2024. Maximize the Confusion of an Exam

Medium | String | Binary Search | Prefix Sum | Sliding Window — Leetcode Link

Problem Description

The problem presents a scenario where a teacher is attempting to create a true/false test that is deliberately designed to be confusing for the students. The confusion is maximized by having the longest possible sequence of consecutive questions with the same answer, either true ('T') or false ('F').

The `answerKey` string is given, representing the correct answers for each question, where each character is either 'T' for true or 'F' for false. Additionally, you are given an integer `k` which indicates the maximum number of times the teacher can change any answer in the `answerKey` from 'T' to 'F' or vice versa.

The objective is to find the maximum length of a subsequence of consecutive characters in the `answerKey` that can be the same character after performing at most `k` modifications.

Intuition

The intuition behind the solution relies on using a sliding window approach. The sliding window is a technique that can efficiently find a subarray or substring matching a certain condition within a larger array or string. In the context of this problem, the sliding window represents the sequence of consecutive answers that can be made the same (all 'T's or all 'F's) with up to `k` changes.

The main idea is to maintain two pointers, `l` and `r`, which respectively represent the left and right ends of the sliding window. These pointers move along the `answerKey`, expanding to the right (`r` increases) as long as there are characters not matching the desired one and the number of allowed changes `k` has not been exceeded. If the count of non-matching characters within the window exceeds `k`, the window is contracted by moving the left pointer (`l`) forward.

By performing this process twice - once to maximize the number of 'T's and once for 'F's - and taking the maximum of these two values, the solution finds the longest sequence of consecutive identical answers that can be obtained through at most `k` modifications.

The function `get(c, k)` handles this sliding window process by taking a character `c` ('T' or 'F') and the number of allowed changes `k`. It keeps track of the window by incrementing pointers and modifying the allowed changes `k` accordingly. The length of the longest valid window found while scanning with either character is then returned.

Finally, the `maxConsecutiveAnswers` function simply returns the maximum length found between the two separate iterations of the `get` function with 'T' and 'F' as the target characters.

Solution Approach

The solution uses a sliding window algorithm to determine the maximum number of consecutive 'T's or 'F's that can be obtained in the `answerKey`, given that we can change up to `k` answers. The definition of the function `get(c, k)` inside the `Solution` class plays a crucial role in implementing this algorithm.

The variables `l` and `r` are pointers used to denote the left and right bounds of the sliding window, initially set to `-1` since the window starts outside the bounds of the `answerKey` string. The `while` loop inside `get(c, k)` drives the expansion of the window to the right.

For every iteration, `r` is incremented to include the next character in the window. If this character does not match the target character `c`, `k` is decremented, which reflects that we use one operation to change this character to `c`.

If at any point the number of operations used (`k`) becomes negative, this means the current window size cannot be achieved as it would require more than `k` operations to make all characters the same as `c`. To rectify this, the left bound of the window (`l`) is moved to the right (`l += 1`), effectively shrinking the window from the left side. If the character at the new left bound (`answerKey[l]`) was a mismatch and we previously used an operation to adjust it, `k` is incremented since that operation is no longer needed for the new smaller window.

After expanding the window as much as possible without exceeding the maximum allowed modifications (`k`), the function calculates the window's size (`r - l`) and returns this value to the caller.

The solution approach calls `get(c, k)` twice, once with `c = 'T'` and once with `c = 'F'`, to compute the maximum length of consecutive answers that can be obtained for both 'T' and 'F', respectively. The maximum of these two values is returned by the `maxConsecutiveAnswers` as the final answer to the problem.

Essentially, the algorithm leverages the sliding window technique to iterate over `answerKey` while keeping the modifications within the limit imposed by `k`, and dynamically adjusting window bounds to maximize the size of the window which represents a sequence of identical characters.

By returning the maximum between one pass with 'T' as the target character and one pass with 'F', the algorithm ensures that it considers both scenarios—maximizing consecutive 'T's and 'F's—ultimately returning the best possible result.

This solution is efficient as both pointers (left and right) move across the `answerKey` in one direction only, keeping the time complexity linear, O(n), where n is the length of the `answerKey`.

Example Walkthrough

Let's consider a small example to illustrate the solution approach.

