1208. Get Equal Substrings Within Budget

Medium String Binary Search Prefix Sum Sliding Window Leetcode Link

Problem Description

In this problem, we are given two strings `s` and `t` of equal length, and an integer `maxCost`. The objective is to convert string `s` into string `t` by changing characters at corresponding positions. Each change comes with a cost, which is the absolute difference in the ASCII values of the characters in the same position in both strings. We aim to find the maximum length of a substring of `s` that can be transformed into the corresponding substring of `t` without exceeding a given `maxCost`. If no substring from `s` can be changed within the cost constraints, the function should return `0`.

Intuition

The solution to this problem involves using the two-pointer technique, which is commonly applied to array or string manipulation problems where a running condition—like a fixed sum or cost—needs to be maintained.

Here, the idea is to iterate over the strings with two pointers, `i` and `j`, marking the beginning and end of the current substring being considered. We start with both pointers at the beginning of the strings and calculate the cost of changing the current characters of `s` to match `t`. This cost is added to a running total `sum`. If at any point the `sum` exceeds `maxCost`, we need to move the `j` pointer (the start of the substring) to the right, effectively shortening the substring and reducing our total cost by removing the cost of changing the character at the `j`th position.

We continue to move the `i` pointer to the right, expanding the substring, and checking if the conversion cost still stays within `maxCost`. After each step, we update our result `ans` with the maximum length of a valid substring found so far, which is the difference between our two pointers plus one (to account for zero-based indexing). This process continues until we've considered every possible substring starting at every possible point in `s`.

The intuition behind the sliding window is that we are looking for the longest possible contiguous sequence (the window) within `s` and `t` where the total conversion cost does not exceed `maxCost`. By sliding the window along the strings, we can explore all options in linear time without the need for nested loops, which would significantly increase the computational complexity.

Solution Approach

The provided solution code implements the sliding window technique to track the longest substring that can be changed within the given `maxCost`.

Here's a step-by-step breakdown of the algorithm used in the solution:

1. Initialize:
   - `n` to store the length of the strings `s` and `t`, ensuring that the length is the same for both.
   - A variable `sum` to keep track of the cumulative cost of changing characters from `s` to `t`.
   - Two pointers, `j` to mark the start and `i` as the current position in the string, which together define the bounds of the current substring.
   - A variable `ans` to store the maximum length of a substring that meets the cost condition.
2. Iterate over each character in both strings using the `i` pointer. For every iteration:
   - Calculate the cost (absolute character difference) between the `i`th character of `s` and `t`, and add it to the `sum`.
3. If at any point the `sum` is greater than `maxCost`, enter a `while` loop that will:
   - Subtract the cost associated with the `j`th character (start of the current substring) from `sum`.
   - Increment `j` to effectively shrink the window and reduce the cost, as we are now removing the starting character of our substring.
4. After adjusting `j` to ensure `sum` doesn't exceed `maxCost`, calculate the current substring length by `i - j + 1` (accounting for zero-indexing), and update `ans` with the maximum length found so far.
5. Continue iterating until all potential substrings have been considered. As a result, `ans` will hold the length of the longest possible substring that can be transformed from `s` into `t` without exceeding the `maxCost`.

Notice that there are no nested loops; the two pointers move independently, which ensures that the complexity of the solution is `O(n)`, where `n` is the length of the strings. This single pass through the data, while adjusting the window's starting point on the fly, exemplifies the efficiency of the sliding window algorithm in such problems.

The code finishes execution once the end of string `s` is reached, and returns the length of the longest substring with the transformation cost within the given budget, `maxCost`.

Example Walkthrough

Let's consider an example with strings `s = "abcde"`, `t = "axcyz"`, and `maxCost = 6`. We need to find the length of the longest substring we can change from `s` to `t` without exceeding the `maxCost`.

