816. Ambiguous Coordinates

Medium String Backtracking Leetcode Link

Problem Description

The problem requires us to reconstruct original 2-dimensional coordinates from a string that represents the coordinates stripped of all commas, decimal points, and spaces. Given a string `s`, which is a compact representation of the coordinates without specific formatting characters, we should find all possible valid coordinates that could have produced `s` when the formatting was removed. For instance, the original coordinate `(1, 3)` becomes `s = "(13)"` after the removal of the comma and space. An important constraint is that our original representation always avoids unnecessary zeros and never places a decimal point at the beginning, meaning strings like "00", "0.0", "0.00", "1.0", "001", "00.01", or ".1" wouldn't exist in our original representation.

The output should be a list of strings of all possible original coordinates, formatted correctly with a single space after the comma. They can be returned in any order.

Intuition

To approach this solution, we iterate over the string and consider potential places where we could reintroduce a decimal point to create valid numbers. The function `f(i, j)` generates all possible numbers that can be made from the substring `s[i:j]`. It does this by attempting to place the decimal point between any two digits and then checking if the resulting numbers are valid according to our given constraints. Specifically, the function ensures:

1. No unnecessary leading zeros in the left part of the number.
2. No trailing zeros after a decimal point in the right part of the number.

The valid numbers are those where the left part isn't prefixed with a zero (except the case where the left part is a single '0'), and the right part isn't suffixed with a zero. If the right part is empty, it signifies that we're considering an integer without a decimal point, which is always valid as long as it doesn't start with a zero.

The main part of the solution involves considering every possible way to split `s` into two parts that could represent the `x` and `y` coordinates. This is achieved by iterating over the length of the string and using two nested loops—one to generate possible `x` coordinates and another to generate possible `y` coordinates—while ensuring we respect the original formatting constraints. This combination of possibilities is then formatted into coordinate strings in the form of `"(x, y)"` and added to the final answer list.

Solution Approach

The implementation of the solution involves defining a helper function `f(i, j)`. This function aims to find all valid numbers that could be made from the substring `s[i:j]`. Essentially, for each possible split within this substring, the function checks if the parts on the left and right side of the split can constitute the integer and decimal parts of a number, respectively.

Let's explore this function step-by-step:

- It iterates from `1` to `j - i` (both inclusive), effectively trying out every split.
- `l` and `r` represent the parts of the string before and after the split. These correspond to what could be the integer and decimal parts, respectively, if a decimal point were placed at the split.
- The variable `ok` is a boolean condition that ensures that `l` does not start with a '0' unless it is exactly '0' and that `r` does not end with a '0'.
- If the condition `ok` is satisfied, the method creates a string representation of the number as `l` + `'.'` + `r`, adding the decimal point only if `k < j - i`, meaning there's indeed a decimal part to add.

Once the helper function `f` is defined, the rest of the solution involves iterating through the input string `s`, excluding the parentheses, and trying all possible two-part splits of the string. The main logic loop uses a comprehensive list:

- It iterates over `i` from `2` to `n - 1` (exclusive of `n`), as we have to leave out the opening and closing parentheses from consideration.
- For each `i`, it uses the helper function `f` to generate possible `x` coordinates from `s[1:i]` and possible `y` coordinates from `s[i:n - 1]`.
- It then combines each of these `x` and `y` pairs into the final formatted string `"({x}, {y})"`.

Note that the main solution uses a list comprehension, which provides a concise way to generate the list of all possible coordinate pairs according to the given problem constraints. This approach is efficient because it generates only valid coordinates and does not need to allocate extra memory for invalid permutations.

Overall, the implementation makes use of:

- Helper functions for modular code and reusability.
- Iteration over string indices to try potential splits.
- String slicing to create substrings.
- Conditions to enforce constraints on leading and trailing zeros.
- List comprehensions for concise and readable construction of the output list.

Example Walkthrough

Let's illustrate the solution approach using a small example. Suppose the input string `s` is `"(123)"`. This string represents the stripped-down version of some 2-dimensional coordinates. Here, our task is to find all possible original coordinates that could have produced `s` using the solution approach outlined above.

First, we define our helper function `f(i, j)` which will be used to generate all valid numbers from the substring `s[i:j]`. Then we execute the following steps:

1. Exclude the parentheses and work with the string `123`.

2. We will try to split this string into all possible `x` and `y` coordinates. Since we need at least one number for both the `x` and `y` coordinates, there are essentially two places we can split this string: after "1" and after "2".

