2767. Partition String Into Minimum Beautiful Substrings

Medium · Hash Table · String · Dynamic Programming · Backtracking · Leetcode Link

Problem Description

The goal is to split a given binary string `s` into the minimum number of contiguous substrings where each substring satisfies two conditions:

1. It does not start with a zero.
2. It represents a number in binary that is a power of 5.

If this partition isn't possible according to the above rules, the output should be -1. This problem is an algorithmic challenge that requires identifying the specific partitions that ensure the minimum count and conform to the rules.

Intuition

To solve this problem, it's crucial to recognize that we're not simply looking for subsequences, but specifically for substrings (which cannot rearrange characters and must remain contiguous). Furthermore, these substrings need to be valid binary representations of powers of 5 without leading zeros.

The concept of dynamic programming may immediately come to mind to handle the optimization aspect of the problem—minimizing the number of substrings. Specifically, we use a depth-first search (DFS) with memoization to ensure efficient computation by avoiding redundant calculations.

Here's how we can approach the solution:

1. Preprocess the binary equivalents of all possible powers of 5 that can be represented within the length of `s` and store them in a hash set `ss`. This preprocessing speeds up the checks needed later by allowing for constant-time lookups to see if a binary substring is a power of 5.
2. Implement the `dfs` function which operates recursively:
   - If at an index `i`, where `s[i]` is '0', we can immediately return infinity (`inf`) because we can't have a leading zero.
   - If we reach the end of the string (`i >= n`), it means we have considered all characters, hence, no further partitions are needed, and we return 0.
   - Otherwise, we iterate from the current index `i` to the end of the string, treating each possible end index `j` as the end of a candidate substring. We calculate the decimal value of the substring as we extend it and check if that value is in `ss`. If it is, we have found a beautiful substring and we add 1 to the result of `dfs(j + 1)`.
3. Optimize by using memoization to cache results of `dfs(i)` for each index `i`. This prevents the algorithm from re-computing the minimum number of substrings for any starting index more than once.
4. Call `dfs(0)` for the start of the string and if the value is infinite, we return -1 since it's impossible to partition the string into beautiful substrings. Otherwise, the value of `dfs(0)` gives us the minimum count of beautiful substrings.

This leads us to a recursive solution with memoization that efficiently computes the minimum required partitions by only considering valid power of 5 numbers and by avoiding redundant checks through memoization.

Solution Approach

The implementation starts with the creation of a set `ss` which contains the binary representation of all the numbers that are powers of 5 up to the maximum length of the binary string `s` provided. This set is pivotal for quickly verifying whether a substring can be considered beautiful.

A recursive function `dfs(i)` is defined that takes an index `i` and returns the minimum number of beautiful substrings starting from this index. The recursion uses two crucial base conditions:

- When the current position `i` is at the end of the string (`i >= n`), which by definition means no further cuts are possible and it returns `0`.
- If the current bit is `0`, indicating a leading zero if it were to be the start of a substring, it returns infinity to signify this cannot form a beautiful substring.

The algorithm then progresses by considering all possible substrings starting from `i` up to the end of the string. For each candidate substring, it performs the following steps:

- It uses bit shifting to calculate the binary to decimal conversion of the substring `s[i...j]` while iterating.
- It checks if the current value, as it accumulates with each bit, exists in the precalculated powers of 5, stored in `ss`. If it does, it means the substring is beautiful.
- It calls `dfs(j + 1)` for the remainder of the string starting from `j + 1`, where `j` is current end of the considered substring. To this `dfs` call, 1 is added representing the current beautiful substring just processed.
- It continues to compute and track the minimum number of beautiful substrings (cuts) as it goes along.

The memoization is achieved using Python's `@cache` decorator on the `dfs` function. This optimization ensures that once a substring starting at index `i` is processed, the result is stored and thus any subsequent calls to `dfs(i)` will simply retrieve the stored result instead of recalculating, reducing the time complexity significantly.

At the end of recursion, if `dfs(0)` returns infinity (`inf`), it means the string `s` cannot be divided into beautiful substrings and thus the function returns `-1`. Otherwise, the minimum number of cuts will be returned.

