761. Special Binary String

Hard Recursion String Leetcode Link

Problem Description

Special binary strings are binary strings that strictly adhere to two rules. The first rule is that they must have an equal number of `0's` and `1's`. The second rule is that every prefix (a substring starting from the first character) must not have more `0's` than `1's`. In other words, as we sequentially read the string from left to right, at no point should the number of `0's` surpass the number of `1's`.

The task is to manipulate a given special binary string `s` through a series of permitted moves to create the lexicographically largest (i.e., the "biggest" or the "last" in dictionary order) string possible. Each move involves selecting two non-empty special substrings that are adjacent to each other and swapping them.

To put this into perspective, consider a simplified version of alphabet ordering where we only have `0` and `1`. Here, `1` is considered greater than `0`, so for example, the string `1100` is lexicographically larger than `1001`.

Intuition

To solve this problem, we should realize that since the manipulated string must remain special, we are somewhat limited in how we can rearrange it. The key insight here is to think of the special binary string as a balanced sequence of parentheses, where `1` maps to an open parenthesis `(`, and `0` maps to a close parenthesis `)`. It's easier to recognize valid substrings if we utilize this analogy, because balanced parentheses are a common and well-understood problem.

Recall that in a correctly balanced string of parentheses, the smallest unit is `()`. Similarly, in our binary string the smallest special substring is `10`. Building on this, we can deduce that any special binary string can be recursively divided into smaller special binary substrings. This is because a balanced parenthesis string is effectively a sequence of smaller balanced parenthesis strings. In terms of binary strings; a special binary string can be seen as a concatenation of `1`, some special string `S`, followed by `0` – `1S0`.

Our approach, therefore, involves recursively dividing the special binary string into these smaller substrings, solving for each smaller problem, and combining the results. After breaking the string down to manageable chunks, we sort them in reverse to ensure that the lexicographically largest subsequence comes first.

The provided solution implements this concept by counting the number of `1's` and `0's`. Each time the counts are equal, we've identified a valid special substring. It's sliced, solved recursively, and wrapped with `1` at the beginning and `0` at the end to maintain its special string status. These substrings are then collected and sorted reversely before being joined to form the largest possible string.

By ensuring a largest-to-smallest order in the final combination, we guarantee that the lexicographically largest string is formed. This forms the basis of the solution's logic and its implementation in code.

Solution Approach

The Python solution's approach uses a simple but clever recursive strategy with sorting to generate the lexicographically largest special binary string. Here's a breakdown of the algorithm and key parts of the implementation:

1. **Base Case**: If the input string `s` is empty, we return an empty string immediately, as there are no moves to make.
2. **Variables Initialization**:
   - `ans`: A list that will hold all valid special binary substrings we find.
   - `cnt`: A counter that helps us track the balance of `1's` and `0's`.
   - `i` and `j`: Two pointers that mark the current position in the string and the beginning of a new special substring, respectively.
3. **Finding Special Binary Substrings**:
   - As we iterate through the string `s`, we increase `cnt` when we find a `1` and decrease it when we find a `0`.
   - Whenever `cnt` becomes `0`, we've identified a complete and valid special binary substring, ranging from indices `j + 1` to `i`.
4. **Recursion**:
   - Once a valid special binary substring is found, we recursively call `makeLargestSpecial` on the inner substring (excluding the outer `1` and `0`), to sort that substring's smaller units and make them as large as possible lexicographically.
   - After the recursion, we wrap this sorted substring with `1` at the start and `0` at the end to maintain its "special" property.
   - This special binary substring is then added to the `ans` list.
5. **Sorting and Rebuilding**:
   - After separating and individually treating all the valid special binary substrings, we sort the `ans` list in reverse to ensure the substrings are arranged lexicographically largest to smallest.
   - The sorted list is then joined into a single string, which is the lexicographically largest special binary string that can be formed.
6. **Data Structures and Patterns**:
   - **Recursion**: Essential for breaking down the problem into smaller subproblems, in the manner of the "Divide and Conquer" paradigm.
   - **List**: Used to store substrings before sorting and joining.
   - **Sorting**: Critical to rearrange the substrings in reverse order for lexicographical precedence.

Through this method, we take advantage of the fact that the lexicographically largest combination of substring sequences comes from sorting them in descending order. The recursion ensures that this sorting logic is applied at every level of the special binary string's structure, thus ensuring the largest possible configuration.

Example Walkthrough

Let's illustrate the solution approach with a small example. Suppose we have the special binary string `s = "11011000"`.