Suppose we have: `answerKey = "TFFTFF"` and `k = 1`.

The aim is to find the maximum length of consecutive answers after changing at most one answer ('T' to 'F' or vice versa).

Step-by-Step Process to Find Maximum Consecutive 'T's

Here, we'll go through the process using `get('T', k)`:

1. Initialize pointers `l` and `r` to `-1` so that our sliding window is initially empty.
2. Start scanning the `answerKey` from left to right.
3. The `r` pointer moves right one step. When `r = 0`, the character is 'T', which matches our target, so we don't need to use a change.
4. Move `r` to `1`, the character is 'F', which is not our target, so we decrement `k` by 1 (now `k = 0` because we need one change to make it a 'T').
5. We can't use more changes (since `k = 0`); continue to move `r` to the right.
6. At `r = 2`, we have another 'T', no changes needed.
7. At `r = 3`, we have 'F'. Since we can't change it (`k` is already 0), we need to move `l` up to release a change. Move `l` to `0` and since `answerKey[l]` was a 'T', no `k` increment.
8. Now we can move `r` forward. At `r = 3`, we change 'F' to 'T'. (Now `k = -1`, which is not allowed, so we must adjust `l` again).
9. Move `l` to `1`. Now `k` becomes `0` again because we've released the change at `r = 1`.
10. At `r = 4` and `5`, we have 'F's, but since we have no changes left, we can't include these in our sequence.

The largest sequence of 'T's that we can obtain is from `l + 1` to `r`, which is from index `2` to `3` (subsequence "TT"). Its length is `3 - 1 = 2`.

Step-by-Step Process to Find Maximum Consecutive 'F's

Repeat the same method using `get('F', k)`:

1. Initialize `l` and `r` to `-1`.
2. Scan `answerKey` from left to right, modifying at most `k` characters to 'F'.
3. The process will find that the longest sequence of 'F's, after one modification, can be from index `1` to `5` (subsequence "FFFFF") if we change the 'T' at index `2` to an 'F'.
4. Its length is `5 - 0 = 5`.

Comparing both cases:

- Maximum length for 'T's: `2`
- Maximum length for 'F's: `5`

Final Answer

The final answer is the maximum of these two lengths, which in this example is `5` since the sequence of 'F's is longer.

This small illustration explains how the sliding window algorithm is used to determine the longest possible sequence of consecutive 'T's or 'F's that can be achieved by making at most `k` modifications. By trying both cases separately and returning the maximum value obtained from them, the algorithm yields the optimal solution to the problem.

Python Solution

```python
class Solution:
    def maxConsecutiveAnswers(self, answer_key: str, k: int) -> int:
        # Helper function to calculate the maximum number of consecutive answers
        # with at most k answers being flipped to the character 'char_to_flip'.
        def get_max_consecutive(char_to_flip, k):
            left = right = max_length = 0
            while right < len(answer_key):
                # Increase the count of flips if we have a character that needs to be flipped.
                if answer_key[right] != char_to_flip:
                    k -= 1
                # If the maximum number of flips is exceeded, move the left pointer to the right,
                # decreasing the count of flips if necessary.
                while k < 0:
                    if answer_key[left] != char_to_flip:
                        k += 1
                    left += 1
                # Update the max_length if the current window is larger.
                max_length = max(max_length, right - left + 1)
                # Move the right pointer to the right.
                right += 1
            return max_length

        # Find the max consecutive answers flipping Ts or Fs and return the max of both cases.
        return max(get_max_consecutive('T', k), get_max_consecutive('F', k))

