1849. Splitting a String Into Descending Consecutive Values

Medium String Backtracking Leetcode Link

Problem Description

This problem asks us to check whether a given string `s`, comprised exclusively of digits, can be split into two or more non-empty substrings such that:

1. The numerical values of these substrings, when converted from strings to numbers, form a sequence in descending order.
2. The difference between the numerical values of each pair of adjacent substrings is exactly 1.

For example, if `s = "543210"`, it is possible to split it into ["54", "3", "2", "10"], where each number is 1 less than the previous, and so the condition is met.

The key constraints are that the substrings must be non-empty and must appear in the same order they do in the original string. There cannot be any rearrangement of the substrings.

Intuition

To solve this problem, we can use a depth-first search (DFS) algorithm. The DFS approach will try to partition the string `s` at different points and recursively check if the resulting substrates satisfy the conditions.

The solution uses three key ideas:

1. **Recursion**: Recursively split the string and check if the current split creates a number that is exactly 1 less than the preceding number.
2. **Backtracking**: If at any point the condition is not met, the function backtracks, trying a different split.
3. **Early Stopping**: If the given string has been entirely traversed and more than one substring meeting the conditions has been found, we can stop the search and return true.

To implement the solution:

- A recursive function `dfs` is defined, which takes parameters `i` (the current starting index in `s`), `x` (the last numerical value of the substring), and `k` (the count of valid splits found so far).
- Initially, `dfs` is called with starting index 0, `x` set as -1 (as there is no previous number), and `k` as 0 (no splits found yet).
- Within `dfs`, we iterate over the string starting from index `i`, attempting to form a new substring from `s[i]` to `s[j]`.
- We check if our substring satisfies the conditions; if it does, we continue the search by recursively calling `dfs` with the next starting index (`j+1`), the new number `y`, and increment `k`.
- If at any point, a full pass through the string is made (`i` equals the length of `s`) and more than one valid substring (`k > 1`) has been found, the function returns true, indicating the splitting is possible. Otherwise, the function eventually returns false.

Using DFS, this solution effectively searches for all possible substrings that might fit the criteria, and upon finding the first valid series of splits, it returns true. If no valid series is found by the time the entire string has been traversed, it returns false.

Solution Approach

The implementation of the solution revolves around a classic DFS algorithm. Here's a step-by-step explanation of the approach using the given Python code:

1. **Initialization**: A helper function, `dfs`, is defined inside the main class `Solution`. This recursive function is at the heart of the DFS implementation. It aims to try all possible splits of the string `s` beginning from index `i`.

2. **Base Case**: If `dfs` reaches the end of the string (`i == len(s)`), it checks whether at least two substrings with the required properties have been found (`k > 1`). If so, the function returns `True`; otherwise, `False`.

3. **Recursive Search**: Inside the `dfs` function, starting from index `i`, the code attempts to build a substring piece by piece by iterating from `i` to each `j` within the loop. The goal is to form a numerical value `y` of the current substring being considered (`s[i:j+1]`).

4. **Conditions Check**: After each additional character is included in `y`, we check if `y` is exactly one less than the previous number `x`. If `x` is -1 (indicating this is the first number), any `y` is acceptable (since there is no prior number to compare with).

   - If this condition is satisfied, the function immediately proceeds with a recursive call, `dfs(j + 1, y, k + 1)`, where `j + 1` is the new starting index for the next substring, `y` is now the last number, and `k + 1` indicates one more valid split has been found.
   - If the returned value from this recursive call is `True`, indicating that proceeding from index `j + 1` has led to a valid sequence, the current `dfs` also returns `True`.

5. **Continuation and Backtracking**: If the condition fails or the recursive call doesn't result in a valid solution, the function continues to the next iteration of the loop, effectively trying a longer substring (more characters included from `s`) or backtracking if all possibilities starting from index `i` have been exhausted.

6. **Return Result**: The `dfs` function will return `False` if none of the iterations and recursive calls result in a valid split sequence. On the other hand, the function will exit with a `True` at the first instance of finding a valid sequence of substrings.

7. **Main Function Call**: To start the process, the `splitString` method in the `Solution` class calls `dfs(0, -1, 0)` indicating it starts looking for splits from the beginning of the string (`i = 0`), with no prior number (`x = -1`), and no splits found yet (`k = 0`).

