2904. Shortest and Lexicographically Smallest Beautiful String

Medium String Sliding Window Leetcode Link

Problem Description

You're tasked to find the lexicographically smallest "beautiful" substring within a binary string `s` which contains exactly `k` occurrences of the character '1'. A "beautiful" substring is defined as having exactly `k` number of '1's within it. If no substring matches this criterion, you need to return an empty string.

The lexicographical order here refers to the natural dictionary order where a string is compared character by character from the left and as soon as a difference is found, the comparison is decided by the difference in those characters like following "abcd" and "abcc", 'd' is greater than 'c'.

For example, consider the binary string `s = "001101"` and `k = 2`. The "beautiful" substrings that contain exactly two '1's are "0110", "1101", and "101". The shortest of these is "101", which is also the lexicographically smallest one. Therefore, "101" is the answer.

Intuition

To solve the problem, we have to efficiently find the shortest "beautiful" substrings and among them determine the one that is smallest lexicographically. The brute-force or enumeration method would be to check all possible substrings which would be very slow for longer strings because we'd need to compare a vast number of substrings. To optimize, we can use a sliding-window approach, accomplished using two pointers, to identify "beautiful" substrings.

Here's how the two-pointer approach helps in finding the solution:

- Use the `i` pointer to mark the beginning of your current window (initially at the start of the string) and `j` to mark the end (also initially at the start).
- The `cnt` variable tracks the count of '1's in the current window. The window extends (by increasing `j`) until the `cnt` of '1's is equal to `k`.
- During this extension, if `cnt` exceeds `k`, or if the current window starts with '0', we move `i` ahead to try and find a smaller window with `k` number of '1's.
- Once we find a window with exactly `k` '1's, we compare it to the current answer. For this, we have three conditions:
  - If there's no answer yet, the current window is our new answer.
  - If the length of the current window is less than the length of the current answer, then the current window becomes our new answer (since shorter is better).
  - If the current window is the same length as the answer but lexicographically smaller, then the current window becomes the new answer.

By the end, our sliding window will have given us the lexicographically smallest "beautiful" substring that is of the shortest length.

Solution Approach

The provided solution uses a two-pointer approach, which is a pattern commonly used to efficiently process subarrays or substrings of a given array or string. Here's a detailed step-by-step explanation of the algorithm based on the reference solution approach:

1. Initialize two pointers, `i` and `j`, which represent the start and end of a sliding window, to `0`. Also, initialize a counter `cnt` to `0` to count the number of '1's within the window, and an empty string `ans` to keep track of the current answer.
2. Slide the right boundary of the window (represented by `j`) to the right by incrementing `j` in a loop until the end of the string. Update `cnt` by checking if the current character `s[j]` is '1'.
3. If `cnt` exceeds `k`, or if the current window starts with '0' (not a valid start for the beautiful substring), slide the left boundary of the window (represented by `i`) to the right until the window is beautiful again. This step ensures we always have a valid beautiful substring within the window or an empty window ready to grow.
4. Whenever `cnt` equals `k`, check if we found a smaller or lexicographically smaller beautiful substring than the current answer. If so, update the `ans`.
   - `not ans`: No answer has been found yet, so any found substring becomes the new answer.
   - `j - i < len(ans)`: Found a shorter beautiful substring.
   - `(j - i == len(ans) and s[i:j] < ans)`: Found an equally short but lexicographically smaller beautiful substring.
5. Continue the above process until `j` reaches the end of the string. At this point, the smallest lexicographically beautiful substring is stored in `ans`.

The main data structure used here is a string to keep the current answer for comparison. No extra space is required apart from the input, making the space complexity `O(1)`, not considering the input and output strings.

The time complexity is `O(n)`, where `n` is the length of the string `s`. This is because each character in `s` is visited at most twice: once when expanding the right boundary (`j`), and once when contracting the left boundary (`i`).

Example Walkthrough

Let's walk through a small example to illustrate the solution approach described.

