2468. Split Message Based on Limit

Hard String Binary Search Leetcode Link

Problem Description

You're given a string called `message` and a positive integer called `limit`. Your task is to divide `message` into one or more parts such that:

1. Each part ends with a suffix formatted as `"<a/b>"` where:
   - `a` is the part's index starting at 1.
   - `b` is the total number of parts.
2. The length of each part including its suffix should exactly equal `limit`. For the last part, the length can be at most `limit`.
3. When the suffixes are removed from each part and the parts concatenated, it should form the original `message`.
4. The solution should minimize the number of parts the message is split into.

If the message cannot be split into parts as per the above conditions, the result should be an empty array.

Intuition

For the given problem, we need to figure out how many parts we can divide the message into. The approach involves iterating over possible numbers of parts and checking whether it's feasible to split the message into that many parts, where each part is the length of `limit`.

To arrive at the solution, we first need to identify the potential number of parts that the message can be split into. This depends on the length of `message` and the given `limit`, considering the length of the suffix that each part will have.

Here's the step-by-step reasoning:

1. Determine the maximum number of possible parts by looking at the size of `message` and the `limit`. This is done by iterating from 1 to the length of the message.
2. For each potential number of parts `k`, calculate the total additional characters needed for the suffixes of all `k` parts. This includes the length of the numbers (`a` and `b` in the suffix), as well as the constant characters ('<' , '/', '>', and the slashes).
3. Check if the message can be split into exactly `k` parts where each part is of length `limit`. We do this by ensuring that the total number of characters taken up by the suffixes and the parts does not exceed `k * limit`.
4. If it's possible to divide the message into `k` parts, we construct the parts by taking as much of the message as we can fit into each part (considering the space required by the suffix) and add the appropriate suffix.
5. If we determine that the message can't be divided into any number of parts due to the constraints, we return an empty array.

The core of this approach relies on efficiently calculating the space taken by the suffixes and determining the capability to fit the message's content within the `limit` provided.

Solution Approach

The solution is implemented using a simple `for`-loop which iterates through possible numbers of parts, with helpful comments in the code to explain what is happening. The algorithm uses string manipulation and arithmetic calculations to determine the feasibility of each potential split. Here is how the approach is executed, explained step by step:

1. **Iterate through the potential number of parts**: The for-loop begins with `k = 1` and goes up to `n + 1` (inclusive), where `n` is the length of the message. `k` represents the current candidate for the total number of parts that `message` could be split into.
2. **Calculate additional characters for suffixes**: Variables `sa`, `sb`, and `sc` are used to track the total length of all suffixes combined. `sa` accumulates the lengths of the number parts (`a` and `b`) in the suffixes. `sb` takes into account the repeated occurrence of the lengths of `b` for each part. `sc` accounts for the constant characters in the suffix (`<`, `>`, `/`) for each part.
   - For each part, there are exactly three constant characters, hence `sc = 3 * k`.
   - The lengths of `a` and `b` can be different because as `k` increases, `b` may become a larger number with more digits. So, for each possible number of parts, we need to incrementally update `sa`.
3. **Check feasibility**: We check if subtracting the length of all the suffixes for `k` parts from `limit * k` is still greater than or equal to the length of the message (`limit * k - (sa + sb + sc) >= n`). This ensures that there is enough room to fit the message alongside the suffixes.
4. **Construct the parts**: If it is possible to split the message for the current `k`, we create a list `ans` that will hold all parts. We then loop from 1 to `k`, for each part calculating its specific tail (e.g., `<1/3>`, `<2/3>`, etc.), and concatenate the corresponding slice of `message` with the tail to form the part. The message slice starts at index `i` and captures enough characters to fill the part up to `limit` when the tail is considered. After each part is constructed, `i` is incremented by the number of characters consumed from `message`.
5. **Return Result**: If a successful split is found, the list `ans` is returned, containing all the parts properly suffixed. If no valid split is found after trying all possible `k`s, an empty list is returned.

By using this approach, the algorithm efficiently identifies the minimum `k` that can be used to satisfy the problem's constraints. It avoids unnecessary iterations by stopping immediately once a viable split is found, making it an effective solution for this problem.

Example Walkthrough

Let's assume we have a `message` with the string "LeetCode" and a `limit` of 5. We want to apply the solution approach to this problem.

1. **Iterate through the potential number of parts**: We know the `message` is 8 characters long. We start our for-loop with `k = 1`, which signifies that we initially try to fit the message into just one part, and will proceed to try 2, 3, etc., if one part doesn't work.
2. **Calculate additional characters for suffixes**: If we tried to fit the message into one part, the suffix would be "<1/1>". Since the suffix contains five characters and the `limit` is 5, it would be impossible to fit any part of the message because the entire limit is used by the suffix alone. Therefore, we cannot split the message into just one part with these constraints.
3. **Check feasibility**: We continue iterating over `k`. The next value is `k = 2`. This time, our suffixes would be "<1/2>" and "<2/2>". Each of these has five characters, so each part of the message can be at most 0 characters long, which is again not feasible since we have an 8-character message to split.
4. **Construct the parts**: Keep iterating. When `k = 3`, the suffixes will be "<1/3>", "<2/3>", and "<3/3>". Each of these suffixes has five characters. This would allow each part to contain exactly 0 characters from `message` which is still not feasible.
5. **Return Result**: We keep trying different `k` values. With `k = 4`, the suffixes will be "<1/4>", "<2/4>", "<3/4>", and "<4/4>". Now, each suffix has five characters, so we can fit exactly 0 characters from `message` in each part, which still does not work.