This algorithm effectively explores all potential ways to split the string through DFS while avoiding a full search when the first valid solution is found. It also uses recursion and backtracking systematically to traverse the decision tree until it either finds a valid sequence of splits or exhausts all possibilities and determines it's not possible.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach with the string `s = "321"`. This string should be possible to split into a sequence of descending numbers that are each 1 less than the previous, specifically into ["3", "2", "1"].

Initial Call:

- The `splitString` method calls `dfs(0, -1, 0)` to start the recursive search.

Action in `dfs` - First Iteration from index `0`:

- We try to form substrings starting from the first character.
- The loop runs from `i = 0` to the end of the string.
- We start by picking `s[0:1]` which is "3", and convert it to number `y = 3`.
- Since the last number `x` was -1 (indicating no previous number), "3" is accepted as the first valid substring.
- We then call `dfs(1, 3, 1)` to proceed to the next character.

Action in `dfs` - Second Iteration from index `1`:

- Now `x = 3` and we start from `i = 1`.
- We pick `s[1:2]` which is "2", and convert it to number `y = 2`.
- We check if `y` is exactly 1 less than `x` (3 - 1 = 2), and it is.
- We then call `dfs(2, 2, 2)` since we have found two substrings "3" and "2".

Action in `dfs` - Third Iteration from index `2`:

- Now `x = 2`, and `i = 2`.
- We pick `s[2:3]` which is "1", and convert it to number `y = 1`.
- Again, we check if `y` is exactly 1 less than `x` (2 - 1 = 1), which it is.
- We now call `dfs(3, 1, 3)` because we have found a third substring "1".

Base Case Reached:

- The call `dfs(3, 1, 3)` has `i == len(s)`, meaning we have reached the end of the string.
- The count of valid splits `k` is 3 (greater than 1), so we return `True`.

Result:

- The valid sequence resulting from the above splits is ["3", "2", "1"], and therefore the initial call of `splitString` returns `True`.
- We have successfully split the string into a descending sequence where each number is exactly 1 less than the previous substring.

This simple example illustrates the DFS-based recursive process of finding valid substrings and how backtracking occurs if a substring does not fit the required criteria. The process continues until the entire string is either successfully split or determined to be unsplittable according to the conditions.

Python Solution

```python
class Solution:
    def split_string(self, s: str) -> bool:
        # Helper function to perform depth-first search
        def dfs(current_index, previous_number, split_count):
            # Base condition: Check if we have reached the end of the string
            if current_index == len(s):
                # Valid split if at least one number has been split before
                return split_count > 1

            # Variable to store the current number being formed
            current_number = 0
            for j in range(current_index, len(s)):
                # Accumulate the digits to form the current number
                current_number = current_number * 10 + int(s[j])

                # Check if it's the start (-1) or if the current number is exactly
                # one less than the previous number, and then recursively call dfs
                if (previous_number == -1 or previous_number - current_number == 1) and dfs(j + 1, current_number, split_count + 1):
                    # If the recursive call returns True, propagate it upwards
                    return True

            # If we cannot find a valid split, return False
            return False

        # Initiate the depth-first search with initial values
        return dfs(0, -1, 0)

