2663. Lexicographically Smallest Beautiful String

Hard Greedy String Leetcode Link

Problem Description

In this problem, we are dealing with the concept of **beautiful strings**. A string is considered beautiful if two conditions are met:

1. It is composed only of the first k letters of the English lowercase alphabet.
2. It does not contain any palindromic substring of length 2 or more.

Given a beautiful string s with a length of n and a positive integer k, the task is to find the next lexicographically smallest beautiful string that is larger than s. The lexicographic order is like dictionary order - for example, b is larger than a, ab is larger than aa, and so on. If the string contains a letter at some position where it differs from another string and this letter is bigger, the entire string is considered larger in lexicographic order.

To solve the problem, we want to:

- Preserve the "beautiful" characteristic of the string.
- Increment the string in the minimal way to find the next lexicographically larger string.

Intuition

The intuitive approach to this problem is to use a greedy strategy. The goal is to find the smallest lexicographical increment possible that results in a string fulfilling the beauty criteria.

Take the following steps:

1. Start from the end of the given string s and move backwards to find the first place where we can increment a character without breaking the beauty rules.
2. Once such a place is found, replace the character at that position with the next possible character that does not create a palindrome with the two previous characters.
3. After replacing a character, it is necessary to build the rest of the string from that point onward with the smallest possible characters from the set of allowed first k characters also while avoiding the creation of palindromic substrings.
4. If no such position exists, then it is not possible to create a lexicographically larger beautiful string, and an empty string should be returned as per the problem statement.

This greedy approach ensures that the resulting string is strictly the next lexicographically larger string since only the necessary changes are made starting from the back, and each replacement ensures the smallest viable increment.

Solution Approach

The provided reference solution uses a backwards greedy algorithm to build the lexicographically smallest beautiful string that comes after the given string s. Here's the step-by-step approach based on the code:

1. We initialize a variable n with the length of the given string s and convert the string to a list of characters, cs.

2. The algorithm then iterates over cs from the last character to the first. For each character, it tries to find the next character in the alphabet that can be placed at that position to make the string lexicographically larger while maintaining the non-palindrome property.

   To do this, it converts the character to its numerical equivalent (`p = ord(cs[i]) - ord('a') + 1`) and iterates from p to k-1 to find a suitable replacement.

3. A replacement is deemed suitable if it isn't the same as the character immediately before it (`cs[i - 1]`) or the one two positions before (`cs[i - 2]`). This ensures that substrings of length 2 or 3 are not palindromic.

4. Once a suitable character is found for position i, the code updates `cs[i]` and moves on to filling the rest of the string from `i + 1` to `n - 1`. For this remaining part, the smallest possible character that does not form a palindrome with the previous two characters is selected. Again, this is checked by comparing against the characters at positions `l - 1` and `l - 2`.

5. When the loop successfully updates the character array cs, the newly formed string is returned by joining the list of characters.

6. If the loop terminates without finding a suitable replacement for any of the positions, it means no lexicographically larger beautiful string can be formed, and therefore an empty string `''` is returned.

The above algorithm effectively uses a linear scan of the input string in the worst-case scenario, making the time complexity linear in relation to the length of s. The use of a fixed-size ASCII offset (`ord('a')`) ensures a constant time operation for character to integer conversion, providing an efficient means to compare and assign characters.

In summary, the approach relies on a greedy algorithm that backs up from the end of the string to find the smallest necessary increments while maintaining the string's beauty, taking advantage of character-integer conversions and simple comparisons to enforce the beauty constraints.

Example Walkthrough

Let's take a small example to illustrate the solution approach. Imagine we have a beautiful string `s = "abac"` and `k = 3` (meaning we can only use the first 3 letters of the English alphabet: a, b, c).

Following the solution approach, we perform the following steps:

1. Start from the end of s, which is the character c. We need to find a place where we can increment a character without creating palindrome substrings.

2. The last character c is already the maximum for `k = 3`, so we move backward. Before c, there is an a, but increasing a to b would form bb, which is a palindrome. Therefore, we continue moving backward.