Continuing this process, we finally arrive at `k = 7`. The suffixes here would be "<1/7>", "<2/7>", ..., to "<7/7>". Now let's calculate:

- Each part's suffix has five characters.
- For `k = 7`, this means each part can hold exactly 0 characters of the message since the `limit` is 5, and the suffix itself uses all 5 characters.

It's evident that we cannot split "LeetCode" into parts of length 5 following the rules since the suffixes alone consume the whole limit. Thus, following the solution approach, we would return an empty array because there's no way to split "LeetCode" with a limit of 5 such that each part includes a suffix and respects the limit.

However, if the `limit` were increased, such as to a `limit` of 10, we would be able to calculate a feasible k and split the message accordingly. For the `limit` of 5 given in this example, since no parts can be constructed that meet the constraints, we are left with no solution.

Python Solution

```python
class Solution:
    def splitMessage(self, message: str, limit: int) -> List[str]:
        # Calculate the length of the message
        message_length = len(message)
        # Initialize sum of lengths of all suffixes
        suffix_length_sum = 0

        # Iterate through the possible number of parts to split the message into
        for parts_count in range(1, message_length + 1):
            # Increment the sum of lengths of suffixes by the length of the current suffix
            suffix_length_sum += len(str(parts_count))
            # Calculate the total length of suffixes for the current number of parts
            total_suffix_length = len(str(parts_count)) * parts_count
            # Calculate the total length of separators needed for all parts ("<", "/", ">", for each part)
            separators_length = 3 * parts_count
            # Check if the message can fit into the specified limit when split into current number of parts
            if limit * parts_count - (suffix_length_sum + total_suffix_length + separators_length) >= message_length:
                # Initialize the list to store the resulting split message parts
                splitted_messages = []
                # Start index for slicing the message
                current_index = 0
                # Generate each part with its corresponding suffix
                for part_number in range(1, parts_count + 1):
                    # Create the string suffix for the current part
                    suffix = f'<{part_number}/{parts_count}>'
                    # Calculate and obtain the substring for the current part based on the limit and suffix
                    substring = message[current_index : current_index + limit - len(suffix)] + suffix
                    # Add the part to the list of split messages
                    splitted_messages.append(substring)
                    # Update the current index to the starting index of the next part
                    current_index += limit - len(suffix)
                # Return the list of split message parts
                return splitted_messages
        # Return an empty list if the message cannot be split within the limit
        return []
```

Java Solution

```java
class Solution {
    public String[] splitMessage(String message, int limit) {
        int messageLength = message.length(); // Length of the original message.
        int sumOfDigits = 0; // To keep track of the sum of the digits of all parts.

        // Initialize the array to hold the split message parts.
        String[] answer = new String[0];

        // Looping over the possible number of parts from 1 to messageLength.
        for (int parts = 1; parts <= messageLength; ++parts) {
            // Length of digits in the current part number.
            int lengthOfCurrentPartDigits = Integer.toString(parts).length();
            // Update the sum of the digits with current part number's digit length.
            sumOfDigits += lengthOfCurrentPartDigits;

            // Total length consumed by the digit parts.
            int totalDigitsLength = lengthOfCurrentPartDigits * parts;
            // Total length consumed by the delimiters "<" and "/>".
            int totalDelimiterLength = 3 * parts;

            // Check if the current breakup fits into the limits.
            if (limit * parts - (sumOfDigits + totalDigitsLength + totalDelimiterLength) >= messageLength) {
                int currentIndex = 0; // Start index for the substring.
                answer = new String[parts]; // Initialize the answer array with the number of parts.

                // Split the message into the determined number of parts.
                for (int part = 1; part <= parts; ++part) {
                    // Generate the tail string for the current part.
                    String tail = String.format("<%d/%d>", part, parts);
                    // Calculate the end index for the substring; it's either the end of the message or the max allowed by the limit,
                    int endIndex = Math.min(messageLength, currentIndex + limit - tail.length());
                    // Create the substring for the current part, add the tail, and store it in the answer array.
                    String splitPart = message.substring(currentIndex, endIndex) + tail;
                    answer[part - 1] = splitPart;
                    // Update the start index for the next part.
                    currentIndex += limit - tail.length();
                }
                // Everything fitted perfectly, break out of the loop.
                break;
            }
        }
        // Return the split message parts.
        return answer;
    }
}
```