To avoid repeatedly constructing the set `ss` for powers of 5, it is precomputed once before the recursive calls. This is performed in a loop that starts with `x = 1` and keeps multiplying `x` by `5`, adding each new power of 5 into the `ss`. This loop runs as long as the numbers being added are within the possible range dictated by the length of the string `s`.

Finally, the recursive function is initiated with `dfs(0)` to solve the entire string and the answer is checked against `inf` to return either the minimum cuts or `-1` if the partitioning isn't possible.

Example Walkthrough

Let's consider a small example to illustrate the solution approach step by step. Suppose we have the binary string `s = "101101101"` which we want to partition into beautiful substrings following the given conditions.

1. **Preprocess powers of 5 in binary**: We first create a set `ss` of all binary strings that represent powers of 5 and are less than or equal to the length of `s`. Here is a small set for illustration: `ss = {"1", "101", "10001", ...}`.

2. **Start with the recursive `dfs` function at index 0**: We call `dfs(0)` and begin to explore all substrings starting from index 0.

3. **Exploring substrings**:

   - We consider the first substring which is "1". Since "1" is in the `ss` set, it represents a power of 5. We then call `dfs(1)` (which is the next index) to explore further the rest of the string "01101101".
   - We can't start a substring with '0', so `dfs(1)` immediately moves on to `dfs(2)`.
   - At index 2, we start with substring "1" again which is valid, and thus proceed to `dfs(3)`.
   - And so on, until `dfs(6)` considers the substring "101" which is also a power of 5.
4. **Using Memoization**: Assume `dfs(7)` is called, and the results for this index are calculated and stored. If `dfs(7)` is called again during the recursive calls, we don't recompute it but rather retrieve the stored result.

5. **Recursive and Memoization Results**: Continues until the entire string is processed. During this process:

   - "101101101" at `dfs(0)` would proceed with "1" and call `dfs(1)`.
   - "01101101" at `dfs(1)` skips the zero and then proceeds with "1" and call `dfs(2)`.
   - "1101101" at `dfs(2)` proceeds with "1" and call `dfs(3)`.
   - "101101" at `dfs(3)` then finds "101", calls `dfs(6)`.
   - "101" at `dfs(6)` is a power of 5 itself, so it calls `dfs(9)`, now at the base case because the end of the string is reached and it returns 0.
   - The result of `dfs(9)` is added to those of `dfs(6)`, `dfs(3)`, `dfs(2)`, and `dfs(0)` to find the minimum number of beautiful substrings. For this example, `dfs(9)` returns `0`, `dfs(6)` returns `1`, `dfs(3)` returns `2`, `dfs(2)` returns `3`, and `dfs(0)` finally returns `4`.
6. **End of Recursion**: After applying the memoization and the entire recursion, if the returned value is infinity (`inf`), our function outputs `-1`, indicating that it's not possible to partition `s` according to the rules. However, for `s = "101101101"`, the minimum number of beautiful substrings returned is 4 which are "1", "1", "1", and "101".

Therefore, for this example, our algorithm would successfully partition the string into the substrings {"1", "1", "1", "101"}, each of which is a binary representation of a power of 5, and since we've used 4 substrings, the output would be 4. If a partition was not found, our algorithm would return -1.

Python Solution

```python
from functools import lru_cache  # Import the lru_cache decorator for memoization

class Solution:
    def minimumBeautifulSubstrings(self, binary_string: str) -> int:
        # A decorator that caches the return values of the function it decorates
        @lru_cache(maxsize=None)
        def min_substrings_from_index(index: int) -> int:
            # If we have reached the end of the binary_string, no more substrings needed
            if index >= length:
                return 0
            # We cannot start with a '0' for a beautiful binary_string
            if binary_string[index] == "0":
                return float('inf')  # Represents an impossible scenario

            current_value = 0
            best_result = float('inf')
            # Iterate over the binary_string starting from current index
            for j in range(index, length):
                # Shift current_value by one bit and add the new bit
                current_value = (current_value << 1) | int(binary_string[j])
                # Check if current_value is a power of 5
                if current_value in powers_of_five:
                    # Calculate the minimum substrings if we take current substring
                    # And then add 1 for the current one
                    best_result = min(best_result, 1 + min_substrings_from_index(j + 1))
            return best_result

        length = len(binary_string)  # Total length of the binary_string
        power_of_five = 1
        # A set to store the powers of 5 values
        powers_of_five = {power_of_five}
        # Generate powers of 5 up to the length of the binary_string
        for i in range(length):
            power_of_five *= 5
            powers_of_five.add(power_of_five)

        # Start the search from index 0
        result = min_substrings_from_index(0)
        # Return -1 if there's no valid partition, otherwise the minimum substrings
        return -1 if result == float('inf') else result

