dict:
459        return {"bookings": self._bookings}
460
461    def __setstate__(self, state: dict):
462        self._bookings = state["bookings"]
463
464    def __getattribute__(self, name: str):
465        if name in ["_bookings", "_by_ticket_id"]:
466            return super().__getattribute__(name)
467        return self.__dict__.get(name)
468
469    def __setattr__(self, name: str, value):
470        if name in ["_bookings", "_by_ticket_id"]:
471            return super().__setattr__(name, value)
472        self.__dict__[name] = value
473
474    def __delattr__(self, name: str):
475        if name in ["_bookings", "_by_ticket_id"]:
476            return super().__delattr__(name)
477        del self.__dict__[name]
478
479    def __dir__(self) -> list:
480        return dir(self.__dict__)
481
482    def __bool__(self) -> bool:
483        return bool(self._bookings)
484
485    def __len__(self) -> int:
486        return len(self._bookings)
487
488    def __iter__(self) -> Iterator[Booking]:
489        return iter(self._bookings)
490
491    def __repr__(self) -> str:
492        return f"RailwayReservationSystem({len(self)} bookings)"
493
494    def __str__(self) -> str:
495        return f"Railway Reservation System with {len(self)} bookings."
496
497    def __eq__(self, other: object) -> bool:
498        if isinstance(other, RailwayReservationSystem):
499            return self._bookings == other._bookings
500        return False
501
502    def __ne__(self, other: object) -> bool:
503        if isinstance(other, RailwayReservationSystem):
504            return self._bookings != other._bookings
505        return True
506
507    def __hash__(self) -> int:
508        return hash(tuple(self._bookings))
509
510    def __getitem__(self, index: int) -> Booking:
511        return self._bookings[index]
512
513    def __setitem__(self, index: int, value: Booking):
514        self._bookings[index] = value
515
516    def __delitem__(self, index: int):
517        del self._bookings[index]
518
519    def __contains__(self, ticket_id: str) -> bool:
520        return ticket_id in self._by_ticket_id
521
522    def __del__(self):
523        del self._bookings
524        del self._by_ticket_id
525
526    def __getstate__(self) -> dict:
527        return {"bookings": self._bookings}
528
529    def __setstate__(self, state: dict):
530        self._bookings = state["bookings"]
531
532    def __getattribute__(self, name: str):
533        if name in ["_bookings", "_by_ticket_id"]:
534            return super().__getattribute__(name)
535        return self.__dict__.get(name)
536
537    def __setattr__(self, name: str, value):
538        if name in ["_bookings", "_by_ticket_id"]:
539            return super().__setattr__(name, value)
540        self.__dict__[name] = value
541
542    def __delattr__(self, name: str):
543        if name in ["_bookings", "_by_ticket_id"]:
544            return super().__delattr__(name)
545        del self.__dict__[name]
546
547    def __dir__(self) -> list:
548        return dir(self.__dict__)
549
550    def __bool__(self) -> bool:
551        return bool(self._bookings)
552
553    def __len__(self) -> int:
554        return len(self._bookings)
555
556    def __iter__(self) -> Iterator[Booking]:
557        return iter(self._bookings)
558
559    def __repr__(self) -> str:
560        return f"RailwayReservationSystem({len(self)} bookings)"
561
562    def __str__(self) -> str:
563        return f"Railway Reservation System with {len(self)} bookings."
564
565    def __eq__(self, other: object) -> bool:
566        if isinstance(other, RailwayReservationSystem):
567            return self._bookings == other._bookings
568        return False
569
570    def __ne__(self, other: object) -> bool:
571        if isinstance(other, RailwayReservationSystem):
572            return self._bookings != other._bookings
573        return True
574
575    def __hash__(self) -> int:
576        return hash(tuple(self._bookings))
577
578    def __getitem__(self, index: int) -> Booking:
579        return self._bookings[index]
580
581    def __setitem__(self, index: int, value: Booking):
582        self._bookings[index] = value
583
584    def __delitem__(self, index: int):
585        del self._bookings[index]
586
587    def __contains__(self, ticket_id: str) -> bool:
588        return ticket_id in self._by_ticket_id
589
590    def __del__(self):
591        del self._bookings
592        del self._by_ticket_id
593
594    def __getstate__(self) -> dict:
595        return {"bookings": self._bookings}
596
597    def __setstate__(self, state: dict):
598        self._bookings = state["bookings"]
599
600    def __getattribute__(self, name: str):
601        if name in ["_bookings", "_by_ticket_id"]:
602            return super().__getattribute__(name)
603        return self.__dict__.get(name)
604
605    def __setattr__(self, name: str, value):
606        if name in ["_bookings", "_by_ticket_id"]:
607            return super().__setattr__(name, value)
608        self.__dict__[name] = value
609
610    def __delattr__(self, name: str):
611        if name in ["_bookings", "_by_ticket_id"]:
612            return super().__delattr__(name)
613        del self.__dict__[name]
614
615    def __dir__(self) -> list:
616        return dir(self.__dict__)
617
618    def __bool__(self) -> bool:
619        return bool(self._bookings)
620
621    def __len__(self) -> int:
622        return len(self._bookings)
623
624    def __iter__(self) -> Iterator[Booking]:
625        return iter(self._bookings)
626
627    def __repr__(self) -> str:
628        return f"RailwayReservationSystem({len(self)} bookings)"
629
630    def __str__(self) -> str:
631        return f"Railway Reservation System with {len(self)} bookings."