1. **Base Case Check**: The string `s` is not empty, so we proceed.
2. **Variables Initialization**:
   - `ans`: Initialize as an empty list to hold all valid special binary substrings found.
   - `cnt`: Initialize as `0`. It is used for tracking the balance of `1's` and `0's`.
   - `i` and `j`: Initialize both as `0`. These pointers will track the current position in the string and the start of a new special substring.
3. **Finding Special Binary Substrings**: We iterate through `s`:
   - At `i = 0`, `s[i] = 1`, so `cnt` becomes `1`.
   - At `i = 1`, `s[i] = 1`, so `cnt` becomes `2`.
   - At `i = 2`, `s[i] = 0`, so `cnt` becomes `1`.
   - At `i = 3`, `s[i] = 1`, so `cnt` becomes `2`.
   - At `i = 4`, `s[i] = 1`, so `cnt` becomes `3`.
   - At `i = 5`, `s[i] = 0`, so `cnt` becomes `2`.
   - At `i = 6`, `s[i] = 0`, so `cnt` becomes `1`.
   - At `i = 7`, `s[i] = 0`, so `cnt` becomes `0`, indicating we've found a complete special binary substring `11011000` (from `j = 0` to `i = 7`).
4. **Recursion**:
   - With the found substring `"11011000"`, we ignore the first `1` and the last `0` (substring `"101100"`). Recursively call `makeLargestSpecial` on this inner substring to sort smaller units.
   - Recursively, we would find two smaller substrings within "`101100`": `"1010"` and `"100"`. We apply the same process to them, recursively sorting inner substrings to get `"1100"` for the first and just `"100"` for the second (since it cannot be broken down further). Wrap them with `1` and `0` to maintain "special" properties, becoming `"1100"` and `"100"` respectively.
   - Add these substrings into `ans`: `ans = ["1100", "100"]`.
5. **Sorting and Rebuilding**:
   - We sort `ans` in reverse to obtain `ans = ["1100", "100"]`.
   - Join the sorted substrings in `ans` to form the final string: `"11001000"`.
6. **Data Structures and Patterns**:
   - Recursion breaks the problem down and applies logic at each level.
   - A list, `ans`, holds and sorts substrings.
   - Sorting is used to ensure lexicographical order is largest to smallest.

Through this process, we've transformed the initial string `"11011000"` to its lexicographically largest form `"11001000"`, by applying the solution's recursive strategy and sorting mechanism.

Python Solution

```python
class Solution:
    def makeLargestSpecial(self, s: str) -> str:
        # Base case: if the input string is empty, return an empty string.
        if s == '':
            return ''

        # Initialize a list to store special binary strings.
        special_strings = []

        # Initialize counters for the current balance of 1's and 0's and the start index of a substring.
        balance = 0
        start_index = 0

        # Iterate through the characters of the input string.
        for i in range(len(s)):
            # Increase balance for '1' and decrease for '0'.
            balance += 1 if s[i] == '1' else -1

            # When the balance is zero, a special string is found.
            if balance == 0:
                # Recursively process the inner substring (excluding the first and last characters),
                # and wrap it with '1' and '0' to make it a special string.
                inner_special = '1' + self.makeLargestSpecial(s[start_index + 1 : i]) + '0'

                # Add the current special string to the list.
                special_strings.append(inner_special)

                # Update the start index to the next character after the current special string.
                start_index = i + 1

        # Sort the list of special strings in descending order to create the largest number.
        special_strings.sort(reverse=True)

        # Join the sorted special strings and return the result.
        return ''.join(special_strings)

```

Java Solution

```java
class Solution {
    // Recursive method that rearranges a special binary string to create the largest possible string.
    public String makeLargestSpecial(String s) {
        // Base case: If the string is empty, return an empty string.
        if (s.isEmpty()) {
            return "";
        }

        // A list to hold special strings during the processing.
        List<String> specialStrings = new ArrayList<>();
        int count = 0; // Counter to check the number of '1's and '0's.

        // Loop through the string to identify special substrings.
        for (int start = 0, end = 0; end < s.length(); ++end) {
            // Increment counter for '1', decrement for '0'.
            count += s.charAt(end) == '1' ? 1 : -1;

            // When the count is zero, we found a balanced part of the special string.
            if (count == 0) {
                // Recursively process the inside of this special substring
                // and add "1" at the beginning and "0" at the end.
                String inner = makeLargestSpecial(s.substring(start + 1, end));
                specialStrings.add("1" + inner + "0");

                // Set the start to the next character after the current special substring.
                start = end + 1;
            }
        }

        // Sort the processed special strings in reverse order to make the largest string.
        specialStrings.sort(Comparator.reverseOrder());

        // Join all sorted special substrings into one string and return it.
        return String.join("", specialStrings);
    }
}

```

C++ Solution

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

class Solution {
public:
    string makeLargestSpecial(string s) {
        // Base case: If the string is empty, return as is.
        if (s.empty()) return s;

        // Define a vector to hold special substrings.
        vector<string> specials;

        int counter = 0; // Initialized to count the balance of 1s and 0s.
        int startIdx = 0; // Index to keep track of the start of a special substring.

