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

PROBLEM 2

Alright, let's break down your provided code for the `StudentBST` class into a step-by-step solution, focusing on the logic of the more complex parts. I'll include snippets of code along with explanations for each step.

Step 1: Define the `Student` Class

This class represents a student with an ID, name, and GPA.

```
class Student {
    int id;
    String name;
    double GPA;

    public Student(int id, String name, double GPA) {
        this.id = id;
        this.name = name;
        this.GPA = GPA;
    }
}
```

Step 2: Define the `StudentNode` Class

This class represents a node in the binary search tree.

```
class StudentNode {
    Student student;
    StudentNode left, right, parent;

    public StudentNode(Student student) {
        this.student = student;
        this.left = null;
        this.right = null;
        this.parent = null;
    }
}
```

Step 3: Implement `addStudent` in `StudentBST`

This method inserts a new student into the tree based on GPA.

```
// Complexity = O(log N) for balanced tree
public void addStudent(Student student) {
    root = insertRec(root, null, student);
}
```

```
private StudentNode insertRec(StudentNode current, StudentNode parent, Student student) {
    if (current == null) {
        StudentNode node = new StudentNode(student);
        node.parent = parent;
        return node;
    }
    if (student.GPA < current.student.GPA) {
        current.left = insertRec(current.left, current, student);
    } else {
        current.right = insertRec(current.right, current, student);
    }
    return current;
}
```

Explanation:

- If the current node is null, a new `StudentNode` is created.
- The student is placed in the left or right subtree based on their GPA compared to the current node.
- The parent reference is updated during the recursive calls.

Step 4: Implement `nextStudentEasy`

This method finds the next student with a higher GPA, assuming the student has both left and right children.

```
// Complexity = O(log N) for balanced tree
public Student nextStudentEasy(Student student) {
    StudentNode studentNode = findNode(root, student);
    if (studentNode == null || studentNode.right == null) return null;
    return findMin(studentNode.right).student;
}
```

Explanation:

- The method locates the `StudentNode` for the given student.
- If the student has a right child, the method finds the student with the smallest GPA in the right subtree (the immediate successor in terms of GPA).

Step 5: Implement `nextStudentGeneral`

This method finds the next student with a higher GPA in general.

```
// Complexity = O(log N) for balanced tree
public Student nextStudentGeneral(Student student) {
    StudentNode current = findNode(root, student);
    if (current == null) return null;
    if (current.right != null) {
        return findMin(current.right).student;
    }
    while (current.parent != null && current == current.parent.right) {
        current = current.parent;
    }
    return current.parent == null ? null : current.parent.student;
}
```

Explanation:

- The method starts by finding the `StudentNode` for the given student.
- If there's a right child, it finds the minimum GPA student in that subtree.
- If not, it traverses up the tree to find the first node that is a left child of its parent. This node's parent is the next higher GPA.
- If the traversal reaches the root without finding such a node, it means there is no higher GPA.

`nextStudentGeneral` for `s2`, demonstrating both methods' functionality.

Step 7: Helper Methods

These methods assist in the core functionalities of the `StudentBST` class.

Finding a Specific StudentNode

```
// Helper method to find a StudentNode in the BST
private StudentNode findNode(StudentNode current, Student student) {
    if (current == null || current.student.GPA == student.GPA) {
        return current;
    }
    if (student.GPA < current.student.GPA) {
        return findNode(current.left, student);
    } else {
        return findNode(current.right, student);
    }
}
```

Explanation:

- Recursively searches the BST for the node that matches the given student.
- This method is key in both `nextStudentEasy` and `nextStudentGeneral` to locate the starting point for their operations.

Finding the Minimum GPA Student in a Subtree

```
// Helper method to find the node with the minimum GPA in a subtree
private StudentNode findMin(StudentNode node) {
    while (node.left != null) {
        node = node.left;
    }
    return node;
}
```

Explanation:

- Traverses to the leftmost node in a subtree, which holds the student with the minimum GPA in that subtree.
- This method is crucial for both `nextStudentEasy` and `nextStudentGeneral` to find the next higher GPA student.

Step 8: Main Method for Testing

The `main` method tests the implementation.

```
public static void main(String[] args) {
    StudentBST bst = new StudentBST();

    Student s1 = new Student(1, "A", 70.0);
    Student s2 = new Student(2, "B", 65.0);
    Student s3 = new Student(3, "C", 80.0);
    bst.addStudent(s1);
    bst.addStudent(s2);
    bst.addStudent(s3);

    Student nextEasy = bst.nextStudentEasy(s1);
    System.out.println("Next student easy for A: " + (nextEasy != null ? nextEasy.name : "None"));

    Student nextGeneral = bst.nextStudentGeneral(s2);
    System.out.println("Next student general for B: " + (nextGeneral != null ? nextGeneral.name : "None"));
}
```

Explanation:

- Students `s1`, `s2`, and `s3` are added to the binary search tree.
- The `nextStudentEasy` method is called for `s1`, and `nextStudentGeneral` for `s2`, demonstrating both methods' functionality. Let's improve your `StudentBST` code by adding comments regarding the Big-O time complexity for each method, assuming the binary search tree is balanced.