632
633    def __eq__(self, other: object) -> bool:
634        if isinstance(other, RailwayReservationSystem):
635            return self._bookings == other._bookings
636        return False
637
638    def __ne__(self, other: object) -> bool:
639        if isinstance(other, RailwayReservationSystem):
640            return self._bookings != other._bookings
641        return True
642
643    def __hash__(self) -> int:
644        return hash(tuple(self._bookings))
645
646    def __getitem__(self, index: int) -> Booking:
647        return self._bookings[index]
648
649    def __setitem__(self, index: int, value: Booking):
650        self._bookings[index] = value
651
652    def __delitem__(self, index: int):
653        del self._bookings[index]
654
655    def __contains__(self, ticket_id: str) -> bool:
656        return ticket_id in self._by_ticket_id
657
658    def __del__(self):
659        del self._bookings
660        del self._by_ticket_id
661
662    def __getstate__(self) -> dict:
663        return {"bookings": self._bookings}
664
665    def __setstate__(self, state: dict):
666        self._bookings = state["bookings"]
667
668    def __getattribute__(self, name: str):
669        if name in ["_bookings", "_by_ticket_id"]:
670            return super().__getattribute__(name)
671        return self.__dict__.get(name)
672
673    def __setattr__(self, name: str, value):
674        if name in ["_bookings", "_by_ticket_id"]:
675            return super().__setattr__(name, value)
676        self.__dict__[name] = value
677
678    def __delattr__(self, name: str):
679        if name in ["_bookings", "_by_ticket_id"]:
680            return super().__delattr__(name)
681        del self.__dict__[name]
682
683    def __dir__(self) -> list:
684        return dir(self.__dict__)
685
686    def __bool__(self) -> bool:
687        return bool(self._bookings)
688
689    def __len__(self) -> int:
690        return len(self._bookings)
691
692    def __iter__(self) -> Iterator[Booking]:
693        return iter(self._bookings)
694
695    def __repr__(self) -> str:
696        return f"RailwayReservationSystem({len(self)} bookings)"
697
698    def __str__(self) -> str:
699        return f"Railway Reservation System with {len(self)} bookings."
700
701    def __eq__(self, other: object) -> bool:
702        if isinstance(other, RailwayReservationSystem):
703            return self._bookings == other._bookings
704        return False
705
706    def __ne__(self, other: object) -> bool:
707        if isinstance(other, RailwayReservationSystem):
708            return self._bookings != other._bookings
709        return True
710
711    def __hash__(self) -> int:
712        return hash(tuple(self._bookings))
713
714    def __getitem__(self, index: int) -> Booking:
715        return self._bookings[index]
716
717    def __setitem__(self, index: int, value: Booking):
718        self._bookings[index] = value
719
720    def __delitem__(self, index: int):
721        del self._bookings[index]
722
723    def __contains__(self, ticket_id: str) -> bool:
724        return ticket_id in self._by_ticket_id
725
726    def __del__(self):
727        del self._bookings
728        del self._by_ticket_id
729
730    def __getstate__(self) -> dict:
731        return {"bookings": self._bookings}
732
733    def __setstate__(self, state: dict):
734        self._bookings = state["bookings"]
735
736    def __getattribute__(self, name: str):
737        if name in ["_bookings", "_by_ticket_id"]:
738            return super().__getattribute__(name)
739        return self.__dict__.get(name)
740
741    def __setattr__(self, name: str, value):
742        if name in ["_bookings", "_by_ticket_id"]:
743            return super().__setattr__(name, value)
744        self.__dict__[name] = value
745
746    def __delattr__(self, name: str):
747        if name in ["_bookings", "_by_ticket_id"]:
748            return super().__delattr__(name)
749        del self.__dict__[name]
750
751    def __dir__(self) -> list:
752        return dir(self.__dict__)
753
754    def __bool__(self) -> bool:
755        return bool(self._bookings)
756
757    def __len__(self) -> int:
758        return len(self._bookings)
759
760    def __iter__(self) -> Iterator[Booking]:
761        return iter(self._bookings)
762
763    def __repr__(self) -> str:
764        return f"RailwayReservationSystem({len(self)} bookings)"
765
766    def __str__(self) -> str:
767        return f"Railway Reservation System with {len(self)} bookings."