Suppose we are given the binary string `s = "0101101"` and `k = 3`. We need to find the smallest lexicographically "beautiful" substring that contains exactly three '1's.

1. Initialize `i` and `j` to `0`, `cnt` to `0` (count of '1's in the current window), and `ans` as an empty string.
2. We advance `j` from `0` to `6` (end of the string) in a loop:
   - At `j=0`: `s[j]='0'`, so `cnt` remains `0`.
   - At `j=1`: `s[j]='1'`, so increment `cnt` to `1`.
   - At `j=2`: `s[j]='0'`, so `cnt` remains `1`.
   - At `j=3`: `s[j]='1'`, increment `cnt` to `2`.
   - At `j=4`: `s[j]='1'`, increment `cnt` to `3`. Now we have a "beautiful" substring "0101" from index `i=0` to `j=4`.
3. We update `ans` to "0101" because the `ans` is currently empty.
4. We continue to increment `j`:
   - At `j=5`: `s[j]='0'`, so `cnt` remains `3`. The substring "01010" is longer than "0101", we don't update `ans`.
   - At `j=6`: `s[j]='1'` and now `cnt` exceeds `k` (it's `4` now). We need to adjust the window.
5. To adjust the window, we slide `i` to the right to decrease `cnt` back to `k`:
   - Increment `i` to `1`: `s[i-1]='0'`, so `cnt` remains `4`.
   - Increment `i` to `2`: `s[i-1]='1'`, decrement `cnt` to `3`. Now, the substring is "10101" from index `i=2` to `j=6`.
6. Check if the new window is smaller or lexicographically smaller than `ans` but since it's not, we do not update `ans`.
7. As `j` has reached the end of `s`, the loop ends, and the smallest lexicographically "beautiful" substring found is "0101" which is stored in `ans`.

Therefore, for the string `s = "0101101"` with `k = 3`, the output will be "0101". This approach ensures we scan the string only once, and efficiently find the substring we're looking for.

Python Solution

```python
class Solution:
    def shortestBeautifulSubstring(self, s: str, k: int) -> str:
        # Initialize pointers and counter
        left_pointer = right_pointer = count_ones = 0
        length_of_s = len(s)
        shortest_substring = ""

        # Iterate over the string while maintaing a sliding window
        while right_pointer < length_of_s:
            # Count occurrences of '1'
            count_ones += s[right_pointer] == "1"

            # If we have more than k '1's or the current character is '0',
            # shrink the window from the left
            while count_ones > k or (left_pointer < right_pointer and s[left_pointer] == "0"):
                count_ones -= s[left_pointer] == "1"
                left_pointer += 1

            # Move the right end of the window forward
            right_pointer += 1

            # Check if we have found a valid beautiful substring
            # with exactly k '1's and update answer if it's the shortest seen so far,
            # or if it's lexicographically smaller than the current best with equal length.
            if count_ones == k and (
                not shortest_substring or
                right_pointer - left_pointer < len(shortest_substring) or
                (right_pointer - left_pointer == len(shortest_substring) and s[left_pointer:right_pointer] < shortest_substring)
            ):
                shortest_substring = s[left_pointer:right_pointer]

        # Return the shortest beautiful substring found
        return shortest_substring

