Summary

1. **Describe how Python executes your implementation of fibonacci(). Give an example of how your function runs if started with the call: fibonacci(3).**

When the function fibonacci(3) is called, Python uses a recursive approach to compute the Fibonacci number.

- Call fibonacci(3): This is not a base case, so it returns fibonacci(2) + fibonacci(1).
- Call fibonacci(2): This is not a base case, so it returns fibonacci(1) + fibonacci(0).
    - fibonacci(1) is a base case and returns 1.
    - fibonacci(0) is a base case and returns 0.
    - So, fibonacci(2) returns 1 + 0 = 1.
- fibonacci(1) is a base case and returns 1.
- So, fibonacci(3) returns 1 + 1 = 2.

2. **What is exponential time complexity?**

Exponential time complexity refers to an algorithm whose growth doubles with each addition to the input data set. This means the runtime increases extremely rapidly. Mathematically, an algorithm with exponential time complexity has a runtime proportional to O(2^n), where n is the size of the input. This is typically inefficient for large inputs due to the rapid growth in computation time.

3. **Explain in your own words how "tail recursion" improves the run time for recursive Fibonacci.**

Tail recursion improves the runtime for the recursive Fibonacci function by eliminating the need to keep track of the previous state once a new state is computed. Tail recursion carries the intermediate results through additional parameters, transforming the exponential time complexity into linear time complexity (O(n)).

4. **What do you think about versions of is_balanced() we incrementally developed? How similar or different do you think they are from each other?**

The versions of is_balanced() developed provided different approaches to solving the problem:

- **Version 1** used a simple count method, which was quick but flawed in some edge cases.
- **Version 2** introduced recursion to handle nested parentheses but was still not perfect.
- **Version 3** used a counter to keep track of open and closed parentheses, addressing the previous issues.
- **Version 4** utilized a stack to ensure proper nesting and pairing of parentheses, which is the most robust and accurate method.

Each version built upon the previous one, improving accuracy and handling more complex cases effectively.

**5. Explain in your own words what a recursive descent parser does.**

A recursive descent parser is a way to read and understand a string of text from start to finish by breaking it down into smaller parts based on set rules. It works by using a set of functions where each function knows how to handle a specific part of the text. These functions call each other to handle different parts of the text, especially when dealing with nested or complex structures. This method is useful for understanding programming languages and math expressions, as it ensures everything follows the correct rules.

Self-Reflection

**What did you learn?**

I learned how recursion works and how to leverage the call stack to solve problems. Additionally, I learned various techniques for checking balanced parentheses and the basics of creating a recursive descent parser.

**What went smoothly?**

The implementation of the Fibonacci sequence using recursion and tail recursion went smoothly. Understanding the step-by-step process of recursion and seeing the call stack in action helped solidify my grasp of the concept.

**What was difficult about the content this week?**

Debugging syntax errors and understanding the flow of the parser took some time to get right.

**How will you approach things differently next time?**

I will spend more time planning the structure of my code before diving into implementation. I will use more test cases to validate each function incrementally.

**Do you have any feedback about the content for this week?**

All good.