2522. Partition String Into Substrings With Values at Most K

Medium Greedy String Dynamic Programming Leetcode Link

Problem Description

You are tasked with partitioning a string `s` which is composed of digits from `1` to `9` into a set of substrings. In the process of partitioning `s`, there are certain constraints that define what constitutes a "good" partition:

- Each digit in `s` can only be a part of one substring, ensuring no overlap.
- Each substring, when converted to an integer, should have a value less than or equal to a given threshold `k`.

The goal of this exercise is to find the minimum number of substrings that can be created while maintaining a "good" partition of `s`. However, if it is not possible to partition `s` according to the criteria, you should return `-1`.

Note:

- The conversion of a substring to its integer value must be taken into account. For instance, the substring "123" is evaluated as the integer 123.
- A substring is a contiguous block of characters found within the larger string `s`.

Intuition

In approaching the problem, the key is to break the string `s` into all possible substrings that comply with the value limit set by `k` and then find the minimal count among those possible partitions. To accomplish this systematically, a recursive approach, also known as Depth-First Search (DFS), can be used:

- Begin at the start of the string and progressively create substrates by adding one digit at a time until the value of the current substring exceeds `k`.
- If the value does not exceed `k`, you recursively call the function for the rest of the string, starting just after the current substring.
- Keep a global record of the least number of substrates needed to partition the string successfully.

Since this approach can result in the same substrings being evaluated multiple times, we use memoization to store the results of subproblems and avoid redundant calculations. This optimization is necessary to ensure that the solution is efficient, especially when dealing with large strings. The pythonic way to achieve memoization is using the `@cache` decorator, which automatically stores the results of the expensive recursive calls.

Lastly, if after applying this method the minimum count remains undefined (`inf`), that means no good partition can be made, hence the function should return `-1`.

Solution Approach

The implementation of the solution is based on a recursive depth-first search (DFS) approach, with memoization to improve efficiency. Here's the breakdown of its algorithm:

1. We define a recursive helper function `dfs`, which receives an index `i`. This function attempts to partition the string starting from `i` and returns the minimum number of substrings needed for a good partition from this index onward.
2. When `dfs` is called with an index `i`, it checks if `i` is equal or greater than the length of `s`. If true, this indicates the end of the string has been reached and no additional substrings are needed, so it returns `0`.
3. The function initializes two variables, `res` and `v`. The former is set to infinity to simulate a maximum possible number, which is used to find the minimum count of substrings; the latter holds the current numerical value of the substring being formed.
4. A loop is used to try forming substrings starting from the index `i`. In each iteration, a digit is added to the current value `v`, which keeps track of the integer value of the current substring.
5. If adding another digit makes `v` greater than `k`, the loop breaks, as the substring is no longer valid.
6. If the current substring's value does not exceed `k`, `dfs` is called recursively with the index just after the current substring to compute the minimum substrings needed for the remainder of `s`.
7. The minimum value between the `res` and the recursive call result is taken to find the optimal number of substrings up to the current digit.
8. After evaluating all possibilities starting from index `i`, the function returns the minimum count found plus `1` to include the current substring in the total.
9. The outer `dfs` function is wrapped with the `@cache` decorator. This decorator ensures that the results of `dfs` calls for each starting index are stored and directly returned upon subsequent calls with the same index.
10. The `dfs` function is initially called with index `0` to start partitioning from the beginning of the string `s`. The result is checked against infinity; if it is less, that means a good partition was found, and the answer is returned. Otherwise, it means no good partition is possible and `-1` is returned.

This solution takes advantage of the fact that Python has built-in support for memoization through the `@cache` decorator as part of the functools library, and it uses recursion to explore all possible partition strategies. This, combined with memoization, allows for a more manageable exploration of the problem space while avoiding redundant calculations.

Example Walkthrough

Let's walk through a simple example to illustrate the solution approach:

Imagine we have the string `s = "1234"` and the threshold `k = 12`. We want to find the minimum number of substrings we can partition `s` into without any of them having an integer value greater than `k`.

Here is how we apply the solution approach:

1. Call the recursive `dfs` function starting at index `0`.
2. The function initializes `res` to infinity and `v` to `0`.
3. It starts trying to form a substring by adding one character at a time, checking whether `v` exceeds `k`:
   - First iteration:
     - The current value `v` is `0`. The digit at `s[0]` is `1`, so `v=1`.
     - Since `v` is less than `k`, recursively call `dfs` with index `1`.
     - The recursive call will try all possible partitions starting from `s[1]`.
   - Second iteration:
     - Now `v=1`, and the next digit is `2`, so `v=12` (concatenating the digits `1` and `2` to form `12`).
     - Since `v` is still less than or equal to `k`, recursively call `dfs` with index `2`.

