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

Group Project (REAL)

Assignment Overview:

This assignment challenges you to design and implement an efficient algorithm to crack a 12-character secret key comprised of the characters 'M', 'O', 'C', 'H', and 'A'. The process involves guessing the key and refining the guesses based on feedback from a provided `guess()` function, which returns the count of matched positions with the correct key. The goal is to discover the secret key with the least number of guesses, demonstrating the application of various algorithmic design paradigms and data structures.

Highlights and Advice for the Secret Key Guesser Assignment:

Highlights:

1. **Clever Use of Feedback**: Unlike typical brute force, your algorithm must use the feedback from `guess()` to eliminate impossible combinations, similar to the game of Mastermind.
2. **Algorithmic Paradigms**: This problem allows you to demonstrate understanding and application of various algorithmic design paradigms such as brute force initially and then optimizing using perhaps a decrease and conquer strategy by eliminating known incorrect options.
3. **Custom Data Structures**: You might need to devise a custom data structure to store partial solutions or guesses that could be efficient in updating and iterating through possible solutions.
4. **Performance Measurement**: Since the number of guesses is a performance metric, you'll need to ensure your solution is not just correct but also efficient.
5. **Practical Application**: This simulates a real-world problem where efficiency and smart computing can save significant resources.

What-to-Avoid:

1. **Brute Force Reliance**: Do not rely solely on brute force; use it as a starting point but quickly employ smarter strategies to minimize the guess count.
2. **Ignoring Feedback**: The `guess()` method's feedback is critical for refining your search. Ignoring it will result in inefficiency.
3. **Complex Data Structures**: While custom data structures are encouraged, avoid overly complex structures that could introduce unnecessary overhead.
4. **Lack of Testing**: Do not overlook thorough testing with different cases, as it's crucial for ensuring your program's robustness and efficiency.
5. **Inefficient Loops**: Avoid nested loops or repetitive checks that can greatly increase the number of operations.

Advice for Writing README.txt:

- **Clarity**: Ensure that the README.txt is clear and concise, providing all necessary information without any ambiguity.
- **Structure**: Organize the content in sections, including installation instructions, usage guidelines, testing procedures, and contact information.
- **Contribution Scores**: Clearly define the roles and contributions of each team member if it's a group project.
- **Video Link**: Include a working link to your video demonstration and ensure it adheres to the length and content requirements.

Advice for Video Presentation:

- **Preparation**: Outline your presentation beforehand and ensure it showcases the assignment's objectives and your implementation.
- **Demonstration**: Show a live demo of your program working, including the output of the `guess()` method and the number of guesses taken.
- **Explanation**: Explain the algorithms and data structures you developed, and discuss the rationale behind your approach.
- **Complexity Analysis**: Provide a brief complexity analysis of your solution in terms of both time and space.
- **Results**: Discuss the results of your testing and how it aligns with the assignment's objectives.
- **Editing**: Edit the video to make it as professional as possible, removing any errors or unnecessary parts.

Next Steps:

Based on the assignment details, the next steps would involve creating the `SecretKeyGuesser` class, developing an algorithm that efficiently utilizes the feedback from the `guess()` method, and ensuring that the code adheres to the specified requirements without using the Java Collection Framework. After the implementation, you would move on to preparing the README.txt, technical report, and video demonstration.

Solution

To develop an efficient solution for the Secret Key Guesser assignment, follow these detailed steps:

Step-by-Step Solution:

1. Understand the Problem

Grasp the nature of the problem you're solving: a modified version of the Mastermind game. Recognize that you have a feedback loop with which to refine your guesses.

2. Plan Your Approach

Determine which paradigms (greedy, divide and conquer, etc.) might help reduce the number of guesses. You need a strategy that smartly narrows down the possibilities after each guess.

3. Set Up the Project

Create your Java project structure, ensuring you have a main class (`SecretKeyGuesser`) and a helper class (`SecretKey`) as described.

4. Implement the SecretKey Class

Though provided, review the `SecretKey` class to understand how it generates feedback. It's crucial for designing your guess strategy.

5. Design Data Structures

Without using the Java Collection Framework, you'll need custom data structures. Consider arrays or your own version of lists to track which characters are confirmed at what positions.

6. Coding the start() Method

This is where your main logic will reside. It will coordinate the guessing process and handle the feedback.

7. Implement the Initial Brute Force Logic

Start with brute force to establish a baseline. Use nested loops to generate all possible combinations. This will give you an appreciation for the complexity and why optimization is necessary.

8. Refine with Feedback Loop

Each guess gives you the number of characters in the correct place. Use this information to avoid repeating incorrect guesses. You could use an array to keep track of potential characters for each position.

9. Optimize Your Guesses

After each guess, update your data structure to reflect the new information. This could mean:

- Locking in characters that are confirmed correct.
- Eliminating characters from positions where they've been ruled out.

