COSC2658 - Data Structures and Algorithms/COSC2469 - Algorithms and Analysis/COSC2203 - Algorithms and Analysis

TEST 1 (SAMPLE)

A. Overview of the Test:

1. **Scope**: Covers data structures (trees, lists, stacks, queues, hash tables, graph representations) and algorithms (sorting, searching, string processing, graphs, geometric problems).
2. **Learning Objectives**: Emphasizes understanding and applying data structures, solving algorithmic problems, and grasping trade-offs in algorithm design.
3. **Test Format**: Consists of three problems involving the development of an Abstract Data Type (ADT) for a game, managing student records using a Binary Search Tree, and an algorithm for finding a pair of integers with the minimum product in a sorted array.
4. **Duration**: 2 hours, plus 10 minutes for submission.
5. **Assessment Method**: Individual-based with screen recording. Submission includes Java files and a plain text file in a .zip format.

B. Preparation Advice for Students:

1. **Review Key Concepts**: Revisit lectures and notes on data structures and algorithms. Pay special attention to trees, stacks, queues, and graph representations.
2. **Practice Coding**: Implement various data structures and algorithms in Java. Focus on writing clean, efficient code and understanding how these structures work under the hood.
3. **Solve Previous Problems**: If available, practice with previous test questions or similar problems. This helps in understanding the test pattern and the type of questions asked.
4. **Understand Algorithmic Trade-offs**: Be clear about the trade-offs in different algorithms, like time versus space complexity and deterministic versus randomized algorithms.
5. **Time Management Skills**: Practice solving problems within a limited time. The test is time-bound, so managing time effectively is crucial.
6. **Prepare Your Environment**: Ensure you have a single-screen setup and the necessary software (IDEs, screen recording tools) installed and working. Familiarize yourself with the allowed resources.
7. **Brush Up on Java Skills**: Since submissions are in Java, ensure you're comfortable with Java syntax, file handling, and common libraries.
8. **Complexity Analysis**: Be prepared to analyze the time and space complexity of your algorithms. This is a key aspect of the test.
9. **Test Your Code**: Regularly test your code for various cases to ensure it handles edge cases and errors gracefully.
10. **Relax and Stay Confident**: Finally, ensure you're well-rested before the test, and approach it with confidence. A calm mind often performs better.

Preparation should be balanced between theoretical knowledge and practical coding skills, with an emphasis on understanding the principles behind data structures and algorithms.

Problems Requirements

1. **Problem 1 - Room Escape ADT**:
   - Task: Develop an Abstract Data Type (ADT) for a Room Escape game.
   - Operations: Include functions like `enterRoom` and `exitRoom`.
   - Goal: Showcase understanding of ADT design, encapsulation, and method implementation.
2. **Problem 2 - Student Records with Binary Search Tree**:
   - Task: Create an application to manage student records.
   - Data Structure: Use a Binary Search Tree (BST) to store and retrieve student information.
   - Objective: Demonstrate ability to implement and manipulate a BST for practical data management.
3. **Problem 3 - Minimum Product Pair in Array**:
   - Task: Develop an algorithm to find a pair of integers in a sorted array that yields the minimum product.
   - Additional Requirement: Analyze the time and space complexity of the algorithm.
   - Focus: Test algorithmic problem-solving skills and understanding of complexity analysis.

Each problem targets specific aspects of data structures and algorithms, from designing and implementing an ADT and a BST, to solving an algorithmic challenge with complexity analysis.

PROBLEM 1

To solve Problem 1, let's break it down into steps and develop the solution with Java code. We'll create an `EscapeRoom` class that supports the operations `enterRoom`, `exitRoom`, and `minOperations`. This class will essentially function as a stack, implementing the Last In, First Out (LIFO) order for room entries and exits.

Step 1: Define the EscapeRoom Class

First, we define the `EscapeRoom` class and its underlying data structure. We'll use a `Stack<String>` to store the rooms.

```
import java.util.Stack;

public class EscapeRoom {
    private Stack<String> roomStack;

    public EscapeRoom() {
        roomStack = new Stack<>();
    }

    // Additional methods will be added here
}
```

Step 2: Implement enterRoom Method

The `enterRoom` method adds a room to the stack. Its time complexity is O(1), as stack operations are constant time.

```
// Complexity = O(1)
public void enterRoom(String room) {
    roomStack.push(room);
}
```

Step 3: Implement exitRoom Method

The `exitRoom` method removes the last added room and returns its name. If the stack is empty, it returns null. Its complexity is also O(1).

```
// Complexity = O(1)
public String exitRoom() {
    return roomStack.isEmpty() ? null : roomStack.pop();
}
```

Step 4: Implement minOperations Method

The `minOperations` method in the `EscapeRoom` class is a crucial part of solving Problem 1. This method calculates the minimum number of `enterRoom` and `exitRoom` operations required to transform the current order of entered rooms into a target winning order. Let's break down the logic and reasoning behind this method.

Objective

The `minOperations` method in the `EscapeRoom` class calculates the minimum number of `enterRoom` and `exitRoom` operations needed to rearrange the current sequence of rooms (`enteredRooms`) into a target sequence (`target`).

Logic

1. **Iterate Through `enteredRooms`**: Compare each room in `enteredRooms` with the corresponding room in `target`.
2. **Count Mismatches**: If a room in `enteredRooms` doesn't match its counterpart in `target`, it's either out of order or not in the target sequence. In this case, count an `exitRoom` operation.
3. **Remaining Rooms in Target**: After iterating, count any remaining unmatched rooms in `target` as `enterRoom` operations.
4. **Use of a Flag (`flag`)**:
    - The code introduces a boolean flag (`flag`) to track when the alignment between `enteredRooms` and `target` breaks. Once this flag is set to `true`, the code increments the `operations` for every subsequent room in `enteredRooms`, indicating `exitRoom` operations are required.