768
769    def __eq__(self, other: object) -> bool:
770        if isinstance(other, RailwayReservationSystem):
771            return self._bookings == other._bookings
772        return False
773
774    def __ne__(self, other: object) -> bool:
775        if isinstance(other, RailwayReservationSystem):
776            return self._bookings != other._bookings
777        return True
778
779    def __hash__(self) -> int:
780        return hash(tuple(self._bookings))
781
782    def __getitem__(self, index: int) -> Booking:
783        return self._bookings[index]
784
785    def __setitem__(self, index: int, value: Booking):
786        self._bookings[index] = value
787
788    def __delitem__(self, index: int):
789        del self._bookings[index]
790
791    def __contains__(self, ticket_id: str) -> bool:
792        return ticket_id in self._by_ticket_id
793
794    def __del__(self):
795        del self._bookings
796        del self._by_ticket_id
797
798    def __getstate__(self) -> dict:
799        return {"bookings": self._bookings}
800
801    def __setstate__(self, state: dict):
802        self._bookings = state["bookings"]
803
804    def __getattribute__(self, name: str):
805        if name in ["_bookings", "_by_ticket_id"]:
806            return super().__getattribute__(name)
807        return self.__dict__.get(name)
808
809    def __setattr__(self, name: str, value):
810        if name in ["_bookings", "_by_ticket_id"]:
811            return super().__setattr__(name, value)
812        self.__dict__[name] = value
813
814    def __delattr__(self, name: str):
815        if name in ["_bookings", "_by_ticket_id"]:
816            return super().__delattr__(name)
817        del self.__dict__[name]
818
819    def __dir__(self) -> list:
820        return dir(self.__dict__)
821
822    def __bool__(self) -> bool:
823        return bool(self._bookings)
824
825    def __len__(self) -> int:
826        return len(self._bookings)
827
828    def __iter__(self) -> Iterator[Booking]:
829        return iter(self._bookings)
830
831    def __repr__(self) -> str:
832        return f"RailwayReservationSystem({len(self)} bookings)"
833
834    def __str__(self) -> str:
835        return f"Railway Reservation System with {len(self)} bookings."
836
837    def __eq__(self, other: object) -> bool:
838        if isinstance(other, RailwayReservationSystem):
839            return self._bookings == other._bookings
840        return False
841
842    def __ne__(self, other: object) -> bool:
843        if isinstance(other, RailwayReservationSystem):
844            return self._bookings != other._bookings
845        return True
846
847    def __hash__(self) -> int:
848        return hash(tuple(self._bookings))
849
850    def __getitem__(self, index: int) -> Booking:
851        return self._bookings[index]
852
853    def __setitem__(self, index: int, value: Booking):
854        self._bookings[index] = value
855
856    def __delitem__(self, index: int):
857        del self._bookings[index]
858
859    def __contains__(self, ticket_id: str) -> bool:
860        return ticket_id in self._by_ticket_id
861
862    def __del__(self):
863        del self._bookings
864        del self._by_ticket_id
865
866    def __getstate__(self) -> dict:
867        return {"bookings": self._bookings}
868
869    def __setstate__(self, state: dict):
870        self._bookings = state["bookings"]
871
872    def __getattribute__(self, name: str):
873        if name in ["_bookings", "_by_ticket_id"]:
874            return super().__getattribute__(name)
875        return self.__dict__.get(name)
876
877    def __setattr__(self, name: str, value):
878        if name in ["_bookings", "_by_ticket_id"]:
879            return super().__setattr__(name, value)
880        self.__dict__[name] = value
881
882    def __delattr__(self, name: str):
883        if name in ["_bookings", "_by_ticket_id"]:
884            return super().__delattr__(name)
885        del self.__dict__[name]
886
887    def __dir__(self) -> list:
888        return dir(self.__dict__)
889
890    def __bool__(self) -> bool:
891        return bool(self._bookings)
892
893    def __len__(self) -> int:
894        return len(self._bookings)
895
896    def __iter__(self) -> Iterator[Booking]:
897        return iter(self._bookings)
898
899    def __repr__(self) -> str:
900        return f"RailwayReservationSystem({len(self)} bookings)"
901
902    def __str__(self) -> str:
903        return f"Railway Reservation System with {len(self)} bookings."
