340. Longest Substring with At Most K Distinct Characters

Medium | Hash Table | String | Sliding Window

Problem Description

In this problem, we are given a string `s` and an integer `k`. Our task is to find the length of the longest substring (which is a contiguous block of characters in the string) within `s` that contains at most `k` distinct characters.

A distinct character means that no matter how many times the character appears in the substring, it is counted only once. For example, if `k = 2`, the substring "aabbc" has 3 distinct characters ('a', 'b', and 'c'), thus, it does not meet the requirement.

The goal here is to achieve this while maximizing the length of the substring. A substring could range from containing a single character up to the length of the entire string, if `k` is sufficiently large to cover all distinct characters in the string `s`.

Intuition

The core intuition behind the solution is to use the two-pointer technique, or more specifically, the sliding window approach. Here is the thinking process that leads us to this approach:

1. We need to examine various substrings of `s` efficiently without having to re-scan the string repeatedly.
2. We can start with a window (or two pointers) at the beginning of the string and expand the window to the right until we exceed `k` distinct characters within the window.
3. Upon exceeding `k` distinct characters, we need to move the left side of the window to the right to potentially drop one of the distinct characters out of the count.
4. We keep track of the count of distinct characters in the current window using a data structure, such as a counter (dictionary), that is updated when the window is adjusted.
5. As we expand and contract the window, we keep a record of the maximum window size (i.e., substring length) that has appeared so far that contains at most `k` distinct characters. This is the number we want to return.

Now, the implementation uses a counter and two indices, `i` and `j`, where `i` is the end of the sliding window, and `j` is the start. We iterate over the string with `i`:

- We include the character at position `i` in the current window by incrementing its count in the counter.
- If adding the current character has led to more than `k` distinct characters in the window, we increment `j`, effectively reducing the size of the window from the left, until the number of distinct characters drops to `k` or lower.
- At each step, we calculate the length of the current window (`i - j + 1`) and update the answer if it's the largest such length we have seen so far.
- This process continues until `i` has reached the end of the string.

The sliding window is moved across the string efficiently to identify the longest substring that satisfies the condition of having at most `k` distinct characters, thus leading us to the solution.

Solution Approach

The solution code implements the sliding window pattern using two pointers to keep track of the current window within the string `s`. This pattern is completed using the Python `Counter` from the collections module as the key data structure to keep counts of each character in the current window.

Here is how it is done step by step:

1. We initialize a `Counter` object, which is a dictionary that will keep track of the counts of individual characters in the current sliding window.

2. Two pointers, `i` and `j`, are created. `i` is used to traverse the string `s` character by character from the start to the end, while `j` is used to keep track of the start of the sliding window. `j` starts at 0 and moves rightward to narrow the window whenever necessary.

3. We initialize a variable `ans` to keep track of the maximum length of the substring found that meets the criteria.

4. We start iterating over the characters of the string `s`, with `i` acting as the end boundary of the window. For each character `c` at index `i`, we:

   - Increment its count in the `Counter` by 1 (`cnt[c] += 1`).

   - Check if the number of distinct characters in the `Counter` has exceeded `k`. This is done by evaluating `len(cnt) > k`. As long as this condition is true, meaning we have more than `k` distinct characters, we need to shrink the window from the left side by increasing `j`.

     - In the inner `while` loop, decrease the count of the character at the beginning of the window (`cnt[s[j]] -= 1`).
     - If the count of that character drops to 0, we remove it from the `Counter` (`cnt.pop(s[j])`), as it's no longer part of the window.
     - We increment `j` to shrink the window from the left.
5. After adjusting the size of the window (if it was needed), we calculate the length of the current window (`i - j + 1`) and update `ans` to be the maximum of its current value and this newly calculated length.

6. After iterating through all characters of `s`, the process concludes, and the final value of `ans` contains the length of the longest substring with at most `k` distinct characters. This value is returned as the result.

Through this method, we avoid unnecessary re-scans of the string and implement an O(n) time complexity algorithm, where `n` is the length of the string `s`. By dynamically adjusting the sliding window and using the `Counter` to track distinct characters, we achieve an efficient approach to solving the problem as outlined.

Example Walkthrough

Let's assume we have a string `s = "aabcabb"`, and the integer `k = 2`. We want to find the length of the longest substring with at most `k` distinct characters.

Here is a step-by-step walkthrough of the solution approach applied to this example:

1. **Initialize the Counter and Pointers**: Create a `Counter` object, `cnt`, to count characters in the current window. Set `i`, `j` to 0 to represent the start and end of the sliding window, and `ans` to 0 as the longest substring length found so far.