   The loop will continue to add characters until the value of `v` exceeds `k` or we reach the end of the string.
4. When the recursive calls return, they give us the minimum number of substrings needed from that index onwards.
5. We compare all these returned values to find the minimum number amongst them and add `1` to include the current substring.
6. Once the final recursive call finishes, we check if the result is infinity. If not, we return that result, which will represent the minimum number of substrings.

Following this process for the string `s = "1234"` with `k = 12`:

- `dfs(0)` starts, and attempts to build substrings. It sees that both `dfs(1)` (after using `"1"`) and `dfs(2)` (after using `"12"`) are valid paths to explore. It cannot go further to `dfs(3)` because `123` is greater than `k`.
- `dfs(1)` will further break down the string `"234"`, and once again, it finds that both `dfs(2)` (using `"2"`) and `dfs(3)` (using `"23"`) are valid. `dfs(3)` calls will continue until it reaches the end of the string, incrementing the substring count as needed.
- `dfs(2)` will also try to explore, but as `dfs(3)` has already computed the minimum substrings from that point, it will provide the pre-computed value due to memoization.

In the end, the minimum substrings that `s` can be partitioned into using this approach are two: `"12"` and `"34"`, both of which are less than or equal to `k`.

This example simplifies the process but effectively demonstrates the recursive and memoization techniques used in the solution approach.

Python Solution

```python
from functools import lru_cache
import math

class Solution:
    def minimumPartition(self, s: str, k: int) -> int:
        # Utility function to perform the Depth-First Search (DFS).
        # Caches the results to avoid recalculating for the same inputs.
        @lru_cache(maxsize=None)
        def dfs(index):
            # If the index is beyond the string length, no more partitions are needed.
            if index >= string_length:
                return 0

            # Initialize the result as infinity to find the minimum partitions.
            # 'value' will store the numeric value generated from the string as we traverse.
            result, value = math.inf, 0

            # Iterate through the string characters starting from the current index.
            for j in range(index, string_length):
                # Accumulate the numeric value by considering the current character as part of a number.
                value = value * 10 + int(s[j])

                # If the generated value exceeds 'k', no valid partition can be made, so break out of the loop.
                if value > k:
                    break

                # Recursively find the minimum number of partitions needed for the remaining substring.
                # Add 1 for the current partition and take the minimum with the current result.
                result = min(result, dfs(j + 1))

            # Return the result of 1 (for the current partition) plus the recursively calculated partitions.
            return result + 1

        # Length of the input string.
        string_length = len(s)

        # Start the partitioning process from the first character.
        answer = dfs(0)

        # If the answer is infinity, no partitioning scheme is valid, so return -1.
        # Otherwise, return the calculated minimum number of partitions.
        return answer if answer < math.inf else -1
```

Java Solution

```java
class Solution {
    private Integer[] memo; // memoization table for storing solutions to subproblems
    private int strLength; // length of the input string
    private String str; // the original input string
    private int maxValue; // the maximum allowed value for partitioning the number
    private final int INF = 1 << 30; // representing infinity, used for comparisons

    // Main method to calculate minimum number of splits to partition
    public int minimumPartition(String s, int k) {
        strLength = s.length();
        memo = new Integer[strLength];
        str = s;
        maxValue = k;
        int result = performDFS(0); // starts the depth-first search from the beginning of the string
        return result < INF ? result : -1; // if the result is infinity, return -1, indicating it's not possible
    }

    // Helper method that uses DFS to find the minimum splits recursively
    private int performDFS(int index) {
        // if the current index has reached the end of the string
        if (index >= strLength) {
            return 0;
        }
        // if the current index's result is already computed, return it
        if (memo[index] != null) {
            return memo[index];
        }
        int result = INF;
        long currentValue = 0; // to store the numeric value of the current partition

        // iterate over the string starting from the current index
        for (int j = index; j < strLength; ++j) {
            currentValue = currentValue * 10 + (str.charAt(j) - '0'); // build the next number by appending the current digit

            // if the currentValue exceeds the maxValue allowed for a partition, abort this loop
            if (currentValue > maxValue) {
                break;
            }
            // Recursively perform DFS for the next part and take the minimum result so far
            result = Math.min(result, performDFS(j + 1));
        }
        // Add 1 for the current split and store the result in memoization table before returning
        memo[index] = result + 1;
        return memo[index];
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <cstring>
#include <functional>

class Solution {
public:
    int minimumPartition(std::string s, int k) {
        int n = s.size();
        std::vector<int> dp(n, 0); // Create a vector for dynamic programming storage
        memset(dp.data(), 0, sizeof(dp[0]) * n);
        const int INF = 1 << 30; // Define an 'infinity' value for initialization

        // Define a recursive lambda function for depth-first search (memoization approach)
        std::function<int(int)> dfs = [&](int startIndex) -> int {
            if (startIndex >= n) return 0; // Base case: if we reach the end of string
            if (dp[startIndex] > 0) return dp[startIndex]; // Memoization to save results

            int result = INF;
            long currentValue = 0;

            // Try all possible partitions starting from the current index
            for (int endIndex = startIndex; endIndex < n; ++endIndex) {
                currentValue = currentValue * 10 + (s[endIndex] - '0'); // Calculate number value
                if (currentValue > k) break; // If current number is bigger than k, break

                result = std::min(result, dfs(endIndex + 1)); // Recursively solve the subproblem
            }

            return dp[startIndex] = result + 1; // Store the minimum number of partitions
        };

        // Start the depth-first search from index 0
        int answer = dfs(0);

        // If the answer is not updated, meaning it's not possible, return -1
        return answer < INF ? answer : -1;
    }
};
```