904
905    def __eq__(self, other: object) -> bool:
906        if isinstance(other, RailwayReservationSystem):
907            return self._bookings == other._bookings
908        return False
909
910    def __ne__(self, other: object) -> bool:
911        if isinstance(other, RailwayReservationSystem):
912            return self._bookings != other._bookings
913        return True
914
915    def __hash__(self) -> int:
916        return hash(tuple(self._bookings))
917
918    def __getitem__(self, index: int) -> Booking:
919        return self._bookings[index]
920
921    def __setitem__(self, index: int, value: Booking):
922        self._bookings[index] = value
923
924    def __delitem__(self, index: int):
925        del self._bookings[index]
926
927    def __contains__(self, ticket_id: str) -> bool:
928        return ticket_id in self._by_ticket_id
929
930    def __del__(self):
931        del self._bookings
932        del self._by_ticket_id
933
934    def __getstate__(self) -> dict:
935        return {"bookings": self._bookings}
936
937    def __setstate__(self, state: dict):
938        self._bookings = state["bookings"]
939
940    def __getattribute__(self, name: str):
941        if name in ["_bookings", "_by_ticket_id"]:
942            return super().__getattribute__(name)
943        return self.__dict__.get(name)
944
945    def __setattr__(self, name: str, value):
946        if name in ["_bookings", "_by_ticket_id"]:
947            return super().__setattr__(name, value)
948        self.__dict__[name] = value
949
950    def __delattr__(self, name: str):
95
```

```

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 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:

```

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[student_id, AllocationRecord] = {}
37         # List of all records (references immutable 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         """
90             Return all records in insertion order.
91         """
92         return list(self._records)
93
94     # ----- Algorithm summary and justification -----
95
96     ALGORITHM CHOICES & JUSTIFICATION
97     -----
98
99     1. SEARCH BY STUDENT ID
100        - Chosen: Hash table (Python dict) keyed by student_id
101        - Time: O(1) average lookup
102        - Justification:
103            * Each student has at most one current allocation, so student_id is a
104              natural unique key.
105            * Hash table gives constant-time access by key without scanning.
106            * Alternative: reference counting
107                - Linear search O(n); poor for many records.
108                - Binary search O(log n); would require keeping list sorted by
109                  student_id and extra bookkeeping when adding/removing.
110
111    2. SORT BY ROOM NUMBER / ALLOCATION DATE
112        - Chosen: Timsort via sorted(iterable, key...)
113        - Time: O(n log n), stable
114        - Justification:
115            * We need to sort on different keys (room number, date) on demand.
116            * Timsort is Python's default; it is stable and efficient for
117              partially ordered data.
118            * Keeping two sorted copies (by room and by date) would give O(log n)
119              insertion but double storage and more complex updates; For typical
120              hostel size, sorting when needed is simpler and fast enough.
121
122 """

```

```

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

```

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
20         and the list is not already sorted by ID.
21
22 SORT BY RATING OR RELEASE YEAR:
23     Algorithm: Merge Sort
24     Justification:
25         - Stable sort preserves order of movies with equal rating/year (e.g. by title or ID).
26         - Time: O(n log n) in all cases; predictable for streaming-sized datasets.
27         - No worst-case O(n^2) as in native Quick Sort; suitable for user-facing ordering.
28         - Works well with linked/recd Data; easy to sort by different keys (rating, year).
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     def sort_movies(
103         movies: List[dict],
104         by: Literal["rating", "year"],
105         ascending: bool = True,
106     ) -> List[dict]:
107         """