        // Iterate over the string to find and process special substrings.
        for (int currentIdx = 0; currentIdx < s.size(); ++currentIdx) {
            // Increment counter if '1', decrement if '0'.
            counter += s[currentIdx] == '1' ? 1 : -1;

            // When counter is 0, we found a special string.
            if (counter == 0) {
                // Make a special string by recursively calling makeLargestSpecial on the inner part.
                // We concatenate '1' at the beginning and '0' at the end.
                specials.push_back("1" + makeLargestSpecial(s.substr(startIdx + 1, currentIdx - startIdx - 1)) + "0");

                // Set the start index for the next special substring.
                startIdx = currentIdx + 1;
            }
        }

        // Sort the special substrings in descending order to make the string largest.
        sort(specials.begin(), specials.end(), greater<string>());

        // Concatenate all special strings together using accumulate.
        return accumulate(specials.begin(), specials.end(), string{});
    }
};

```

Typescript Solution

```typescript
function makeLargestSpecial(s: string): string {
    // Base case: If the string is empty, return it as is.
    if (s.length === 0) {
        return s;
    }

    // Define an array to hold special substrings.
    const specials: string[] = [];

    let counter = 0; // Initialize to count the balance of '1's and '0's.
    let startIdx = 0; // Index to keep track of the start of a special substring.

    // Iterate over the string to find and process special substrings.
    for (let currentIdx = 0; currentIdx < s.length; currentIdx++) {
        // Increment counter if '1', decrement if '0'.
        counter += (s[currentIdx] === '1') ? 1 : -1;

        // When counter is 0, we have found a special string.
        if (counter === 0) {
            // Construct a special string by recursively calling makeLargestSpecial on the inner part,
            // then concatenate '1' at the beginning and '0' at the end.
            specials.push('1' + makeLargestSpecial(s.substring(startIdx + 1, currentIdx)) + '0');

            // Set the start index for the next potential special substring.
            startIdx = currentIdx + 1;
        }
    }

    // Sort the special substrings in descending order to make the string largest.
    specials.sort((a, b) => b.localeCompare(a));

    // Concatenate all special strings together by joining the array elements.
    return specials.join('');
}

// The method can be exported if needed, to be used in other modules.
export { makeLargestSpecial };

```

Time and Space Complexity

Time Complexity

The time complexity of the function mainly stems from the recursive calls and the sorting operation. Let's break it down:

1. **Recursion**: The function `makeLargestSpecial` is called recursively every time a special binary string is detected (when `cnt` returns to 0). The maximum depth of recursion is bounded by `n` (the length of the string), since in the worst case scenario, the whole string is a special binary string, which needs to be decomposed entirely.

2. **Sorting**: After each inner special string is made largest through the recursive call, the resulting substrings are sorted in reverse order. This sorting step takes `O(k log k)` where `k` is the number of special substrings at the same level of recursion. Since all special substrings from the entire input string are eventually sorted at the top level, in the worst-case scenario, this sorting can take up to `O(n log n)` time where `n` is the length of the input string.

Considering both recursion and sorting, and the fact that sorting can happen at each level of recursion, the overall worst-case time complexity is `O(n^2 log n)` because the sorting time (`O(n log n)`) could potentially be performed `n` times (at each character position in the string in the worst case).

Space Complexity

The space complexity is primarily due to the recursive call stack and the space needed for storing intermediate substrings in the `ans` array:

1. **Call Stack**: Since the recursion can go as deep as the length of the string in the worst-case scenario, we could potentially have a call stack of depth `n`. This contributes `O(n)` in space complexity.

2. **Intermediate Strings**: The `ans` list stores intermediate substrings, which, together, will not exceed the length of the input string. Thus, this contributes `O(n)` in space complexity. Additionally, substring slicing creates new strings, which may also contribute to the space complexity.

In summary, considering the call stack and the storage for intermediate strings, the space complexity of the algorithm is `O(n)`. Note that the sorting operation is in-place and doesn't significantly add to space complexity.