2182. Construct String With Repeat Limit

Medium | Greedy | String | Counting | Heap (Priority Queue) | Leetcode Link

Problem Description

You are provided with a string `s` and an integer `repeatLimit`. The task is to create a new string, called `repeatLimitedString`, using the characters from `s`. The catch is that no character can repeat more than `repeatLimit` times consecutively. It is not necessary to use all characters from `s`. The goal is to build the lexicographically largest `repeatLimitedString` possible.

A string is considered lexicographically larger if at the position where two strings start to differ, the string with the character higher in the alphabet comes first. If they are identical up to the length of the shorter one, then the longer string is considered larger.

Intuition

To form the lexicographically largest string, we should always prefer to use the largest character available (closest to 'z'). We start from the end of the alphabet and move towards the start ('z' to 'a'), using up to `repeatLimit` occurrences of the current character before we must use a different character to avoid breaking the repeat rule.

The intuition behind the solution is as follows:

- Count the occurrences of each character in the input string `s`. This is done to quickly find out how many times we can use each character without counting repeatedly.
- Start constructing the `repeatLimitedString` by iterating from the highest character ('z') down to the lowest ('a'). Add up to `repeatLimit` instances of the current character to the result.
- If there are still occurrences of the current character left (more than `repeatLimit`), add a character next in line (the highest character remaining that is not the current one). This step ensures that we do not break the repeat limit for the current character.
- Repeat the process of adding the current character (up to the `repeatLimit`) and then the next in line (just once) until we run out of characters or we can no longer add the current character without breaking the repeat limit.
- If we reach a point where no other characters are available to be inserted between repeats of the current character, we terminate the process as we can't add more of the current character than allowed.
- By following this process and prioritizing the largest character possible at each step, we ensure that the result is the lexicographically largest possible string under the given constraints.

Solution Approach

The solution approach involves a few key steps and makes use of a frequency counter and a greedy strategy to construct the required `repeatLimitedString`.

Here's a step-by-step walkthrough of the implementation:

1. **Counting Frequencies**: We use an array `cnt` of size 26 (since there are 26 letters in the English alphabet) to count the occurrences of each character in string `s`. The ASCII value is used to map characters 'a' to 'z' to indices 0 to 25.

2. **Iterating in Reverse**: We loop through the `cnt` array in reverse order starting from the index corresponding to 'z' (25) down to 'a' (0). This allows us to consider characters in decreasing lexicographic order.

3. **Adding Characters to the Result**: In each iteration, we add up to `repeatLimit` instances of the character corresponding to the current index to the answer list `ans`, decrementing the count in `cnt`. The character to be added is obtained by computing `chr(ord('a') + i)`, where `i` is the current index.

4. **Maintaining the Repeat Limit**: If the count of the current character is not exhausted after adding `repeatLimit` instances, we need to add a different character to ensure no character is repeated more than `repeatLimit` times in a row.

5. **Finding the Next Character**: We use another pointer `j` and decrement it to find the next available character with a nonzero count. We then add one instance of this character to `ans`, ensuring the repeat limit condition is satisfied.

6. **Stopping Conditions**: There are two stopping conditions in the while loop. The loop ends when the count of the current character reaches 0 (`cnt[i] == 0`), or when there is no other character left to insert (`j < 0`).

By combining the usage of the frequency counter with a greedy approach, we ensure that we always construct the string with the highest lexicographical order possible under the given constraints of the `repeatLimit`.

Here's a portion of the code that visualizes this process:

```python
for i in range(25, -1, -1):
    j = i - 1
    while True:
        for _ in range(min(repeatLimit, cnt[i])):
            cnt[i] -= 1
            ans.append(chr(ord('a') + i))
        if cnt[i] == 0:
            break
        while j >= 0 and cnt[j] == 0:
            j -= 1
        if j < 0:
            break
        cnt[j] -= 1
        ans.append(chr(ord('a') + j))
return ''.join(ans)
```

In the above code, `ans` is a list that eventually contains the characters of the `repeatLimitedString` in the desired order. The `join` method is used to combine these characters into a string before returning it as the final answer.

Example Walkthrough

Let's consider a small example to illustrate the solution approach. Suppose we have the string `s = "aabab"` and a `repeatLimit` of 2. We want to construct the lexicographically largest `repeatLimitedString` by using the characters from `s` according to the given constraints.