108             Sort movies by 'rating' or 'year' using Merge Sort (stable, O(n log n)).
109             Returns a new sorted list; does not mutate the original.
110             """
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
117         return result
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
134         if not movies:
135             return []
136         result = [m.copy() for m in movies]
137         _merge_sort_range(result, key, ascending, 0, len(result) - 1)
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 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:

```

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
68         return self._by_id.get(crop_id)
69
70     def sort_by_moisture(self, descending: bool = False) -> list[Crop]:
71         """
72
73
74
75
76
77
78
79
79
80
81
82
83
84
85
86
87
88
89
89
90
91
92
93
94
95
96
97
98
99
99
100
101
102
103
104
105
106
107
108
109
109
110
111
112
113
114
115
116
117
118
119
119
120
121
122
123
124
125
126
127
128
129
129
130
131
132
133
134
135
136
137
138
139
139
140
141
142
143
144
145
146
147
148
149
149
150
151
152
153
154
155
156
157
158
159
159
160
161
162
163
164
165
166
167
168
169
169
170
171
172
173
174
175
176
177
178
179
179
180
181
182
183
184
185
186
187
188
189
189
190
191
192
193
194
195
196
197
198
199
199
200
201
202
203
204
205
206
207
208
209
209
210
211
212
213
214
215
216
217
218
219
219
220
221
222
223
224
225
226
227
227
228
229
229
230
231
232
233
234
235
236
237
237
238
239
239
240
241
242
243
244
245
245
246
247
247
248
249
249
250
251
252
253
254
255
255
256
257
257
258
259
259
260
261
262
263
263
264
265
265
266
267
267
268
269
269
270
271
272
272
273
274
274
275
276
276
277
278
278
279
279
280
280
281
281
282
282
283
283
284
284
285
285
286
286
287
287
288
288
289
289
290
290
291
291
292
292
293
293
294
294
295
295
296
296
297
297
298
298
299
299
300
300
301
301
302
302
303
303
304
304
305
305
306
306
307
307
308
308
309
309
310
310
311
311
312
312
313
313
314
314
315
315
316
316
317
317
318
318
319
319
320
320
321
321
322
322
323
323
324
324
325
325
326
326
327
327
328
328
329
329
330
330
331
331
332
332
333
333
334
334
335
335
336
336
337
337
338
338
339
339
340
340
341
341
342
342
343
343
344
344
345
345
346
346
347
347
348
348
349
349
350
350
351
351
352
352
353
353
354
354
355
355
356
356
357
357
358
358
359
359
360
360
361
361
362
362
363
363
364
364
365
365
366
366
367
367
368
368
369
369
370
370
371
371
372
372
373
373
374
374
375
375
376
376
377
377
378
378
379
379
380
380
381
381
382
382
383
383
384
384
385
385
386
386
387
387
388
388
389
389
390
390
391
391
392
392
393
393
394
394
395
395
396
396
397
397
398
398
399
399
400
400
401
401
402
402
403
403
404
404
405
405
406
406
407
407
408
408
409
409
410
410
411
411
412
412
413
413
414
414
415
415
416
416
417
417
418
418
419
419
420
420
421
421
422
422
423
423
424
424
425
425
426
426
427
427
428
428
429
429
430
430
431
431
432
432
433
433
434
434
435
435
436
436
437
437
438
438
439
439
440
440
441
441
442
442
443
443
444
444
445
445
446
446
447
447
448
448
449
449
450
450
451
451
452
452
453
453
454
454
455
455
456
456
457
457
458
458
459
459
460
460
461
461
462
462
463
463
464
464
465
465
466
466
467
467
468
468
469
469
470
470
471
471
472
472
473
473
474
474
475
475
476
476
477
477
478
478
479
479
480
480
481
481
482
482
483
483
484
484
485
485
486
486
487
487
488
488
489
489
490
490
491
491
492
492
493
493
494
494
495
495
496
496
497
497
498
498
499
499
500
500
501
501
502
502
503
503
504
504
505
505
506
506
507
507
508
508
509
509
510
510
511
511
512
512
513
513
514
514
515
515
516
516
517
517
518
518
519
519
520
520
521
521
522
522
523
523
524
524
525
525
526
526
527
527
528
528
529
529
530
530
531
531
532
532
533
533
534
534
535
535
536
536
537
537