2. **Start Iterating with Pointer `i`**: Move `i` from the start to the end of the string `s`.

   `i = 0`: `cnt = {'a': 1}`, substring is `a`, `ans = 1`.

   `i = 1`: `cnt = {'a': 2}`, substring is `aa`, `ans = 2`.

   `i = 2`: `cnt = {'a': 2, 'b': 1}`, substring is `aab`, `ans = 3`.

3. **Exceeding `k` Distinct Characters**: At the next character, we check if the number of distinct characters will exceed `k`.

   `i = 3`: We attempt to insert `c` into `cnt`. Doing so would increase distinct character count to `3` (`cnt = {'a': 2, 'b': 1, 'c': 1}`), which is greater than `k`. We need to move `j` right until we get at most `k` distinct characters.

4. **Shrinking the Window**:

   Start incrementing `j`, decreasing the count of the character `s[j]` in `cnt`, and remove the character from `cnt` if its count reaches 0.

   `j = 1`: We decrease the count of `a` and update `cnt = {'a': 1, 'b': 1, 'c': 1}`. We still have more than `k` distinct characters.

   `j = 2`: We decrease the count of `a` and remove it from `cnt` (`cnt = {'b': 1, 'c': 1}`). Now we have `k` distinct characters.

   After shrinking, the substring is now `bc` (`s[2:4]`), and `ans` remains `3`.

5. **Continue Process**:

   `i = 4`: Add `a` to `cnt`, `cnt = {'b': 1, 'c': 1, 'a': 1}`, substring is `bca`, `ans` remains `3`.

   `i = 5`: Add `b` to `cnt`, `cnt = {'b': 2, 'c': 1, 'a': 1}`. We have exceeded `k` again, so start moving `j`.

   `j = 3`: Reduce count of `b`, `cnt = {'b': 1, 'c': 1, 'a': 1}`. Still too many distinct characters.

   `j = 4`: Remove `c`, `cnt = {'b': 1, 'a': 1}`, and now the substring is `ab` (`s[4:6]`), update `ans` to `2`.

   `i = 6`: Add `b`, `cnt = {'b': 2, 'a': 1}`, and substring is `abb` (`s[4:7]`). Now, update `ans` to `3`, as this is the length of the current window.

6. **Finish Iteration**:

   After the final iteration, `ans` contains the length of the longest satisfactory substring, which is `3` in this case, representing the substring `aab` or `abb`.

By using the sliding window technique, the algorithm efficiently finds the longest substring that satisfies the given constraint with a time complexity of O(n), where n is the length of the string `s`.

Python Solution

```python
from collections import Counter

class Solution:
    def lengthOfLongestSubstringKDistinct(self, s: str, k: int) -> int:
        # Initialize a counter to keep track of the frequency of each character
        char_count = Counter()

        # Length of the input string
        n = len(s)

        # Initialize the answer and the start index (j) of the current window
        max_length = start_index = 0

        # Enumerate over the characters of the string with index (i) and character (char)
        for i, char in enumerate(s):
            # Increment the count of the current character
            char_count[char] += 1

            # If the number of distinct characters exceeds k, shrink the window
            while len(char_count) > k:
                # Decrement the count of the character at the start index
                char_count[s[start_index]] -= 1

                # If the count goes to zero, remove the character from the counter
                if char_count[s[start_index]] == 0:
                    del char_count[s[start_index]]

                # Move the start index forward
                start_index += 1

            # Update the maximum length found so far
            max_length = max(max_length, i - start_index + 1)

        # Return the maximum length of substring with at most k distinct characters
        return max_length

# The code can now be run with an instance of the Solution class
# For example: Solution().lengthOfLongestSubstringKDistinct("eceba", 2) should return 3

```

Java Solution

```java
import java.util.HashMap;
import java.util.Map;

class Solution {

    public int lengthOfLongestSubstringKDistinct(String s, int k) {
        // Map to store the frequency of each character in the current window
        Map<Character, Integer> charCountMap = new HashMap<>();
        int n = s.length();
        int longestSubstringLength = 0; // variable to store the length of the longest substring
        int left = 0; // left pointer for the sliding window

        // Iterate through the string using the right pointer of the sliding window
        for (int right = 0; right < n; ++right) {
            // Step 1: Update the count of the current character
            char currentChar = s.charAt(right);
            charCountMap.put(currentChar, charCountMap.getOrDefault(currentChar, 0) + 1);

            // Step 2: Shrink the window from the left if count map has more than 'k' distinct characters
            while (charCountMap.size() > k) {
                char leftChar = s.charAt(left);
                charCountMap.put(leftChar, charCountMap.get(leftChar) - 1);
                // Remove the character from map when count becomes zero
                if (charCountMap.get(leftChar) == 0) {
                    charCountMap.remove(leftChar);
                }
                left++; // shrink the window from the left
            }

            // Step 3: Update the longest substring length if the current window is larger
            longestSubstringLength = Math.max(longestSubstringLength, right - left + 1);
        }

        return longestSubstringLength; // Return the length of the longest substring found
    }
}