Complexity

The method has a time complexity of O(N), with N being the length of `enteredRooms`, because it involves a single linear traversal of the `enteredRooms` array.

Example

For `minOperations(["A", "B", "C"], ["A", "C", "B"])`, the method would count two `exitRoom` operations (for "C" and "B") and two `enterRoom` operations (to re-enter "B" and "C" in the correct order), totaling four operations.

This approach efficiently minimizes the operations by only considering necessary adjustments to achieve the target sequence.

Example

Consider `minOperations(["A", "B", "C"], ["A", "C", "B"])`.

- First, we match "A" in both arrays. No operation needed here.
- Then, we find "C" in `enteredRooms` but expect "B" according to `target`. This mismatch requires two `exitRoom` operations (to remove "C" and then "B").
- Finally, we need to `enterRoom` for "B" and "C" in the correct order.

Thus, the total number of operations is 4.

This method demonstrates a strategic approach to problem-solving, balancing between analyzing the current state and working towards the desired state with minimal steps. It's a fine example of applying algorithmic thinking to a practical scenario.

```
// Complexity = O(N) where N is the number of rooms in enteredRooms
    public int minOperations(String[] target, String[] enteredRooms) {
        int operations = 0;
        int targetIndex = 0;
        // Remove extra or out-of-order rooms
        boolean flag = false;
        for (String room : enteredRooms) {
            if (!flag && targetIndex < target.length && room.equals(target[targetIndex])) {
                targetIndex++;
            } else {
                operations++; // Need to exit this room
                flag = true;
            }
        }

        // Add missing rooms from the target
        operations += (target.length - targetIndex); // Remaining rooms to enter

        return operations;
    }
```

Step 5: Main Method and Testing

Finally, write a main method to test these operations.

```
public class EscapeRoomTest {
    public static void main(String[] args) {
        EscapeRoom escapeRoom = new EscapeRoom();

        // Testing enterRoom
        escapeRoom.enterRoom("A");
        escapeRoom.enterRoom("B");
        escapeRoom.enterRoom("C");

        // Testing exitRoom
        System.out.println(escapeRoom.exitRoom()); // Outputs "C"

        // Testing minOperations
        System.out.println(escapeRoom.minOperations(new String[]{"A", "B", "C"}, new String[]{"A", "B"})); // Outputs 1
        System.out.println(escapeRoom.minOperations(new String[]{"A", "B", "C"}, new String[]{"A", "B", "C", "D"})); // Outputs 1
        System.out.println(escapeRoom.minOperations(new String[]{"A", "B", "C"}, new String[]{"A", "C", "B"})); // Outputs 4
    }
}
```

Explanation

- **enterRoom**: Simply pushes a room onto the stack.
- **exitRoom**: Pops the top room from the stack, or returns null if empty.
- **minOperations**: Calculates the number of extra rooms to exit and rooms left to enter to match the target order. It's a linear scan through `enteredRooms`, thus O(N).

The `EscapeRoom` class demonstrates basic stack operations and algorithmic thinking to solve the given problem. This code should be saved in `EscapeRoom.java` as per the problem's instructions.

Certainly! Here are some test cases that students can use to test the `EscapeRoom` class. Each test case includes the input (the sequence of method calls and their parameters) and the expected output.

Test Case 1:

- **Input:**
  - `enterRoom("A")`
  - `enterRoom("B")`
  - `enterRoom("C")`
  - `exitRoom()`
  - `minOperations(new String[]{"A", "B", "C"}, new String[]{"A", "B"})`
- **Expected Output:**
  - `exitRoom()`: "C"
  - `minOperations(...)`: 1

Test Case 2:

- **Input:**
  - `enterRoom("X")`
  - `enterRoom("Y")`
  - `exitRoom()`
  - `exitRoom()`
  - `minOperations(new String[]{"X", "Y", "Z"}, new String[]{"X", "Y"})`
- **Expected Output:**
  - `exitRoom()`: "Y"
  - `exitRoom()`: "X"
  - `minOperations(...)`: 1

Test Case 3:

- **Input:**
  - `enterRoom("Room1")`
  - `enterRoom("Room2")`
  - `minOperations(new String[]{"Room1", "Room2", "Room3"}, new String[]{"Room1", "Room2", "Room4"})`
- **Expected Output:**
  - `minOperations(...)`: 2

Test Case 4:

- **Input:**
  - `enterRoom("Alpha")`
  - `enterRoom("Beta")`
  - `enterRoom("Gamma")`
  - `minOperations(new String[]{"Alpha", "Beta", "Gamma", "Delta"}, new String[]{"Alpha", "Beta", "Gamma"})`
- **Expected Output:**
  - `minOperations(...)`: 1

Test Case 5:

- **Input:**
  - `enterRoom("1")`

- `enterRoom("2")`
- `enterRoom("3")`
- `exitRoom()`
- `exitRoom()`
- `minOperations(new String[]{"1", "2", "3", "4"}, new String[]{"1"})`

- **Expected Output:**
  - `exitRoom()`: "3"
  - `exitRoom()`: "2"
  - `minOperations(...)`: 3

These test cases cover various scenarios, including entering and exiting rooms in different orders and calculating the minimum operations required to achieve a target room sequence. Students can use these to ensure their implementation of `EscapeRoom` works as expected.

Full code : https://paste.ubuntu.com/p/d5DqF9FNsT/

Problem 2, 3

↳ *2 cells hidden*