538
538
539
539
540
540
541
541
542
542
543
543
544
544
545
545
546
546
547
547
548
548
549
549
550
550
551
551
552
552
553
553
554
554
555
555
556
556
557
557
558
558
559
559
560
560
561
561
562
562
563
563
564
564
565
565
566
566
567
567
568
568
569
569
570
570
571
571
572
572
573
573
574
574
575
575
576
576
577
577
578
578
579
579
580
580
581
581
582
582
583
583
584
584
585
585
586
586
587
587
588
588
589
589
590
590
591
591
592
592
593
593
594
594
595
595
596
596
597
597
598
598
599
599
600
600
601
601
602
602
603
603
604
604
605
605
606
606
607
607
608
608
609
609
610
610
611
611
612
612
613
613
614
614
615
615
616
616
617
617
618
618
619
619
620
620
621
621
622
622
623
623
624
624
625
625
626
626
627
627
628
628
629
629
630
630
631
631
632
632
633
633
634
634
635
635
636
636
637
637
638
638
639
639
640
640
641
641
642
642
643
643
644
644
645
645
646
646
647
647
648
648
649
649
650
650
651
651
652
652
653
653
654
654
655
655
656
656
657
657
658
658
659
659
660
660
661
661
662
662
663
663
664
664
665
665
666
666
667
667
668
668
669
669
670
670
671
671
672
672
673
673
674
674
675
675
676
676
677
677
678
678
679
679
680
680
681
681
682
682
683
683
684
684
685
685
686
686
687
687
688
688
689
689
690
690
691
691
692
692
693
693
694
694
695
695
696
696
697
697
698
698
699
699
700
700
701
701
702
702
703
703
704
704
705
705
706
706
707
707
708
708
709
709
710
710
711
711
712
712
713
713
714
714
715
715
716
716
717
717
718
718
719
719
720
720
721
721
722
722
723
723
724
724
725
725
726
726
727
727
728
728
729
729
730
730
731
731
732
732
733
733
734
734
735
735
736
736
737
737
738
738
739
739
740
740
741
741
742
742
743
743
744
744
745
745
746
746
747
747
748
748
749
749
750
750
751
751
752
752
753
753
754
754
755
755
756
756
757
757
758
758
759
759
760
760
761
761
762
762
763
763
764
764
765
765
766
766
767
767
768
768
769
769
770
770
771
771
772
772
773
773
774
774
775
775
776
776
777
777
778
778
779
779
780
780
781
781
782
782
783
783
784
784
785
785
786
786
787
787
788
788
789
789
790
790
791
791
792
792
793
793
794
794
795
795
796
796
797
797
798
798
799
799
800
800
801
801
802
802
803
803
804
804
805
805
806
806
807
807
808
808
809
809
810
810
811
811
812
812
813
813
814
814
815
815
816
816
817
817
818
818
819
819
820
820
821
821
822
822
823
823
824
824
825
825
826
826
827
827
828
828
829
829
830
830
831
831
832
832
833
833
834
834
835
835
836
836
837
837
838
838
839
839
840
840
841
841
842
842
843
843
844
844
845
845
846
846
847
847
848
848
849
849
850
850
851
851
852
852
853
853
854
854
855
855
856
856
857
857
858
858
859
859
860
860
861
861
862
862
863
863
864
864
865
865
866
866
867
867
868
868
869
869
870
870
871
871
872
872
873
873
874
874
875
875
876
876
877
877
878
878
879
879
880
880
881
881
882
882
883
883
884
884
885
885
886
886
887
887
888
888
889
889
890
890
891
891
892
892
893
893
894
894
895
895
896
896
897
897
898
898
899
899
900
900
901
901
902
902
903
903
904
904
905
905
906
906
907
907
908
908
909
909
910
910
911
911
912
912
913
913
914
914
915
915
916
916
917
917
918
918
919
919
920
920
921
921
922
922
923
923
924
924
925
925
926
926
927
927
928
928
929
929
930
930
931
931
932
932
933
933
934
934
935
935
936
936
937
937
938
938
939
939
940
940
941
941
942
942
943
943
944
944
945
945
946
946
947
947
948
948
949
949
950
950
951
951
952
952
953
953
954
954
955
955
956
956
957
957
958
958
959
959
960
960
961
961
962
962
963
963
964
964
965
965
966
966
967
967
968
968
969
969
970
970
971
971
972
972
973
973
974
974
975
975
976
976
977
977
978
978
979
979
980
980
981
981
982
982
983
983
984
984
985
985
986
986
987
987
988
988
989
989
990
990
991
991
992
992
993
993
994
994
995
995
996
996
997
997
998
998
999
999
1000
1000
1001
1001
1002
1002
1003
1003
1004
1004
1005
1005
1006
1006
1007
1007
1008
1008
1009
1009
1010