```

Java Solution

```java
class Solution {
    public String shortestBeautifulSubstring(String s, int k) {
        int start = 0; // 'start' is the beginning index of the current substring
        int end = 0; // 'end' is the ending index of the current substring (exclusive)
        int count = 0; // Count of current number of '1's
        int n = s.length(); // Length of the string 's'
        String answer = ""; // Initialize the answer as an empty string

        // Iterate through the string 's' with 'end' as the right boundary
        while (end < n) {
            // Increase count if the current character is '1'
            count += s.charAt(end) - '0';

            // Shrink the window from the left if the count is greater than 'k'
            // or if the leading character is '0' and the window size is greater than 1.
            while (count > k || (start < end && s.charAt(start) == '0')) {
                count -= s.charAt(start) - '0'; // Decrease the count while moving 'start' to the right
                ++start; // Move the start index to the right
            }

            ++end; // Move the end index to the right

            // Get the current substring from start to end
            String currentSubstring = s.substring(start, end);

            // Check if the current substring is beautiful,
            // and if it's the shortest one seen so far or lexicographically smaller.
            if (count == k && (answer.isEmpty() || end - start < answer.length() ||
                    (end - start == answer.length() && currentSubstring.compareTo(answer) < 0))) {
                answer = currentSubstring; // Update the answer with the current substring
            }
        }
        // Return the shortest lexicographically smallest beautiful substring found
        return answer;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function that returns the shortest substring which contains 'k' number of '1's
    string shortestBeautifulSubstring(string str, int k) {
        int startIndex = 0, endIndex = 0, oneCount = 0;
        int strSize = str.size();
        string shortestSubstring = "";

        // Iterate over the string to find the valid substrings
        while (endIndex < strSize) {
            // Increment oneCount if current character is '1'
            oneCount += (str[endIndex] == '1');

            // Shrink the window from the left if
            // we have more than k '1's or the current
            // window starts with a '0'
            while (oneCount > k || (startIndex < endIndex && str[startIndex] == '0')) {
                oneCount -= (str[startIndex++] == '1');
            }

            // Move to the next character in the string
            ++endIndex;

            // Extract the current valid substring
            string currentSubstring = str.substr(startIndex, endIndex - startIndex);

            // Check if the current substring satisfies the conditions:
            // 1) It contains exactly k '1's
            // 2) It is shorter than the previously recorded shortest substring
            // 3) Or, it is the same length as the previously recorded shortest
            //    substring but lexicographically smaller
            if (oneCount == k &&
                (shortestSubstring.empty() || endIndex - startIndex < shortestSubstring.size() ||
                 (endIndex - startIndex == shortestSubstring.size() && currentSubstring < shortestSubstring))) {
                shortestSubstring = currentSubstring;
            }
        }

        // Return the shortest substring that meets the conditions
        return shortestSubstring;
    }
};

```

Typescript Solution

```typescript
function shortestBeautifulSubstring(word: string, threshold: number): string {
    let startIndex = 0;  // Start index of the current substring
    let endIndex = 0;    // End index of the current substring
    let oneCount = 0;    // Count of '1's in the current substring
    const length = word.length;  // Length of the input string
    let answer: string = '';  // The shortest beautiful substring found

    // Traverse the string
    while (endIndex < length) {
        // Increase count if '1' is found
        oneCount += word[endIndex] === '1' ? 1 : 0;

        // Shrink the window from the left if the count of '1's is more than the threshold
        // or if the current character is '0' and the window size is more than 1
        while (oneCount > threshold || (startIndex < endIndex && word[startIndex] === '0')) {
            oneCount -= word[startIndex] === '1' ? 1 : 0;
            startIndex++;
        }

        // Move to next character in the string
        endIndex++;

        // Get the current substring from startIndex to endIndex (non-inclusive)
        const currentSubstring = word.slice(startIndex, endIndex);

        // If count of '1's equals threshold, and there is no answer yet or current substring is
        // shorter or lexicographically smaller than the previous answer, update the answer
        if (oneCount === threshold && (answer === '' || endIndex - startIndex < answer.length || (endIndex - startIndex === answer.le
            answer = currentSubstring;
        }
    }

    // Return the shortest beautiful substring
    return answer;
}

```

Time and Space Complexity

The time complexity of the given code is `O(n)` where `n` is the length of the string `s`. This is because there are two pointers `i` and `j`, and each pointer only moves from the beginning to the end of the string in a linear fashion. There are no nested loops, and each character of the string is processed at most twice (once when `j` increments and potentially once when `i` increments).

The space complexity of the code is `O(1)` if we only take into account the space used for variables and pointers, which is constant and does not depend on the input size. If we consider the space required for the output string `ans`, in the worst case it could be as large as the input string, leading to a space complexity of `O(n)` where `n` is the length of the string `s`.