Typescript Solution

```typescript
function minimumPartition(s: string, k: number): number {
    const n: number = s.length;
    const dp: number[] = new Array(n).fill(0); // Dynamic programming storage
    const INF: number = 1 << 30; // Initialize 'infinity' value for impossible cases

    // A recursive function for depth-first search with memoization
    const dfs = (startIndex: number): number => {
        if (startIndex >= n) return 0; // Base case: if we reach the end of the string

        if (dp[startIndex] !== 0) return dp[startIndex]; // Use memoized results to save computation time

        let result = INF;
        let currentValue = 0;

        // Try all possible partitions by considering the current index as the starting point
        for (let endIndex = startIndex; endIndex < n; endIndex++) {
            currentValue = currentValue * 10 + (s.charCodeAt(endIndex) - '0'.charCodeAt(0)); // Calculate the numeric value of the

            if (currentValue > k) break; // Break if the current number exceeds 'k'

            // Recursively solve the subproblem and update the result with the minimum partitions needed
            result = Math.min(result, dfs(endIndex + 1));
        }

        // Store the minimum number of partitions in dp and return
        dp[startIndex] = result + 1;
        return dp[startIndex];
    };

    // Start the DFS from the first index
    const answer = dfs(0);

    // Return the answer if it's valid, otherwise return -1 to indicate impossibility
    return answer < INF ? answer : -1;
}

// The types are specified based on TypeScript best practices
// and function definition requirements. This version does not utilize classes
// but defines methods at the global scope as requested.
```

Time and Space Complexity

Time Complexity

The function uses a depth-first search (DFS) approach with memoization, implemented through the `@cache` decorator. This exploration can visit each index `i` at most once for each unique state, resulting in a finite number of states limited by the length `n` of the string `s`.

When we call `dfs(i)`, in the worst case, it iterates through the rest of the string to calculate the potential partitions that do not exceed the value `k`. Each recursive `dfs(j + 1)` call attempts to partition the string at a different index `j`. Since `v` can grow up to `k`, once `v` exceeds `k`, the loop breaks, constraining the number of iterations.

The time complexity can be seen as `O(n * t)`, where `n` is the length of the string `s` and `t` is the average number of iterations performed in the loop before `v` exceeds `k`. The average number of iterations `t` is difficult to precisely quantify without specific knowledge about `k` and the digit makeup of `s`, but it can be bounded by `n`.

Thus, the worst-case time complexity is `O(n^2)` as a rough estimate, although in practical scenarios it could be significantly less, depending on the nature of input `s` and the value of `k`.

Space Complexity

The space complexity of the code is driven by the recursion depth and the space needed to store the memoization states. The maximum recursion depth can be `n`, and the `@cache` decorator will store a result for each unique call to `dfs(i)`. Since we can have at most `n` unique calls to the `dfs` function, the space complexity is `O(n)` because of memoization.

However, we must also consider the implicit call stack due to recursion, which adds to the space complexity. In the worst case, the call stack's maximum depth would also be `n` since we could be making a recursive call for every character in the string. Thus, the overall space complexity remains `O(n)`.