1419. Minimum Number of Frogs Croaking

Medium String Counting Leetcode Link

Problem Description

Your task is to find the minimum number of different frogs that could have produced a given string `croakOfFrogs`. Each frog croaks in a specific sequence 'c', 'r', 'o', 'a', 'k'. It's possible for multiple frogs to croak at the same time, which results in an interleaving of these sequences. The goal is to find out how many distinct frogs are needed to generate the string as it's given, with the stipulation that a valid "croak" sound from a frog is represented by the characters `"croak"` appearing sequentially. If the string does not represent a combination of valid "croak" sounds, the function should return `-1`.

Intuition

To solve this problem, we need to track the progress of each croak as it's being formed.

Firstly, since each croak is composed of 5 characters, if the total length of `croakOfFrogs` isn't a multiple of 5, it's impossible to form croaks with equal numbers of 'c's, 'r's, 'o's, 'a's and 'k's and we can immediately return `-1`.

Next, we create a counter array `cnt` that keeps track of how many frogs are currently at each stage of croaking. The index in the array corresponds to the characters in the order 'c', 'r', 'o', 'a', 'k', initialized as [0, 0, 0, 0, 0].

Now, we iterate through the characters in `croakOfFrogs` and update the `cnt` array for the stages:

1. Every time we encounter a 'c', we start a new croak, hence we increment the count for 'c'.
2. For the other characters ('r', 'o', 'a', 'k'), we check if there's a preceding character sound that's in progress (the previous character count should be greater than 0). If not, the string is invalid and we return `-1`.
3. If a preceding character is found, we decrement the counter for that character (since the frog has moved to the next stage of croaking) and increment the counter for the current character.

To find the minimum number of frogs needed, we keep a counter `x` which increments every time we start a "croak" with 'c' and decrement when a croak is completed with 'k'.

The answer will be the maximum value of `x` at any point, which represents the peak number of concurrent croaking frogs required to generate the given string.

Finally, we check if `x` is 0 after the iteration, which would mean all croaks are completed. If any croaks are left unfinished (x is not 0), we return `-1` as the string wouldn't be valid in that case. Otherwise, we return the maximum number recorded in `x` as the answer.

The key insight here is ensuring that the sequence of 'croak' is valid and tracking the peak concurrency, which equates to the minimum number of different frogs required.

Solution Approach

The implementation follows a straightforward approach using a list to keep track of the stages of croaking for multiple frogs simultaneously.

1. **Initialization**: A dictionary `idx` is created to map each character 'c', 'r', 'o', 'a', 'k' to an index `0` through `4`. This maps each character to its respective position in the croaking sequence. Also, an array `cnt` of 5 zeroes is initialized to count the number of frogs at each stage of croaking.
2. **Check Length**: The first if-statement checks if the length of `croakOfFrogs` is a multiple of 5 (since each complete croak sequence has 5 letters). If not, the function returns `-1` immediately because it's impossible to form valid croak sequences otherwise.
3. **Iteration**: The algorithm iterates over each character in `croakOfFrogs`. Using `map(idx.get, croakOfFrogs)`, each character is replaced by its corresponding index. This allows the function to work with indices instead of characters, facilitating easier increment/decrement operations.
4. **Counting Progress**: For each character index `i`:
   - If we encounter a 'c' (`i == 0`), it represents the start of a new croak, so the count of 'c' (`cnt[0]`) is incremented, and the variable `x` is also incremented. The variable `ans` keeps track of the maximum value that `x` takes, representing the peak concurrency of croaking frogs.
   - For indices `1` to `4` (`r`, `o`, `a`, `k`), the algorithm checks if there's a preceding character count that's not zero (which would indicate that a frog is in the previous stage of croaking). If such a count does not exist (`cnt[i - 1] == 0`), it means the sequence is incomplete or out of order, so `-1` is returned.
   - If a valid sequence is maintained, the counter for the preceding stage is decremented (`cnt[i - 1] -= 1`), indicating that one frog has moved on to the current stage.
   - When a 'k' is encountered (`i == 4`), it signals the end of a croak, so `x` is decremented.
5. **Final Check**: After the loop, if `x` is not `0`, it means there are incomplete croak sequences, and thus the input string is invalid. The function returns `-1` in this case. If `x` is `0`, the function returns `ans`, which contains the maximum number of frogs that were croaking at the same time at any point during the input string, which is the answer to the problem.

The core algorithm hinges on the idea of maintaining the sequential integrity of the "croak" sounds and tracking the maximum number of overlapping croaks. This is done by incrementing and decrementing counters in a list based on the sequential sound stages of the frogs.

This approach effectively uses constant space (since the size of `cnt` is always five) and linear time complexity relative to the length of the input string.