C++ Solution

```cpp
#include <string>
#include <vector>
using namespace std;

class Solution {
public:
    vector<string> splitMessage(string message, int limit) {
        int messageLength = message.size(); // Total length of the message
        int sumOfDigits = 0; // Sum of the digits of the message parts
        vector<string> splitMessages; // Store the resulting split messages

        // Iterate through the possible number of message parts
        for (int partCount = 1; partCount <= messageLength; ++partCount) {
            int lengthOfDigits = to_string(partCount).size(); // Length of the digits in this part
            sumOfDigits += lengthOfDigits; // Update the sum of digits
            int totalDigitsLength = lengthOfDigits * partCount; // Total length of all digits in all parts
            int totalSeparatorsLength = 3 * partCount; // Total length of separators (i.e., "<>/<>" part)

            // Check if splitting the message into 'partCount' parts is possible within the limit
            if (partCount * limit - (sumOfDigits + totalDigitsLength + totalSeparatorsLength) >= messageLength) {
                int currentIndex = 0; // Current position in the message for split

                // Construct each message part and add to the splitMessages vector
                for (int partIndex = 1; partIndex <= partCount; ++partIndex) {
                    string tail = "<" + to_string(partIndex) + "/" + to_string(partCount) + ">"; // The part indicator
                    // Substring from the current index to the maximum allowed length minus tail size
                    string part = message.substr(currentIndex, limit - tail.size()) + tail;
                    splitMessages.emplace_back(part); // Add the constructed part to the result

                    currentIndex += limit - tail.size(); // Move the current index forward
                }
                break; // Once the message has been split successfully, exit the loop
            }
        }
        return splitMessages; // Return the split messages
    }
};
```

Typescript Solution

```typescript
// TypeScript syntax does not use include statements like C++, imports are done differently.

// Function to split a long message into multiple parts with a specific length limit
function splitMessage(message: string, limit: number): string[] {
    const messageLength = message.length; // Total length of the message
    let sumOfDigits = 0; // Sum of the digits of the message parts
    let splitMessages: string[] = []; // Store the resulting split messages

    // Iterate through the possible number of message parts
    for (let partCount = 1; partCount <= messageLength; ++partCount) {
        const lengthOfDigits = partCount.toString().length; // Length of the digits in this part
        sumOfDigits += lengthOfDigits; // Update the sum of digits
        const totalDigitsLength = lengthOfDigits * partCount; // Total length of all digits in all parts
        const totalSeparatorsLength = 3 * partCount; // Total length of separators (i.e., "<>/<>" part)

        // Check if splitting the message into 'partCount' parts is possible within the limit
        if (partCount * limit - (sumOfDigits + totalDigitsLength + totalSeparatorsLength) >= messageLength) {
            let currentIndex = 0; // Current position in the message for split

            // Construct each message part and add to the splitMessages array
            for (let partIndex = 1; partIndex <= partCount; ++partIndex) {
                const tail = `<${partIndex}/${partCount}>`; // The part indicator
                // Substring from the current index to the max allowed length minus tail size
                const part = message.substring(currentIndex, currentIndex + limit - tail.length) + tail;
                splitMessages.push(part); // Add the constructed part to the result

                currentIndex += limit - tail.length; // Move the current index forward
            }
            break; // Once the message has been split successfully, exit the loop
        }
    }
    return splitMessages; // Return the split messages
}

// The given TypeScript function can now be called globally with a string message and a limit.
```

Time and Space Complexity

Time Complexity

The given code snippet computes a way to split a message into multiple parts with a given `limit` on the length of each part including the suffix that indicates the part number and the total number of parts, in the format `<j/k>`.

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

1. The outer for loop runs from `1` to `n + 1` where `n` is the length of the message. In the worst case, this would run `n` times.
2. Inside the loop, there are calculations that take constant time `O(1)` for each iteration, namely computing `len(str(k))`, `sa`, `sb`, and `sc`. These operations do not depend on the size of the input message and are thus constant-time operations.
3. The condition in the `if` statement is checked `k` times, which again takes constant time for each individual check.
4. If the condition is met, a nested loop will construct the message parts. This loop will iterate a maximum of `k` times where `k` is less than or equal to `n`. Each iteration of the inner loop includes slicing the message, which can take up to `O(n)` time, and concatenating strings, which is also `O(n)` in Python since strings are immutable and a new string is created every time concatenation happens.

Given that string concatenation is the most time-consuming operation and it could be performed `k` times within the inner loop, we can consider this operation as `O(kn)`.

However, since `k` is at most `n`, the upper bound on the time complexity of the inner loop is `O(n^2)`.

Hence, the **worst-case time complexity** of the entire function can be stated as `O(n^3)` since the nested for loop is inside another loop which runs `n` times.

Space Complexity

For space complexity, we can consider the following:

1. The list `ans` stores at most `k` strings, and each string could be up to the limit in length.
2. The temporary variables `sa`, `sb`, `sc`, `i`, `j`, and `tail` use a fixed amount of space.
3. The string slicing and concatenation operations within the inner loop do not allocate more than `n` characters at a time, which is within the bounds of the original message string.

Since the strings in `ans` can potentially grow to the length of the input message, the space complexity is not constant. However, the space used is at most proportional to the size of the input message, leading to a potential **space complexity** of `O(n)`, assuming that `limit` is not significantly larger than `n`.