1. **Counting Frequencies**:
   - We count each character's frequency and store it in an array `cnt`. Here's what the `cnt` array will look like for each alphabet character:
     - 'a': 3
     - 'b': 2
   - The other characters are not present in `s` and would have a count of 0.
2. **Iterating in Reverse**:
   - We start from 'z' and go down to 'a', so the first character we consider from `s` that has a non-zero frequency is 'b'.
3. **Adding Characters to the Result**:
   - We add 'b' to our result. Since `repeatLimit` is 2 and we have 2 'b's, we add both of them: `repeatLimitedString = "bb"`.
   - We decrement the count of 'b' in `cnt` to 0.
4. **Maintaining the Repeat Limit**:
   - Now that we've used all 'b's, we move to the next available character with the highest lexicographical value, which is 'a'.
   - We can add up to 2 'a's (due to `repeatLimit`) to `repeatLimitedString` which now becomes: `repeatLimitedString = "bbaa"`.
5. **Finding the Next Character**:
   - We have one 'a' remaining, but we cannot add it immediately because that would violate the `repeatLimit`.
   - We search for the next available character, but as we've used up all 'b's, and no characters are left, we're only left with 'a'.
6. **Stopping Conditions**:
   - We've reached the stopping condition because we have no characters left that we can insert to avoid having more than 2 consecutive 'a's.
   - Thus, we cannot add the last 'a'.

The lexicographically largest `repeatLimitedString` we can create with the given `s` and `repeatLimit` is `bbaa`.

By following the described solution approach, we constructed the lexicographically largest string without breaking the rules of the given constraints.

Python Solution

```python
class Solution:
    def repeatLimitedString(self, s: str, repeat_limit: int) -> str:
        # Initialize a list to keep track of the count of each character
        char_count = [0] * 26

        # Count the occurrences of each character in the input string s
        for c in s:
            char_count[ord(c) - ord('a')] += 1

        # Initialize a list to build the answer string
        answer = []

        # Iterate over characters from 'z' to 'a'
        for i in range(25, -1, -1):
            # Find the next character to place if we hit the repeat limit
            next_char_index = i - 1

            # Keep placing characters until we run out
            while True:
                # Place the current character (up to the repeat limit) in the answer
                for _ in range(min(repeat_limit, char_count[i])):
                    char_count[i] -= 1
                    answer.append(chr(ord('a') + i))

                # If we have placed all instances of the current character, move to the next
                if char_count[i] == 0:
                    break

                # Find the next highest character which we haven't exhausted
                while next_char_index >= 0 and char_count[next_char_index] == 0:
                    next_char_index -= 1

                # If there are no more characters to use, we are done
                if next_char_index < 0:
                    break

                # Place the next character
                char_count[next_char_index] -= 1
                answer.append(chr(ord('a') + next_char_index))

        # Return the constructed string
        return ''.join(answer)

```

Java Solution

```java
class Solution {

    public String repeatLimitedString(String s, int repeatLimit) {
        // Array to hold the count of each letter in the string
        int[] letterCount = new int[26];

        // Counting the occurrences of each character in the string
        for (char ch : s.toCharArray()) {
            letterCount[ch - 'a']++;
        }

        // StringBuilder to build the result string
        StringBuilder result = new StringBuilder();

        // Iterate from the letter 'z' to 'a'
        for (int i = 25; i >= 0; --i) {
            // Pointer to check for the next available smaller character
            int nextCharIndex = i - 1;

            while (true) {
                // Append the current character up to repeatLimit times
                for (int k = Math.min(repeatLimit, letterCount[i]); k > 0; --k) {
                    letterCount[i]--; // Decrement the count of the current character
                    result.append((char) ('a' + i)); // Add the current character to the result
                }

                // If all occurrences of the current character have been used up, break out of the loop
                if (letterCount[i] == 0) {
                    break;
                }

                // Find the next available character which has at least one occurrence left
                while (nextCharIndex >= 0 && letterCount[nextCharIndex] == 0) {
                    --nextCharIndex;
                }

                // If there are no more characters left to use, break out of the loop
                if (nextCharIndex < 0) {
                    break;
                }

                // Append the next available character
                letterCount[nextCharIndex]--;
                result.append((char) ('a' + nextCharIndex));
            }
        }

        // Return the result as a string
        return result.toString();
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    string repeatLimitedString(string s, int repeatLimit) {
        // Count occurrences of each character in the input string
        vector<int> charCounts(26, 0);
        for (char ch : s) {
            charCounts[ch - 'a']++;
        }

        // Initialize the answer string
        string result;

        // Iterate over the characters starting from 'z' to 'a'
        for (int i = 25; i >= 0; --i) {
            int nextCharIndex = i - 1;

            // Continue to construct the string until all chars are used
            while (true) {
                // Add the current character up to repeatLimit times
                int repeatCount = min(charCounts[i], repeatLimit);
                for (int k = repeatCount; k > 0; --k) {
                    charCounts[i]--;
                    result.push_back('a' + i);
                }

                // Break the loop if this character is fully used
                if (charCounts[i] == 0) break;

                // Find the next available character with remaining count
                while (nextCharIndex >= 0 && charCounts[nextCharIndex] == 0) {
                    --nextCharIndex;
                }

                // Break if there are no more characters to use
                if (nextCharIndex < 0) break;

                // Add the next available character to the result string
                charCounts[nextCharIndex]--;
                result.push_back('a' + nextCharIndex);
            }
        }

        // Return the constructed string
        return result;
    }
};