```

C++ Solution

```cpp
#include <string>
#include <unordered_map>
#include <algorithm>

class Solution {
public:
    // Finds the length of the longest substring with at most k distinct characters
    int lengthOfLongestSubstringKDistinct(std::string s, int k) {
        std::unordered_map<char, int> charCountMap; // Map to store character counts

        int maxSubstringLength = 0; // Maximum length of substring found
        int leftPointer = 0; // Left pointer for sliding window
        int n = s.size(); // Length of the input string

        for (int rightPointer = 0; rightPointer < n; ++rightPointer) {
            // Increase char count for the current position
            charCountMap[s[rightPointer]]++;

            // If we have more than k distinct chars, contract the window from the left
            while (charCountMap.size() > k) {
                // Decrease the count of the char at the left pointer
                charCountMap[s[leftPointer]]--;

                // If the count drops to 0, remove it from the map
                if (charCountMap[s[leftPointer]] == 0) {
                    charCountMap.erase(s[leftPointer]);
                }

                // Move the left pointer to the right
                ++leftPointer;
            }

            // Update maxSubstringLength if we've found a larger window
            maxSubstringLength = std::max(maxSubstringLength, rightPointer - leftPointer + 1);
        }

        return maxSubstringLength; // Return the max length found
    }
};

```

Typescript Solution

```typescript
// Imports a generic map class to emulate the unordered_map feature in C++
import { Map } from "typescript-collections";

// Finds the length of the longest substring with at most k distinct characters
function lengthOfLongestSubstringKDistinct(s: string, k: number): number {
    let charCountMap = new Map<char, number>(); // Map to store character counts

    let maxSubstringLength = 0; // Maximum length of substring found
    let leftPointer = 0; // Left pointer for sliding window
    let n = s.length; // Length of the input string

    for (let rightPointer = 0; rightPointer < n; rightPointer++) {
        // Increase char count for the current character
        let count = charCountMap.getValue(s[rightPointer]) || 0;
        charCountMap.setValue(s[rightPointer], count + 1);

        // If we have more than k distinct characters, contract the window from the left
        while (charCountMap.size() > k) {
            // Decrease the count of the character at the left pointer
            let leftCharCount = charCountMap.getValue(s[leftPointer]) || 0;
            charCountMap.setValue(s[leftPointer], leftCharCount - 1);

            // If the count drops to 0, remove it from the map
            if (charCountMap.getValue(s[leftPointer]) === 0) {
                charCountMap.remove(s[leftPointer]);
            }

            // Move the left pointer to the right
            leftPointer++;
        }

        // Update maxSubstringLength if we've found a larger window
        maxSubstringLength = Math.max(maxSubstringLength, rightPointer - leftPointer + 1);
    }

    return maxSubstringLength; // Return the max length found
}

```

Time and Space Complexity

The given code snippet defines a function that determines the length of the longest substring with no more than `k` distinct characters in a given string `s`.

Time Complexity

The time complexity of the function can be analyzed as follows:

- There are two pointers (`i` and `j`) that traverse the string `s` only once. The outer loop runs for all characters from position 0 to `n - 1` where `n` is the length of the string `s`, hence contributing O(n) to the time complexity.
- Inside the loop, there is a `while` loop that shrinks the sliding window from the left when the number of distinct characters exceeds `k`. This `while` loop does not run for each element in `s` multiple times. Each character is removed from the window only once, accounting for another O(n) over the whole run of the algorithm.

Therefore, the total time complexity of the algorithm is `O(n)` where `n` is the number of characters in the input string `s`.

Space Complexity

The space complexity of the function can be analyzed as follows:

- A `Counter` object is used to keep track of the frequency of each character in the current window. In the worst case, the counter could store a distinct count for every character in the string `s`. However, since the number of distinct characters is limited by `k`, the `Counter` would hold at most `k` elements.

Therefore, the space complexity of the algorithm is `O(k)` where `k` is the maximum number of distinct characters that the substring can have.