1010
1011
1011
1012
1012
1013
1013
1014
1014
1015
1015
1016
1016
1017
1017
1018
1018
1019
1019
1020
1020
1021
1021
1022
1022
1023
1023
1024
1024
1025
1025
1026
1026
1027
1027
1028
1028
1029
1029
1030
1030
1031
1031
1032
1032
1033
1033
1034
1034
1035
1035
1036
1036
1037
1037
1038
1038
1039
1039
1040
1040
1041
1041
1042
1042
1043
1043
1044
1044
1045
1045
1046
1046
1047
1047
1048
1048
1049
1049
1050
1050
1051
1051
1052
1052
1053
1053
1054
1054
1055
1055
1056
1056
1057
1057
1058
1058
1059
1059
1060
1060
1061
1061
1062
1062
1063
1063
1064
1064
1065
1065
1066
1066
1067
1067
1068
1068
1069
1069
1070
1070
1071
1071
1072
1072
1073
1073
1074
1074
1075
1075
1076
1076
1077
1077
1078
1078
1079
1079
1080
1080
1081
1081
1082
1082
1083
1083
1084
1084
1085
1085
1086
1086
1087
1087
1088
1088
1089
1089
1090
1090
1091
1091
1092
1092
1093
1093
1094
1094
1095
1095
1096
1096
1097
1097
1098
1098
1099
1099
1100
1100
1101
1101
1102
1102
1103
1103
1104
1104
1105
1105
1106
1106
1107
1107
1108
1108
1109
1109
1110
1110
1111
1111
1112
1112
1113
1113
1114
1114
1115
1115
1116
1116
1117
1117
1118
1118
1119
1119
1120
1120
1121
1121
1122
1122
1123
1123
1124
1124
1125
1125
1126
1126
1127
1127
1128
1128
1129
1129
1130
1130
1131
1131
1132
1132
1133
1133
1134
1134
1135
1135
1136
1136
1137
1137
1138
1138
1139
1139
1140
1140
1141
1141
1142
1142
1143
1143
1144
1144
1145
1145
1146
1146
1147
1147
1148
1148
1149
1149
1150
1150
1151
1151
1152
1152
1153
1153
1154
1154
1155
1155
1156
1156
1157
1157
1158
1158
1159
1159
1160
1160
1161
1161
1162
1162
1163
1163
1164
1164
1165
1165
1166
1166
1167
1167
1168
1168
1169
1169
1170
1170
1171
1171
1172
1172
1173
1173
1174
1174
1175
1175
1176
1176
1177
1177
1178
1178
1179
1179
1180
1180
1181
1181
1182
1182
1183
1183
1184
1184
1185
1185
1186
1186
1187
1187
1188
1188
1189
1189
1190
1190
1191
1191
1192
1192
1193
1193
1194
1194
1195
1195
1196
1196
1197
1197
1198
1198
1199
1199
1200
1200
1201
1201
1202
1202
1203
1203
1204
1204
1205
1205
1206
1206
1207
1207
1208
1208
1209
1209
1210
1210
1211
1211
1212
1212
1213
1213
1214
1214
1215
1215
1216
1216
1217
1217
1218
1218
1219
1219
1220
1220
1221
1221
1222
1222
1223
1223
1224
1224
1225
1225
1226
1226
1227
1227
1228
1228
1229
1229
1230
1230
1231
1231
1232
1232
1233
1233
1234
1234
1235
1235
1236
1236
1237
1237
1238
1238
1239
1239
1240
1240
1241
1241
1242
1242
1243
1243
1244
1244
1245
1245
1246
1246
1247
1247
1248
1248
1249
1249
1250
1250
1251
1251
1252
1252
1253
1253
1254
1254
1255
1255
1256
1256
1257
1257
1258
1258
1259
1259
1260
1260
1261
1261
1262
1262
1263
1263
1264
1264
1265
1265
1266
1266
1267
1267
1268
1268
1269
1269
1270
1270
1271
1271
1272
1272
1273
1273
1274
1274
1275
1275
1276
1276
1277
1277
1278
1278
1279
1279
1280
1280
1281
1281
1282
1282
1283
1283
1284
1284
1285
1285
1286
1286
1287
1287
1288
1288
1289
1289
1290
1290
1291
1291
1292
1292
1293
1293
1294
1294
1295
1295
1296
1296
1297
1297
1298
1298
1299
1299
1300
1300
1301
1301
1302
1302
1303
1303
1304
1304
1305
1305
1306
1306
1307
1307
1308
1308
1309
1309
1310
1310
1311
1311
1312
1312
1313
1313
1314
1314
1315
1315
1316
1316
1317
1317
1318
1318
1319
1319
1320
1320
1321
1321
1322
1322
1323
1323
1324
1324
1325
1325
1326
1326
1327
1327
1328
1328
1329
1329
1330
1330
1331
1331
1332
1332
1333
1333
1334
1334
1335
1335
1336
1336
1337
1337
1338
1338
1339
1339
1340
1340
1341
1341
1342
1342
1343
1343
1344
1344
1345
1345
1346
1346
1347
1347
1348
1348
1349
1349
1350
1350
1351
1351
1352
1352
1353
1353
1354
1354
1355
1355
1356
1356
1357
1357
1358
1358
1359
1359
1360
1360
1361
1361
1362
1362
1363
1363
1364
1364
1365
1365
1366
1366
1367
1367
1368
1368
1369
1369
1370
1370
1371
1371
1372
1372
1373
1373
1374
1374
1375
1375
1376
1376
1377
1377
1378
1378
1379
1379
1380
1380
1381
1381
1382
1382
1383
1383
1384
1384
1385
1385
1386
1386
1387
1387
1388
1388
1389
1389
1390