Example Walkthrough

Let's illustrate the solution approach using a small example.

Consider the input string `croakOfFrogs` as `"croakcroa"`.

1. **Initialization**: We start by mapping each character ('c', 'r', 'o', 'a', 'k') to its index (0 - 4) and initialize the `cnt` counter array as [0, 0, 0, 0, 0].
2. **Check Length**: The length of `"croakcroa"` is 9, which is not a multiple of 5. Therefore, by our algorithm, this string cannot be made up of a valid sequence of "croak" sounds. As per the approach, we should return `-1` immediately.

However, for illustrative purposes, let's pretend that the string was `"croakcroak"` to show how the process would continue with a valid string length.

3. **Iteration**: Now we map the characters to their respective indices as we iterate through `"croakcroak"`, giving us the indices [0, 1, 2, 3, 4, 0, 1, 2, 3, 4].
4. **Counting Progress**: We process the indexed characters in order:
   - For the first 'c' (`i == 0`), we increment `cnt[0]` to 1 and also increment `x` to 1. The `ans` is now 1.
   - Next, we read 'r' (`i == 1`), we see that `cnt[0]` is 1 (since there is an ongoing 'c' croak), so we decrement `cnt[0]` to 0 and increment `cnt[1]` to 1. No change in `x` or `ans`.
   - This process continues for 'o', 'a', and 'k', with the following states: `cnt` becomes [0, 0, 0, 0, 0], and `x` comes back to 0 each time after reading 'k'.
   - Upon the next 'c', we again increment `cnt[0]` to 1 and `x` to 1, pushing `ans` to 2 because we have two concurrent croaks.
   - As we continue, `cnt` eventually returns to [0, 0, 0, 0, 0] and `x` to 0, after finishing the second "croak".
5. **Final Check**: We finished processing the string with `x` being 0, confirming all "croak" sounds are complete. `ans` is 2, meaning at one point during the process, there were two frogs croaking at the same time. This is the minimum number of frogs required to produce the sequence `"croakcroak"`.

This walkthrough simplifies the process of the solution approach, emphasizing the sequence tracking and concurrency accounting that is at the heart of the algorithm.

Python Solution

```python
class Solution:
    def minNumberOfFrogs(self, croak_of_frogs: str) -> int:
        # If string length is not a multiple of 5, it can't be a combination of 'croak'
        if len(croak_of_frogs) % 5 != 0:
            return -1

        # Create an index map for the characters in the word 'croak'
        char_to_index = {char: index for index, char in enumerate('croak')}
        # Initialized list to keep count of the characters processed
        char_count = [0] * 5
        # Initialize variables for max frogs needed at one time and current counting
        max_frogs = 0
        current_frogs = 0

        # Iterate through the characters in the input string
        for char in map(char_to_index.get, croak_of_frogs):
            # Increment the count for this character in the sequence
            char_count[char] += 1

            # If the character is 'c', we might need another frog
            if char == 0:
                current_frogs += 1
                # Update max frogs if we have more concurrent frogs than before
                max_frogs = max(max_frogs, current_frogs)
            else:
                # If the prior character in the sequence hasn't occurred an equal or greater number of times,
                # the sequence is broken and we return -1
                if char_count[char - 1] == 0:
                    return -1
                # Decrement the count of the previous character to match a frog's croak sequence
                char_count[char - 1] -= 1

                # If the character is 'k', it completes the croak sound for a frog
                if char == 4:
                    # A frog has completed croaking, so we decrease the count of current frogs
                    current_frogs -= 1

        # If there are still frogs croaking by the end, return -1, else return the max frogs needed
        return -1 if current_frogs else max_frogs