```

Java Solution

```java
import java.util.HashSet;
import java.util.Set;

public class Solution {
    private Integer[] memoization; // memoization array to store results of subproblems
    private String inputString; // the input string
    private Set<Long> powersOfFive; // set containing powers of five
    private int inputLength; // the length of the input string

    // Method to calculate the minimum number of beautiful substrings
    public int minimumBeautifulSubstrings(String s) {
        inputLength = s.length();
        this.inputString = s;
        memoization = new Integer[inputLength];
        powersOfFive = new HashSet<>();

        // Precompute powers of 5 and add to the set
        long power = 1;
        for (int i = 0; i <= inputLength; ++i) {
            powersOfFive.add(power);
            power *= 5;
        }

        // Attempt to find the minimum beautiful substrings starting from the first character
        int result = findMinimum(0);

        // If the result exceeds the length of the input string, return -1, indicating no such decomposition
        return result > inputLength ? -1 : result;
    }

    // Helper method to recursively find the minimum number of beautiful substrings starting at index i
    private int findMinimum(int i) {
        if (i >= inputLength) { // base case: if we've reached the end of the string
            return 0;
        }
        if (inputString.charAt(i) == '0') { // no substring starting with a '0' can be beautiful
            return inputLength + 1;
        }
        if (memoization[i] != null) { // return the precomputed value if available
            return memoization[i];
        }

        long binaryValue = 0; // to store the numerical value of the substring in binary
        int ans = inputLength + 1; // initialize the minimum with an upper bound

        // Loop to consider all substrings starting at 'i'
        for (int j = i; j < inputLength; ++j) {
            binaryValue = (binaryValue << 1) | (inputString.charAt(j) - '0'); // accumulate the binary value
            if (powersOfFive.contains(binaryValue)) { // if the binary value is a power of five
                // Attempt to find the minimum starting at the next character and add 1 for the current substring
                ans = Math.min(ans, 1 + findMinimum(j + 1));
            }
        }

        // Store the result in the memoization array before returning
        return memoization[i] = ans;
    }
}

```

C++ Solution

```cpp
#include <unordered_set>
#include <string>
#include <cstring>
#include <functional>
using namespace std;

class Solution {
public:
    // Function to calculate the minimum number of beautiful substrings.
    int minimumBeautifulSubstrings(string s) {
        unordered_set<long long> beautifulNumbers;
        int n = s.size();
        long long powerOfFive = 1;
        // Populate a set with powers of 5. These represent the "beautiful numbers" in binary.
        for (int i = 0; i <= n; ++i) {
            beautifulNumbers.insert(powerOfFive);
            powerOfFive *= 5;
        }

        // Array to store minimum beautiful substrings starting at each index
        int minSubstrings[n];
        memset(minSubstrings, -1, sizeof(minSubstrings));

        // Lambda function to calculate minimum beautiful substrings using DFS
        function<int(int)> dfs = [&](int idx) {
            if (idx >= n) { // If the entire string has been processed
                return 0; // Base case: no more substrings, so return 0
            }
            if (s[idx] == '0') { // Beautiful substrings cannot start with '0'
                return n + 1; // Return a big number which will not be minimum
            }
            if (minSubstrings[idx] != -1) { // Check if already computed
                return minSubstrings[idx];
            }
            long long num = 0;
            int ans = n + 1; // Initialize the answer with a large number
            for (int j = idx; j < n; ++j) {
                num = (num << 1) | (s[j] - '0'); // Convert binary to decimal
                if (beautifulNumbers.count(num)) { // If it's a beautiful number
                    // Take minimum of current answer and 1 plus the answer from next index
                    ans = min(ans, 1 + dfs(j + 1));
                }
            }
            return minSubstrings[idx] = ans; // Memoize and return the answer
        };

        int ans = dfs(0); // Start DFS from index 0
        return ans > n ? -1 : ans; // If answer is greater than n, no beautiful substring is found, return -1
    }
};