1390
1391
1391
1392
1392
1393
1393
1394
1394
1395
1395
1396
1396
1397
1397
1398
1398
1399
1399
1400
1400
1401
1401
1402
1402
1403
1403
1404
1404
1405
1405
1406
1406
1407
1407
1408
1408
1409
1409
1410
1410
1411
1411
1412
1412
1413
1413
1414
1414
1415
1415
1416
1416
1417
1417
1418
1418
1419
1419
1420
1420
1421
1421
1422
1422
1423
1423
1424
1424
1425
1425
1426
1426
1427
1427
1428
1428
1429
1429
1430
1430
1431
1431
1432
1432
1433
1433
1434
1434
1435
1435
1436
1436
1437
1437
1438
1438
1439
1439
1440
1440
1441
1441
1442
1442
1443
1443
1444
1444
1445
1445
1446
1446
1447
1447
1448
1448
1449
1449
1450
1450
1451
1451
1452
1452
1453
1453
1454
1454
1455
1455
1456
1456
1457
1457
1458
1458
1459
1459
1460
1460
1461
1461
1462
1462
1463
1463
1464
1464
1465
1465
1466
1466
1467
1467
1468
1468
1469
1469
1470
1470
1471
1471
1472
1472
1473
1473
1474
1474
1475
1475
1476
1476
1477
1477
1478
1478
1479
1479
1480
1480
1481
1481
1482
1482
1483
1483
1484
1484
1485
1485
1486
1486
1487
1487
1488
1488
1489
1489
1490
1490
1491
```

```

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("1. 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}!")
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:.5f} - {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:.4f} - {c.name} (ID: {c.crop_id})")
130
131        print("\nDone.\n")
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 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:

```

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 #     - Alternative: 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__(self,
36         flight_id: str,
37         airline: str,
38         departure_time: str,
39         arrival_time: str,
40         status: str,
41     ):
42         self.flight_id = flight_id
43         self.airline = airline
44         self.departure_time = departure_time # e.g. "14:30" or "2025-02-20 14:30"
45         self.arrival_time = arrival_time
46         self.status = status # e.g. "On Time", "Delayed", "Cancelled"
47
48     def __repr__(self):
49         return (
50             f"Flight(id={self.flight_id}, airline={self.airline}, "
51             f"dep={self.departure_time}, arr={self.arrival_time}, status={self.status})"
52         )
53
54
55 class AirportFlightSystem:
56     """
57     Manages flight data with:
58     - O(1) search by flight ID (hash table / dict)
59     - O(n log n) sort by departure or arrival time (Timsort via sorted())
60     """
61
62     def __init__(self):
63         self._flights_list: list[Flight] = []
64         self._by_id: dict[str, Flight] = {} # Hash table for search by flight ID
65
66     def add_flight(self, flight: Flight) -> None:
67         """Add a flight; keeps list and index in sync."""
68         self._flights_list.append(flight)
69         self._by_id[flight.flight_id] = flight
70
71     def search_by_flight_id(self, flight_id: str) -> Optional[Flight]:
72         """
73         Search flight by ID using hash table lookup.
74         Algorithm: Hash table lookup - O(1) average time.
75         """
76         return self._by_id.get(flight_id)
77
78     def _parse_time(self, time_str: str) -> datetime:
79         """Parse time string for comparison. Supports 'HH:MM' or 'YYYY-MM-DD HH:MM'."""
80         time_str = time_str.strip()
81         for fmt in ("%Y-%m-%d %H:%M", "%H:%M", "%H:%M:%S"):
82             try:
83                 return datetime.strptime(time_str, fmt)
84             except ValueError:
85                 continue
86

```

```

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        """
113            Return current list of flights (unsorted).
114        """
115        return self._flights_list.copy()
116
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("== 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)
```