3. Before a, we have b. We can't increment b to c because that will create the palindrome cac.

4. We reach the first character a. We can increment a to b resulting in the string bbac. However, this forms a palindrome bb. Hence, we have to increment b to c, ensuring no direct repetition with its neighboring character.

5. Now we have cbac, but we are not allowed to have direct repetitions or create palindromic substrings such as cac. Since c is already the maximum allowed character for our k, we must convert the subsequential characters to the smallest possible characters (a), ensuring that we do not form a palindrome. However, placing a after c would give us caa, which is a palindrome; thus we must place a b instead.

6. Following this pattern, the next characters will be the lowest possible values ensuring no palindromic substrings are created. After b, the smallest allowed character which doesn't create a palindrome is a.

7. The resulting string is cbab which is the smallest lexicographic increment from our original string abac that keeps the string beautiful.

So, the solution algorithm would output cbab as the next lexicographically smallest beautiful string after abac when using the first 3 lowercase letters of the alphabet.

Python Solution

```python
class Solution:
    def smallestBeautifulString(self, s: str, k: int) -> str:
        # Get the length of the string
        string_length = len(s)

        # Convert the string to a list for easier manipulation
        char_list = list(s)

        # Iterate backwards through the character list
        for i in range(string_length - 1, -1, -1):
            # Find the position of the current character in the alphabet (1-based)
            position = ord(char_list[i]) - ord('a') + 1

            # Try replacing current character with the next possible beautiful character
            for replacement_pos in range(position, k):
                # Calculate the replacement character based on its position
                replacement_char = chr(ord('a') + replacement_pos)

                # Check if the previous two characters are the same as the potential replacement character
                # If so, skip this replacement
                if (i > 0 and char_list[i - 1] == replacement_char) or (i > 1 and char_list[i - 2] == replacement_char):
                    continue

                # Assign the replacement character to the current position
                char_list[i] = replacement_char

                # For remainder of the string, place lexicographically smallest possible character
                for l in range(i + 1, string_length):
                    # Iterate over possible characters
                    for m in range(k):
                        next_char = chr(ord('a') + m)
                        # Skip if the character matches any of the previous two characters
                        if (l > 0 and char_list[l - 1] == next_char) or (l > 1 and char_list[l - 2] == next_char):
                            continue
                        # Assign and break since the smallest possible character has been found
                        char_list[l] = next_char
                        break

                # Return the modified string after the first replacement
                return ''.join(char_list)

        # Return an empty string if no possibility to make the string beautiful
        return ''
```

Java Solution

```java
class Solution {
    public String smallestBeautifulString(String s, int k) {
        // Length of the input string
        int n = s.length();
        // Convert the input string to a character array for manipulation
        char[] chars = s.toCharArray();

        // Iterating from the end of the character array
        for (int i = n - 1; i >= 0; --i) {
            // Calculate the numeric position of the current character
            int position = chars[i] - 'a' + 1;
            // Iterate through the alphabet starting from the character after the current one
            for (int j = position; j < k; ++j) {
                // Convert the numeric position into a character
                char nextChar = (char) ('a' + j);
                // Check if replacing the current character with the next one obeys the rules
                // (next character should not be the same as the previous two characters)
                if ((i > 0 && chars[i - 1] == nextChar) || (i > 1 && chars[i - 2] == nextChar)) {
                    continue;
                }

                // Assign the new character to the current position
                chars[i] = nextChar;
                // Iterate over the remaining characters in the array starting from i+1
                for (int l = i + 1; l < n; ++l) {
                    // Iterate over all possible characters within the given limit 'k'
                    for (int m = 0; m < k; ++m) {
                        nextChar = (char) ('a' + m);
                        // Apply the same rule for the next characters
                        if ((l > 0 && chars[l - 1] == nextChar) || (l > 1 && chars[l - 2] == nextChar)) {
                            continue;
                        }
                        // Assign the chosen character and break to the next position
                        chars[l] = nextChar;
                        break;
                    }
                }
                // Return the new string after constructing it from the character array
                return String.valueOf(chars);
            }
        }
        // If no valid string can be formed, return an empty string
        return "";
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    // Function to create the smallest lexicographically string that follows specific rules
    string smallestBeautifulString(string s, int k) {
        int n = s.size(); // Get the length of the input string

        // Iterate backward through the string
        for (int i = n - 1; i >= 0; --i) {
            int position = s[i] - 'a' + 1; // Convert character to zero-based position

            // Try each possible character starting from the current position to k - 1
            for (int j = position; j < k; ++j) {
                char c = 'a' + j; // Convert integer position j to its corresponding character

                // Skip characters that are illegal due to the rules (same as previous one or two)
                if ((i > 0 && s[i - 1] == c) || (i > 1 && s[i - 2] == c)) {
                    continue;
                }

                s[i] = c; // Set the current character of the string s

                // Update the rest of the string after the current position
                for (int l = i + 1; l < n; ++l) {
                    // Try each character from 'a' to 'a' + (k - 1)
                    for (int m = 0; m < k; ++m) {
                        c = 'a' + m;

                        // Apply the same rules, skip if the character is illegal
                        if ((l > 0 && s[l - 1] == c) || (l > 1 && s[l - 2] == c)) {
                            continue;
                        }

                        // Set the next character in the string s
                        s[l] = c;
                        break; // Break to fill the next position in the string
                    }
                }

                return s; // Return the updated string once it is built
            }
        }

        return ""; // Return an empty string if no valid string can be formed
    }
};
```