```

Java Solution

```java
class Solution {
    private String str; // Variable to hold the input string

    // Method to check if we can make a split where each number is one less than the previous
    public boolean splitString(String s) {
        this.str = s; // Assign the given string to the class variable
        // Begin depth-first search from the start of the string, with initial value -1 and no splits
        return dfs(0, -1, 0);
    }

    // Recursive DFS method to check for valid splits
    // i: starting index for the next number
    // lastNumber: the last number found in the split
    // splitCount: number of valid splits found so far
    private boolean dfs(int i, long lastNumber, int splitCount) {
        // If we have reached the end of the string, check if we made more than one valid split
        if (i == str.length()) {
            return splitCount > 1;
        }

        long currentNumber = 0; // Variable to store the current number being formed
        // Iterate over the string to form the next number
        for (int j = i; j < str.length(); ++j) {
            // Append the next digit to the current number
            currentNumber = currentNumber * 10 + (str.charAt(j) - '0');
            // Check if the current number is 1 less than the last number or if this is the first number
            if ((lastNumber == -1 || lastNumber - currentNumber == 1) &&
                // Continue the DFS search from the next index, with the current number as the last number
                dfs(j + 1, currentNumber, splitCount + 1)) {
                // If a valid split is found, return true
                return true;
            }
        }
        // If no valid split is found, return false
        return false;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to check if the input string can be split into consecutive decreasing integers.
    bool splitString(string s) {
        // Lambda function to perform Depth First Search (DFS) to find if the string can be split.
        // `start` is the index to start searching from, `prevValue` is the previous value found,
        // and `count` is the number of splits made so far.
        function<bool(int, long long, int)> dfs = [&](int start, long long prevValue, int count) -> bool {
            // Base case: if we have reached the end of the string and made at least one split.
            if (start == s.size()) {
                return count > 1;
            }
            // The current value being formed.
            long long currentValue = 0;
            // Iterate through the string starting from `start`.
            for (int j = start; j < s.size(); ++j) {
                // Forming the current value.
                currentValue = currentValue * 10 + (s[j] - '0');
                // To prevent overflow, break if the number is too large.
                if (currentValue > 1e10) {
                    break;
                }
                // If this is the first number or if the current number is exactly
                // one less than the previous, recursively call dfs.
                if ((prevValue == -1 || prevValue - currentValue == 1) &&
                    dfs(j + 1, currentValue, count + 1)) {
                    // If dfs call is successful, return true indicating that the string
                    // can be split according to the problem requirements.
                    return true;
                }
            }
            // If no valid split could be found, return false.
            return false;
        };

        // Call the dfs starting from index 0 with an undefined previous value (-1) and no splits (0 count).
        return dfs(0, -1, 0);
    }
};

```

Typescript Solution

```typescript
// Type alias for the depth-first search function type.
type DFSFunction = (start: number, prevValue: number, count: number) => boolean;

// Function to perform the Depth First Search (DFS) to find if the string can be split
// into consecutive decreasing integers.
const depthFirstSearch: DFSFunction = (start, prevValue, count) => {
    // Base case: if we have reached the end of the string and made at least one split.
    if (start === s.length) {
        return count > 1;
    }

    // The current value being formed.
    let currentValue = 0;
    // Iterate through the string starting from start.
    for (let j = start; j < s.length; j++) {
        // Forming the current value.
        currentValue = currentValue * 10 + (s[j].charCodeAt(0) - '0'.charCodeAt(0));
        // To prevent overflow, break if the number is too large.
        if (currentValue > 1e10) {
            break;
        }
        // If this is the first number or if the current number is exactly
        // one less than the previous, recursively call depthFirstSearch.
        if ((prevValue === -1 || prevValue - currentValue === 1) &&
            depthFirstSearch(j + 1, currentValue, count + 1)) {
            // If dfs call is successful, return true indicating that the string
            // can be split according to the problem requirements.
            return true;
        }
    }
    // If no valid split could be found, return false.
    return false;
};

// Function to check if the input string can be split into consecutive decreasing integers.
const splitString = (s: string): boolean => {
    // Call the depthFirstSearch starting from index 0 with an undefined previous value (-1) and no splits (0 count).
    return depthFirstSearch(0, -1, 0);
};

// Example usage
let s: string = "1234";   // Example input string
console.log(splitString(s)); // It will log the output of the splitString function.

```

Time and Space Complexity

The given Python code is a solution to the problem of splitting a string into a sequence of decreasing consecutive numbers. It uses a depth-first search (DFS) approach to recursively try out different splits.

Time Complexity

The DFS function, `dfs`, is called recursively, at each step it tries to construct a new number, `y`, by adding digits one by one from the input string `s`. For each number `y` formed, a check is made whether the previous number `x` is exactly one more than `y`(i.e., `x - y == 1`). The function `dfs` is potentially called once for each new number `y` formed, which can happen for each starting position `i` in the string `s`.

The worst case time complexity is `O(n * 2^n)`, where `n` is the length of the string. This is because at each step, we have two choices: either include the current digit in the current number `y` or start a new number beginning with the current digit. We make `n` choices at most once for each of the `2^n` potential splits of the string.

Space Complexity

The space complexity of the code is `O(n)`. This comes from two factors:

1. The recursive call stack, which at maximum depth could be `O(n)`, as in the worst case the recursion could go as deep as the length of the string for a split starting at each digit of the string.
2. The variable `y` in the inner loop, which is re-constructed for each recursive call but doesn't add to the space complexity as it is just an integer and not a recursive structure.

Hence, the space requirement grows linearly with the input size, dominated by the depth of the recursive calls.