```

Java Solution

```java
class Solution {

    public int maxConsecutiveAnswers(String answerKey, int k) {
        // Find the max length of consecutive answers by calling the get method twice,
        // once for character 'T' and then for 'F'
        return Math.max(getMaxLength('T', k, answerKey), getMaxLength('F', k, answerKey));
    }

    // Helper method to get the max length of consecutive characters in answerKey
    // after at most k characters can be flipped
    public int getMaxLength(char targetChar, int k, String answerKey) {
        int left = 0; // Initialize the left pointer
        int right = 0; // Initialize the right pointer
        int maxLen = 0; // Variable to keep track of maximum length

        // Traverse through the string using the right pointer
        while (right < answerKey.length()) {

            // if the current character does not match the target,
            // decrement k (flip the character)
            if (answerKey.charAt(right++) != targetChar) {
                --k;
            }

            // If we have flipped more than k characters, move left pointer
            // until k is non-negative (backtrack on the string)
            while (k < 0) {
                if (answerKey.charAt(left++) != targetChar) {
                    ++k; // Reset flip counter k because we are undoing the previous flips
                }
            }

            // Calculate the max length of the window
            maxLen = Math.max(maxLen, right - left);
        }

        // Return the max length of consecutive characters
        return maxLen;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to find the maximum number of consecutive answers that can be made 'true' or 'false'
    // by flipping at most k answers.
    int maxConsecutiveAnswers(string answerKey, int k) {
        // Calculate the maximum consecutive sequence for both 'T' and 'F' and return the larger one.
        return max(getMaxConsecutiveByFlipping(answerKey, k, 'T'), getMaxConsecutiveByFlipping(answerKey, k, 'F'));
    }

private:
    // Helper function to determine the maximum number of consecutive characters that match the given character `targetChar`
    // by flipping at most `k` characters that do not match.
    int getMaxConsecutiveByFlipping(const string& answerKey, int k, char targetChar) {
        int left = 0; // Left pointer of the sliding window
        int right = 0; // Right pointer of the sliding window
        int maxConsecutive = 0; // Track the maximum count of consecutive characters found so far

        // Use a sliding window to find the longest sequence that can be made of `targetChar` by flipping at most `k` chars.
        while (right < answerKey.size()) {
            // If the current character is not the target, decrement k (as we would need to flip it).
            if (answerKey[right] != targetChar) --k;

            // Move the right pointer to expand the window.
            ++right;

            // If k is negative, it means we have flipped more than allowed.
            // We must move the left pointer to shrink the window until k is non-negative again.
            while (k < 0) {
                // If the character at the left of the window is not the target, we increment k (as we no longer need to flip this ch
                if (answerKey[left] != targetChar) ++k;
                // Move the left pointer to the right to shrink the window.
                ++left;
            }

            // Update the maximum length if needed.
            maxConsecutive = max(maxConsecutive, right - left);
        }

        // Return the length of the longest sequence after processing the complete string.
        return maxConsecutive;
    }
};

```

Typescript Solution

```typescript
function maxConsecutiveAnswers(answerKey: string, k: number): number {
    // Calculate the length of the answer key
    const lengthOfAnswerKey = answerKey.length;

    // Function to get the maximum count of consecutive 'T' or 'F'
    const getMaxCount = (target: 'T' | 'F'): number => {
        let leftIndex = 0; // initializing the left pointer
        let changesLeft = k; // the allowed number of changes
        for (const char of answerKey) {
            // If the character is not the target, decrement changes left
            if (char !== target) {
                changesLeft--;
            }
            // If no changes are left, move the left pointer forward
            if (changesLeft < 0 && answerKey[leftIndex++] !== target) {
                changesLeft++;
            }
        }
        // The length minus the left index gives the max count for the target
        return lengthOfAnswerKey - leftIndex;
    };

    // The maximum consecutive answers will be the max value for 'T' or 'F'
    return Math.max(getMaxCount('T'), getMaxCount('F'));
}

```

Time and Space Complexity

Time Complexity

The function `get` uses a sliding window approach to count the maximum consecutive answers by flipping a certain number of `T` or `F`. The inner while loop runs once for each character in `answerKey` as both `l` and `r` iterate over the indices from start to end without stepping backwards. Thus, the complexity of the function `get` is linear with respect to the length of the `answerKey`, which is denoted as `n`. Since the function `get` is called twice (once for `T` and once for `F`), the overall time complexity of `maxConsecutiveAnswers` is `O(n)`, as the two linear passes are additive, not multiplicative.

Space Complexity

The space complexity of the code is `O(1)`. The function `get` uses only a fixed number of variables (`l`, `r`, and `k`), which doesn't depend on the size of the input `answerKey`. There are no data structures used that scale with the input size, and the memory usage remains constant regardless of the length of `answerKey`.