Typescript Solution

```typescript
function smallestBeautifulString(inputString: string, alphabetSize: number): string {
    // Split the input string into an array of characters
    const characters: string[] = inputString.split('');
    // Get the length of the character array
    const length = characters.length;

    // Loop backwards through the characters
    for (let index = length - 1; index >= 0; --index) {
        // Calculate the alphabet position of the current character and increment it by 1 (to find next character)
        const currentCharPos = characters[index].charCodeAt(0) - 'a'.charCodeAt(0) + 1;

        // Loop over the possible characters (starting from currentCharPos to alphabetSize)
        for (let charCode = currentCharPos; charCode < alphabetSize; ++charCode) {
            // Get the character representation of charCode
            let nextChar = String.fromCharCode(charCode + 'a'.charCodeAt(0));

            // Check the adjacent characters to avoid consecutive duplicates
            if ((index > 0 && characters[index - 1] === nextChar) ||
                (index > 1 && characters[index - 2] === nextChar)) {
                continue;
            }

            // Update the current character in the array
            characters[index] = nextChar;

            // Fill the rest of the string with non-repeating characters
            for (let nextIndex = index + 1; nextIndex < length; ++nextIndex) {
                // Try characters from 'a' to the kth letter in the alphabet
                for (let nextCharCode = 0; nextCharCode < alphabetSize; ++nextCharCode) {
                    nextChar = String.fromCharCode(nextCharCode + 'a'.charCodeAt(0));

                    // Check the adjacent characters to avoid consecutive duplicates
                    if ((nextIndex > 0 && characters[nextIndex - 1] === nextChar) ||
                        (nextIndex > 1 && characters[nextIndex - 2] === nextChar)) {
                        continue;
                    }

                    // Update the character at nextIndex with the non-repeating character
                    characters[nextIndex] = nextChar;
                    break;
                }
            }

            // Convert the array back to a string and return the resultant string
            return characters.join('');
        }
    }

    // If no beautiful string could be formed, return an empty string
    return '';
}
```

Time and Space Complexity

Time Complexity

The time complexity of the code is actually not `O(n)`. In the worst case, both of the nested loops could iterate up to k times for each position. The outer loop runs from `n-1` down to `0`, contributing an `O(n)` factor. Inside this loop, we have the second loop iterating through the possible characters less than k, giving a factor of `O(k)`. Similarly, the innermost loop, which resets each character after the current position, iterates up to k times for each remaining position, leading to a factor of `O(k * (n - i))`. Therefore, the worst-case time complexity is closer to `O(n * k^2)`.

Space Complexity

The space complexity is indeed `O(1)`, assuming that the space needed for the input string s is not counted as additional space since we're transforming it in-place into a list cs. The only additional space used is for a few variables to store indices and characters, which does not scale with the input size, hence constant space complexity.