3. Now, let's consider splitting after "1": a. `x` would be `1`, and `y` would be `23`. b. We apply the `f()` function to `y`. There are two possibilities: `23` as an integer or `2.3` with a decimal point. c. Our valid coordinates now are `("(1, 23)", "(1, 2.3)")`.

4. Follow a similar process for splitting after "2": a. `x` would be `12`, and `y` would be `3`. b. This time, since `y` is a single-digit number, `3`, there is no room to insert a decimal, so it's only considered as an integer. c. Our valid coordinate here is `("(12, 3)")`.

Putting it all together, for the input string `s = "(123)"`, the function would return the coordinates `("(1, 23)", "(1, 2.3)", "(12, 3)")`.

Notice that we did not create invalid combinations like `("(1.2, 3)")` or `("(1, 2.30)")` because these would have either unnecessary zeros or a decimal at the end, which are against our initial constraints. The solution correctly applies the constraints while splitting the input string and reconstructing original coordinates.

Python Solution

```python
class Solution:
    def ambiguousCoordinates(self, s: str) -> List[str]:
        # Helper function to find all possible decimal representations for a section of the string
        def find_all_representations(start, end):
            representations = []
            for split_point in range(1, end - start + 1):
                left = s[start: start + split_point]
                right = s[start + split_point: end]

                # A valid combination:
                # If left is "0" or does not start with "0", AND
                # right does not end with "0" (No trailing zeros in decimals)
                is_valid = (left == '0' or not left.startswith('0')) and not right.endswith('0')
                if is_valid:
                    # If split_point is at the end, don't add a dot; otherwise, add the dot to form a decimal
                    representations.append(left + ('.' if split_point < end - start else '') + right)
            return representations

        length = len(s)
        result = []
        # Iterate through all possible splits of the input string
        for i in range(2, length - 1):
            # Find all valid representations for the left and right parts as per the current split
            for x in find_all_representations(1, i):
                for y in find_all_representations(i, length - 1):
                    # Combine representations into coordinate format
                    result.append(f'({x}, {y})')

        return result

```

Java Solution

```java
class Solution {

    // This method generates all possible valid coordinates after adding a comma.
    public List<String> ambiguousCoordinates(String s) {
        int length = s.length();
        List<String> results = new ArrayList<>();

        // Loop through the string to find all possible splits for x and y coordinates.
        for (int i = 2; i < length - 1; ++i) {
            // Get all possible valid strings for the first coordinate (x).
            for (String x : getPossibleNumbers(s, 1, i)) {
                // Get all possible valid strings for the second coordinate (y).
                for (String y : getPossibleNumbers(s, i, length - 1)) {
                    // Combine x and y coordinates into a valid point representation.
                    results.add(String.format("(%s, %s)", x, y));
                }
            }
        }
        return results;
    }

    // This helper method returns all possible valid strings that can be formed from a given substring.
    private List<String> getPossibleNumbers(String s, int start, int end) {
        List<String> possibleNums = new ArrayList<>();

        // Split the given range into two parts and check if they form valid numbers.
        for (int k = 1; k <= end - start; ++k) {
            String leftPart = s.substring(start, start + k);
            String rightPart = s.substring(start + k, end);

            // Check if the left part isn't led by '0' unless it's "0" and the right part doesn't end with '0'.
            boolean isValid = ("0".equals(leftPart) || !leftPart.startsWith("0")) && !rightPart.endsWith("0");

            if (isValid) {
                // Add the decimal point if there is a right part, otherwise just add the left part.
                possibleNums.add(leftPart + (k < end - start ? "." : "") + rightPart);
            }
        }
        return possibleNums;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    vector<string> ambiguousCoordinates(string s) {
        int length = s.size(); // Length of the input string
        vector<string> results; // Vector to store all possible coordinates

        // Lambda function to generate all decimal combinations for a given substring [i, j)
        auto generateDecimals = [&](int i, int j) {
            vector<string> decimals;
            for (int k = 1; k <= j - i; ++k) {
                string left = s.substr(i, k);       // Left part of the decimal
                string right = s.substr(i + k, j - i - k); // Right part of the decimal
                // Check if left part is not padded with zeros and right part does not end with zero
                bool isValid = (left == "0" || left[0] != '0') && (right.empty() || right.back() != '0');
                if (isValid) {
                    // Concatenate left and right parts with a dot if the decimal has two parts
                    decimals.push_back(left + (k < j - i ? "." : "") + right);
                }
            }
            return decimals;
        };

        // Iterate over all possible splits of the input string into two parts
        for (int i = 2; i < length - 1; ++i) {
            // Generate all possible decimals for the first part of the string
            for (auto& x : generateDecimals(1, i)) {
                // Generate all possible decimals for the second part of the string
                for (auto& y : generateDecimals(i, length - 1)) {
                    // Combine the two parts into a coordinate and add to the results
                    results.emplace_back("(" + x + ", " + y + ")");
                }
            }
        }
        return results; // Return all possible ambiguous coordinates
    }
};