```

Typescript Solution

```typescript
function minimumBeautifulSubstrings(s: string): number {
    // Set to hold powers of 5
    const powersOfFive: Set<number> = new Set();
    const lengthOfS = s.length;

    // Array to hold the minimum beautiful substrings from each index
    const minSubstrFromIndex: number[] = new Array(lengthOfS).fill(-1);

    // Pre-calculate powers of 5 and store in the set
    for (let i = 0, power = 1; i <= lengthOfS; ++i) {
        powersOfFive.add(power);
        power *= 5;
    }

    // Helper function to perform a depth-first search
    const depthFirstSearch = (startIndex: number): number => {
        // Base case: If we've reached the end of the string, return 0
        if (startIndex === lengthOfS) {
            return 0;
        }
        // If the current character is a '0', it cannot be beautiful, return large number
        if (s[startIndex] === '0') {
            return lengthOfS + 1;
        }
        // Use memoization to avoid recalculating
        if (minSubstrFromIndex[startIndex] !== -1) {
            return minSubstrFromIndex[startIndex];
        }
        // Initialize a large number for comparison
        minSubstrFromIndex[startIndex] = lengthOfS + 1;

        // Look ahead in the string to find valid beautiful substrings
        for (let endIndex = startIndex, binaryValue = 0; endIndex < lengthOfS; ++endIndex) {
            // Incrementally construct the binary value represented by the substring
            binaryValue = (binaryValue << 1) | (s[endIndex] === '1' ? 1 : 0);
            // Check if the current binary value is a power of 5
            if (powersOfFive.has(binaryValue)) {
                // If it is, update the minimum count and recurse
                minSubstrFromIndex[startIndex] = Math.min(minSubstrFromIndex[startIndex], 1 + depthFirstSearch(endIndex + 1));
            }
        }

        // Return the minimum count of beautiful substrings starting from the current index
        return minSubstrFromIndex[startIndex];
    };

    // Start the depth-first search from the first character
    const answer = depthFirstSearch(0);

    // If the answer is larger than the length of the string, no valid solution exists, return -1
    return answer > lengthOfS ? -1 : answer;
}

```

Time and Space Complexity

The given Python code defines a recursive function `dfs` to compute the minimum number of beautiful substrings. The time and space complexity analysis are as follows:

Time Complexity

The time complexity of this code is $O(n^2)$. Here's the detailed reasoning:

- The function `dfs` is called recursively and it iterates over the entire length of the string `s` for each starting index `i`. In the worst-case scenario, the inner loop could run through the remaining part of the string, which gives us `n` iterations when starting at the first character, `n-1` on the second, down to `1` iteration at the last character. Summing these up gives us the arithmetic series $n + (n-1) + ... + 1$, which has a sum of $(n(n+1))/2$, resulting in $O(n^2)$ time complexity.
- Each recursive call involves a constant time check and a loop which shifts and adds digits to `x`, but these operations are $O(1)$ for each individual call.
- The calls to `min` and to check `if x in ss` are also $O(1)$ since checking membership in a set is constant time on average.

Space Complexity

The space complexity of this code is $O(n)$ for the following reasons:

- The recursive function `dfs` could at most be called `n` times consecutively before reaching the base case (a call stack of depth `n`), thus the space complexity due to recursion is $O(n)$.
- The hash set `ss` which contains at most `n` elements also requires $O(n)$ space.
- There's a cache used to store the results of subproblems in `dfs` due to the `@cache` decorator. The number of distinct subproblems is proportional to the length of the string `s`, also leading to a space complexity of $O(n)$.

Therefore, the overall space complexity is dominated by these factors, resulting in $O(n)$.