```

Java Solution

```java
class Solution {

    public int minNumberOfFrogs(String croakOfFrogs) {
        int length = croakOfFrogs.length();
        // If the length of the string is not a multiple of 5, it's not possible to form a 'croak'.
        if (length % 5 != 0) {
            return -1;
        }

        int[] charToIndex = new int[26]; // This maps characters to their respective indices in the sequence 'croak'.
        String sequence = "croak";
        for (int i = 0; i < 5; ++i) {
            charToIndex[sequence.charAt(i) - 'a'] = i;
        }

        int[] counts = new int[5]; // Counts for each character in the sequence.
        int maxFrogs = 0, currentFrogs = 0; // maxFrogs is the maximum number of frogs croaking at the same time.

        for (int i = 0; i < length; ++i) {
            int currentIndex = charToIndex[croakOfFrogs.charAt(i) - 'a']; // Map the char to its index in 'croak'.
            ++counts[currentIndex];

            if (currentIndex == 0) {
                // If the character is 'c', one more frog starts croaking.
                maxFrogs = Math.max(maxFrogs, ++currentFrogs);
            } else {
                // If counts of the previous character in sequence are less than 0, it is an invalid sequence.
                if (--counts[currentIndex - 1] < 0) {
                    return -1;
                }
                if (currentIndex == 4) {
                    // The character is 'k', so a frog has finished croaking.
                    --currentFrogs;
                }
            }
        }
        // If there are still frogs croaking (currentFrogs > 0), the sequence is invalid.
        return currentFrogs > 0 ? -1 : maxFrogs;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int minNumberOfFrogs(string croakOfFrogs) {
        // The total length of the input string must be divisible by 5 since each "croak" counts for 5 characters.
        int length = croakOfFrogs.size();
        if (length % 5 != 0) {
            return -1; // Early return if an invalid size is checked to ensure each frog completes its croak.
        }

        // 'indices' array maps each character 'c', 'r', 'o', 'a', 'k' to their respective indices 0 to 4.
        int indices[26] = {};
        string sequence = "croak";
        for (int i = 0; i < 5; ++i) {
            indices[sequence[i] - 'a'] = i;
        }

        // 'counts' array tracks the number of ongoing croaks at each stage of "c", "r", "o", "a", "k".
        int counts[5] = {};
        int maxFrogs = 0; // Store the maximum number of frogs croaking at the same time.
        int currentCroaks = 0; // Current number of ongoing croaks.

        // Iterate through the input string.
        for (char& c : croakOfFrogs) {
            int idx = indices[c - 'a']; // Get the index for the current character.
            ++counts[idx];

            if (idx == 0) {
                // If the character is 'c', it could be the start of a new croak sound.
                maxFrogs = max(maxFrogs, ++currentCroaks);
            } else {
                // If it's not 'c', we must have heard the previous character to continue a valid croak.
                if (--counts[idx - 1] < 0) {
                    return -1; // Invalid sequence, as 'c' did not come before 'r', 'o', 'a', or 'k'.
                }
                if (idx == 4) {
                    // If the character is 'k', it denotes the end of a croak. Hence, decrease ongoing croaks.
                    --currentCroaks;
                }
            }
        }

        // If 'currentCroaks' is greater than 0, it means some croaks didn't finish with 'k'.
        // If all croaks are completed, the maximum number of concurrent croaks is our answer.
        return currentCroaks > 0 ? -1 : maxFrogs;
    }
};

```

Typescript Solution

```typescript
// TypeScript function that calculates the minimum number of frogs
// needed to compose a given sequence of "croak" sounds.
// If the sequence is invalid, it returns -1.
function minNumberOfFrogs(croakOfFrogs: string): number {
    const sequenceLength = croakOfFrogs.length;

    // If the length of the sequence is not a multiple of 5, it cannot be a valid sequence of "croak"
    if (sequenceLength % 5 !== 0) {
        return -1;
    }

    // Helper function to get the index of a character in the word "croak"
    const getCroakIndex = (character: string): number => 'croak'.indexOf(character);

    // Array to count the number of times each character of "croak" has been encountered
    const characterCount: number[] = [0, 0, 0, 0, 0];

    // Variable representing the maximum number of frogs needed at any point in the sequence
    let maxFrogs = 0;

    // Variable representing the current number of active frogs during the processing of the sequence
    let activeFrogs = 0;

    for (const character of croakOfFrogs) {
        const index = getCroakIndex(character);
        characterCount[index]++;

        // If the character is 'c', we might need a new frog or use one that just finished croaking
        if (index === 0) {
            maxFrogs = Math.max(maxFrogs, ++activeFrogs);
        } else {
            // Before a character can be used, the previous character in "croak" must have been encountered
            if (--characterCount[index - 1] < 0) {
                return -1;
            }
            // If the character is 'k', a frog has finished croaking
            if (index === 4) {
                --activeFrogs;
            }
        }
    }
    // If there are any active frogs remaining, it means the sequence did not end with complete "croak" sounds
    return activeFrogs > 0 ? -1 : maxFrogs;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the function is `O(N)`, where `N` is the length of the input string `croakOfFrogs`. This is because the function consists of a single loop that iterates through each character of the input string exactly once. The operations within the loop are constant time lookups in the `idx` dictionary and simple arithmetic operations, which do not depend on the size of the input string.

Space Complexity

The space complexity of the function is `O(1)`. The additional space required by the algorithm includes a fixed-size `cnt` array of 5 elements to keep track of the 'croak' sequence counts, the constant size `idx` dictionary with a mapping from characters to indices, and a few integer variables (`ans`, `x`). Since these do not grow with the size of the input, the space required remains constant, irrespective of the input size.