```

Typescript Solution

```typescript
function ambiguousCoordinates(s: string): string[] {
    // Remove the parentheses from the input string
    s = s.slice(1, s.length - 1);
    const stringLength = s.length;

    /**
     * Generate all possible strings that can be created by inserting a decimal point
     * into the given string if it represents a valid number after the insertion.
     * @param {string} str The string to insert a decimal point into.
     * @return {string[]} An array of strings representing valid numbers.
     */
    const generateDecimals = (str: string): string[] => {
        const possibleDecimals: string[] = [];
        for (let i = 1; i < str.length; i++) {
            const numberWithDecimal = `${str.slice(0, i)}.${str.slice(i)}`;
            // Check if the resulting string is still a valid number
            if (`${Number(numberWithDecimal)}` === numberWithDecimal) {
                possibleDecimals.push(numberWithDecimal);
            }
        }
        // Check if the string itself is a valid number without a decimal point
        if (`${Number(str)}` === str) {
            possibleDecimals.push(str);
        }
        return possibleDecimals;
    };

    const results: string[] = [];
    // Try each possible split of the string into two parts
    for (let i = 1; i < stringLength; i++) {
        // Generate all valid numbers for the left part
        for (const left of generateDecimals(s.slice(0, i))) {
            // Generate all valid numbers for the right part
            for (const right of generateDecimals(s.slice(i))) {
                // Combine both parts with parentheses and a comma to form coordinates
                results.push(`(${left}, ${right})`);
            }
        }
    }

    return results;
}

```

Time and Space Complexity

The provided code function `ambiguousCoordinates` takes a string `s` and generates a list of strings that represent all possible coordinates that can be formed by inserting a comma and a decimal point inside `s`.

Time Complexity

The time complexity of the function can be analyzed as follows:

- The function `f` is called for every possible partition of the string `s`. In the worst-case scenario, it tries to place a decimal point in every possible position within a substring.
- For a given partition of `s` into two parts (left and right), the `f` function iterates over all potential points to place a decimal in the left part, which can be at most `O(n)`, where `n` is the length of the string `s`.
- Since the decimal can be placed anywhere in the left part, and for every placement of the decimal, a new string representing the coordinate is created, there are `O(n)` such operations in the worst case for each partition.
- The outer loop runs for `n - 2` partitions (from 2 to `n - 1`), so there are `O(n)` possible partitions.

Combining these insights, we have:

- The function `f` has a complexity of `O(n)` for each call (considering string slicing and concatenation operations).
- It is called twice for each of `O(n)` partitions (`O(n)` times for left partition and `O(n)` times for right partition).

This gives us a worst-case time complexity of `O(n^3)`. Mathematically:

```
O(2 * n * n * n) -> O(n^3)
```

Space Complexity

The space complexity can be analyzed as follows:

- The `res` list holds all the possible variations of a partitioned string with and without decimals. This holds at most `O(n)` strings of length `O(n)`.
- Storage for the final list of combinations is needed, which, in the worst case, would store `O(n^2)` coordinates (since for each of the `O(n)` left partitions, there are `O(n)` right partitions).

Thus, we have:

- Storage for `f`'s intermediate results is `O(n^2)`, because there are `O(n)` substrings and each substring has a length of `O(n)`.
- The final list of combinations can store `O(n^2)` strings.

This gives us a worst-case space complexity of `O(n^2)`. Mathematically:

```
O(n^2 + n^2) -> O(n^2)
```