```

Typescript Solution

```typescript
function repeatLimitedString(s: string, repeatLimit: number): string {
    // Count occurrences of each character in the input string
    const charCounts: number[] = new Array(26).fill(0);
    for (const ch of s) {
        charCounts[ch.charCodeAt(0) - 'a'.charCodeAt(0)]++;
    }

    // Initialize the answer string
    let result: string = '';

    // Iterate over the characters starting from 'z' to 'a'
    for (let i = 25; i >= 0; --i) {
        let nextCharIndex = i - 1;

        // Continue to construct the string until all chars are used
        while (true) {
            // Add the current character up to repeatLimit times
            const repeatCount = Math.min(charCounts[i], repeatLimit);
            for (let k = repeatCount; k > 0; --k) {
                charCounts[i]--;
                result += String.fromCharCode('a'.charCodeAt(0) + i);
            }

            // Break the loop if this character is fully used
            if (charCounts[i] === 0) break;

            // Find the next available character with remaining count
            while (nextCharIndex >= 0 && charCounts[nextCharIndex] === 0) {
                --nextCharIndex;
            }

            // Break if there are no more characters to use
            if (nextCharIndex < 0) break;

            // Add the next available character to the result string
            charCounts[nextCharIndex]--;
            result += String.fromCharCode('a'.charCodeAt(0) + nextCharIndex);
        }
    }

    // Return the constructed string
    return result;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the code can be broken down into the following major parts:

1. **Character Counting:** The first loop counts the frequency of each character in the string `s`. This loop runs exactly `len(s)` times. Therefore, this part has a time complexity of `O(n)` where `n` is the length of the string `s`.

2. **Main Loop for Building the Answer:** The outer loop runs 26 times (once for each lowercase English letter). The inner loop could, in the worst case, run roughly `n / repeatLimit` times if all characters are the same and need to be interspersed with another character after hitting the `repeatLimit`. However, since we only insert a single character in between repetitions and then continue with the same character until the `repeatLimit` is hit again, the actual number of visits to each character count (`cnt[i]`) will be proportional to its frequency. This means that the operation count is effectively linear with respect to the string length `n`. Therefore, we can consider this part also to have a time complexity of `O(n)`.

3. **Looking for the Next Character:** Within the inner `while` loop, there's another loop that searches for the next available character (`j`) if the repeat limit is reached. In the worst-case scenario, this could traverse all characters smaller than `i`, adding an extra factor of at most 26 (constant time) per repeat limit hit. Thus, this does not affect the overall linear complexity relative to `n`.

Hence the total time complexity is `O(n)` since the character search is bounded by a constant and doesn't depend on `n`.

Space Complexity

The space complexity is determined by the additional space used by the algorithm which is not part of the input or output:

1. **Character Count Array:** The `cnt` array has a space complexity of `O(1)` since it is always a fixed size of 26 regardless of the input size.

2. **Output List:** The `ans` list ultimately stores each character from the input string in some order, and hence it will have a space complexity of `O(n)`.

Thus, the total space complexity of the algorithm is `O(n)` due to the output list size being proportional to the input size.