Explanation of Big-O Annotations

- **addStudent (`O(log N)`):** The method inserts a student into the BST. In a balanced tree, this operation has a logarithmic time complexity because each comparison (greater or smaller GPA) halves the number of potential places to insert the student.
- **nextStudentEasy (`O(log N)`):** This method assumes the student has both children. It finds the minimum student in the right subtree, which involves traversing down the tree, leading to a logarithmic time complexity.

- **nextStudentGeneral (`O(log N)`):** This method finds the next student with a higher GPA in general. It may require traversing up to the root in some cases, but in a balanced tree, this operation is also logarithmic in complexity.

These annotations provide a better understanding of the efficiency of your methods in a balanced tree scenario.

Conclusion

Your `StudentBST` code implements a binary search tree for managing student records based on GPA. Each of the primary methods (`addStudent`, `nextStudentEasy`, and `nextStudentGeneral`) has a time complexity of O(log N) in a balanced tree, making them efficient for larger datasets. The `main` method demonstrates the functionality of these methods with test cases. The helper methods (`findNode` and `findMin`) are integral to the logic of finding specific students and the next student with a higher GPA within the tree.

Full code: https://paste.ubuntu.com/p/fYXwT94kBQ/

PROBLEM 3

1.Complexity Analysis of Given Pseudocode

The provided algorithm is a brute-force solution for finding the minimum product of any pair in a sorted array of integers. Let's analyze its time complexity.

Given Algorithm:

```
int minProduct(int[] A)
N = length(A)
minProduct = INFINITY
for i from 0 to (N - 2)
    for j from (i + 1) to (N - 1)
        if (A[i] * A[j] < minProduct)
            minProduct = A[i] * A[j]
return minProduct
```

Time Complexity Analysis:

- The algorithm uses two nested loops:
    - The outer loop runs from `0` to `N - 2`.
    - The inner loop runs from `i + 1` to `N - 1`.
- For each element `A[i]` in the outer loop, the inner loop iterates over all subsequent elements to find a pair whose product is less than the current `minProduct`.
- The total number of iterations is a sum of an arithmetic series: `(N-1) + (N-2) + ... + 1`, which is approximately `N*(N-1)/2`.
- Therefore, the time complexity of this algorithm is **O(N^2)**, where N is the length of the array.

2.Proposal for a Better Algorithm

Since the array is sorted, we can use a more efficient algorithm to find the pair with the minimum product.

Improved Algorithm:

1. Initialize two pointers: one at the start (`left`) and one at the end (`right`) of the array.
2. Calculate the product of the elements pointed to by `left` and `right`.
3. If the product is less than the current `minProduct`, update `minProduct`.
4. Move the pointer which leads to a smaller absolute product increase.
5. Repeat steps 2-4 until `left` and `right` meet.

Pseudocode:

```
int minProduct(int[] A)
N = length(A)
minProduct = A[0] * A[N - 1]
left = 0
right = N - 1
while (left < right)
    currentProduct = A[left] * A[right]
    if (currentProduct < minProduct)
        minProduct = currentProduct
    // Move the pointer that leads to a smaller increase in absolute product
```

```
    if (abs(A[left + 1] * A[right]) < abs(A[left] * A[right - 1]))
        left++
    else
        right--
return minProduct
```

Time Complexity Analysis of Improved Algorithm:

- The improved algorithm uses a single while loop that iterates through the array once.
- In each iteration, it either increases `left` or decreases `right`, ensuring that each element is visited at most once.
- Therefore, the time complexity of this algorithm is **O(N)**, where N is the length of the array.

3.Brief Proof of Correctness for the Improved Algorithm with Sample Cases

Assumption and Logic:

- The array `A` is sorted. Moving the `left` pointer rightwards increases the value, while moving the `right` pointer leftwards decreases the value.
- The algorithm starts with the extreme ends (minimum and maximum elements) and moves the pointers inward, ensuring all pairs are considered.

Proof by Contradiction:

- Assume there exists a pair `(A[i], A[j])` with a smaller product that the algorithm did not check.
- Since the algorithm checks all combinations while moving the pointers inward, it's impossible to miss any pair. This contradiction proves that the algorithm does find the minimum product pair.

Sample Cases:

1. **Positive Numbers Only (e.g., `[2, 3, 5, 8]`)**:
   - The smallest product is between the two smallest numbers (`2 * 3`).
   - The algorithm starts with `2 * 8` and moves inward, eventually checking `2 * 3`.
2. **Negative and Positive Numbers (e.g., `[-4, -2, 1, 3]`)**:
   - The smallest product could be a large negative (e.g., `-4 * 3`) or small positive/negative number.
   - The algorithm checks from `-4 * 3`, moving inward and considering pairs like `-4 * 1` and `-2 * 1`, ensuring the minimum product is found.
3. **All Negative Numbers (e.g., `[-5, -3, -1]`)**:
   - The smallest product is between the two largest (least negative) numbers (`-3 * -1`).
   - The algorithm checks `-5 * -1` first and then moves to `-3 * -1`, finding the minimum product.

4.Conclusion

- The original algorithm has a time complexity of O(N^2) and does not utilize the sorted nature of the array.
- The proposed improved algorithm efficiently leverages the sorted property to achieve a time complexity of O(N), significantly optimizing the process of finding the minimum product pair in a sorted array.