10. Develop a Smarter Algorithm

Move beyond brute force by implementing a heuristic or algorithm that makes educated guesses based on previous feedback. This might resemble algorithms used in genetic programming or other optimization techniques.

11. Implement Counter Logic

Maintain a count of how many guesses you've made. Make sure it's incremented properly within your `start()` method.

12. Termination Condition

Ensure your `start()` method terminates when the correct key is guessed (when `guess()` returns 12).

13. Write Utility Functions

Consider writing helper methods for:

- Generating the next guess based on current knowledge.

- Updating your data structures with feedback from `guess()`.

14. Complexity Analysis

Analyze the time complexity of your refined algorithm. You should aim for something significantly more efficient than the initial brute force's factorial time complexity.

15. Testing and Evaluation

Test your program with different secret keys to ensure accuracy and efficiency. Pay attention to how the number of guesses scales with different inputs.

16. Prepare Documentation

Document your code and the logic behind it. Write a README.txt file that includes:

- How to run your program.
- A high-level overview of your approach.
- Contribution scores if this is a group project.
- A link to your video demonstration.

17. Technical Report

Write a technical report that covers:

- The design of your system.
- Detailed explanations of the data structures and algorithms used.
- Complexity analysis for each part of your solution.
- An evaluation section where you detail your testing process and results.

18. Video Demonstration

Create a video that demonstrates your solution. Show the program running and explain the key parts of your code and the algorithmic choices you made.

19. Final Review and Submission

Go through all deliverables to ensure they meet the requirements. Then submit your project as per the instructions provided by your instructor.

This step-by-step approach will help you systematically tackle the Secret Key Guesser assignment, ensuring that you address all aspects from problem understanding to final submission.

Conceptual Algorithm:

The algorithm will use a feedback loop to refine guesses. It will start with a brute force approach and then use the feedback to eliminate impossible combinations.

1. **Initialize a list of all possible characters** ('M', 'O', 'C', 'H', 'A').
2. **Create a 12-character initial guess** using a simple pattern (e.g., 'MMMMMMMMMMMM').
3. **Use the feedback** to narrow down the correct characters for each position.
4. **Iteratively refine the guess** by changing one character at a time in the positions not yet confirmed.

Pseudocode and Code Snippets:

Initial Setup:

```
// Constants for the puzzle
final char[] POSSIBLE_CHARS = {'M', 'O', 'C', 'H', 'A'};
final int KEY_LENGTH = 12;

// Initialize all possible guesses for each position
List<List<Character>> possibleGuesses = new ArrayList<>();
for (int i = 0; i < KEY_LENGTH; i++) {
    possibleGuesses.add(new ArrayList<>(Arrays.asList(POSSIBLE_CHARS)));
}

// The current guess (initially filled with the first possible character)
char[] currentGuess = new char[KEY_LENGTH];
Arrays.fill(currentGuess, POSSIBLE_CHARS[0]);
```

Feedback Loop and Guess Refinement:

```
// Main loop to refine guesses based on feedback
int feedback;
int numberOfGuesses = 0;
do {
    String guess = new String(currentGuess);
    feedback = secretKey.guess(guess);
    numberOfGuesses++;

    if (feedback != KEY_LENGTH) {
        refineGuesses(currentGuess, feedback, possibleGuesses);
    }

} while (feedback != KEY_LENGTH);

// Output the correct key and the number of guesses
System.out.println("Correct key: " + new String(currentGuess));
System.out.println("Number of guesses: " + numberOfGuesses);
```

Refining the Guesses Based on Feedback:

```
private void refineGuesses(char[] currentGuess, int feedback, List<List<Character>> possibleGuesses) {
    // Logic to refine possible guesses based on feedback
    // This would involve iterating over the possibleGuesses list and removing
    // characters that are now known not to be in certain positions
    // Update the currentGuess array with the new information
}
```

Time Complexity Analysis:

- **Brute Force Approach**: Initially, the time complexity is (O(5^{12})), which is the total number of possible combinations.
- **Refinement Loop**: Each iteration of refinement will potentially eliminate a large number of impossible guesses, but it's hard to give an exact complexity without a specific refinement strategy. However, it's safe to say that it will be less than the brute force approach.
- **Feedback Utilization**: The key to reducing complexity lies in how effectively the feedback is used. If each feedback loop eliminates half of the possible characters for any position, we approach a binary search-like improvement.

Final Considerations:

- The algorithm must be efficient in terms of the number of guesses. A truly random approach is not suitable as it doesn't leverage feedback. The more information gained from each guess, the more refined and effective subsequent guesses can be.
- It's essential to track which characters have been confirmed for each position to avoid unnecessary guesses.
- Implementing a systematic way to iterate over the `possibleGuesses` and updating the `currentGuess` is crucial for the algorithm's efficiency.

Remember, the actual implementation of `refineGuesses` will determine the success of your guessing strategy. It should update `currentGuess` based on the feedback, which is the number of characters in the correct positions, and iterate over `possibleGuesses` to

↳ 1 cell hidden