1. **Initial Setup**:
   - Length `n = 5` (since both strings are of length 5).
   - Sum `sum = 0`, to keep track of the cumulative cost.
   - Pointers `i = 0` and `j = 0`, marking the current character and the start of the substring, respectively.
   - Answer `ans = 0`, to store the maximum length of a valid substring.
2. **First Character**:
   - `i = 0`, comparing 'a' from `s` with 'a' from `t`, the cost is `0` (since they are the same).
   - `sum = 0`, and `sum <= maxCost`.
   - Therefore, `ans` becomes `i - j + 1 = 1`.
3. **Second Character**:
   - `i = 1`, comparing 'b' from `s` with 'x' from `t`, the cost is `|'b' - 'x'| = 22`.
   - `sum = 22`, which is greater than `maxCost`.
   - Move `j` to the right (`j = 1`) to remove the cost of the first character.
   - Since `sum` now exceeds `maxCost`, we cannot include this character in our substring, and `ans` remains `1`.
4. **Third Character**:
   - With `j` now at 1 and `i` incrementing to 2, we compare 'c' with 'c' and the cost is `0`.
   - `sum = 0` (we discarded the previous sum since we moved `j`), and `sum <= maxCost`.
   - `ans` is updated to `i - j + 1 = 2`.
5. **Fourth Character**:
   - `i = 3`, comparing 'd' from `s` with 'y' from `t`, the cost is `|'d' - 'y'| = 21`.
   - `sum = 21`, which is still within `maxCost`.
   - `ans` is updated to `i - j + 1 = 3`.
6. **Fifth Character**:
   - `i = 4`, comparing 'e' with 'z' gives a cost of `|'e' - 'z'| = 21`.
   - Adding this cost makes `sum = 42`, which exceeds `maxCost`.
   - We must move `j` right again; now `j` should be at position 3, where 'd' is located in `s`, and subtract the cost of 'd' and 'y'.
   - This brings `sum` to 21 again (as the cost for 'e' and 'z' is 21), and `ans` remains `3`.

At the end of our iteration, `ans` holds the value `3`, which is the length of the longest valid substring that could be changed from `s` to `t` without exceeding `maxCost`. Therefore, the answer to this example is `3`.

Python Solution

```python
class Solution:
    def equalSubstring(self, s: str, t: str, max_cost: int) -> int:
        # Initialize variables:
        # n - length of the input strings
        # total_cost - accumulated cost of transforming s into t
        # start_index - start index for the current substring
        # max_length - the maximum length of a substring that satisfies the cost condition
        n = len(s)
        total_cost = 0
        start_index = 0
        max_length = 0

        # Iterate over the characters in both strings
        for end_index in range(n):
            # Calculate the cost for the current index by taking the absolute difference of
            # the character codes of the current characters of s and t
            total_cost += abs(ord(s[end_index]) - ord(t[end_index]))

            # If the total cost exceeds the max_cost, shrink the window from the left till
            # the total_cost is less than or equal to max_cost
            while total_cost > max_cost:
                total_cost -= abs(ord(s[start_index]) - ord(t[start_index]))
                start_index += 1

            # Update max_length if the length of the current substring (end_index - start_index + 1)
            # is greater than the previously found max_length
            max_length = max(max_length, end_index - start_index + 1)

        # Return the maximum length of a substring that can be obtained under the given cost
        return max_length
```

Java Solution

```java
class Solution {
    public int equalSubstring(String s, String t, int maxCost) {
        // Length of the input strings
        int length = s.length();

        // This will hold the cumulative cost of transformations
        int cumulativeCost = 0;

        // This will keep track of the maximum length substring that meets the condition
        int maxLength = 0;

        // Two-pointer technique:
        // Start and end pointers for the sliding window
        for (int start = 0, end = 0; end < length; ++end) {
            // Calculate and add the cost of changing s[end] to t[end]
            cumulativeCost += Math.abs(s.charAt(end) - t.charAt(end));

            // If the cumulative cost exceeds maxCost, shrink the window from the start
            while (cumulativeCost > maxCost) {
                // Remove the cost of the starting character as we're about to exclude it
                cumulativeCost -= Math.abs(s.charAt(start) - t.charAt(start));
                // Move the start pointer forward
                ++start;
            }

            // Update the maximum length found so far (end - start + 1 is the current window size)
            maxLength = Math.max(maxLength, end - start + 1);
        }

        // Return the final maximum length found
        return maxLength;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    int equalSubstring(string s, string t, int maxCost) {
        int length = s.size(); // Stores the length of the input strings
        int maxLength = 0; // Stores the maximum length of equal substring within maxCost
        int currentCost = 0; // Current cost of making substrings equal
        // Two pointers for the sliding window approach
        int start = 0; // Start index of the current window
        int end; // End index of the current window

        // Iterate through the string with the end pointer of the sliding window
        for (end = 0; end < length; ++end) {
            // Calculate the cost of making s[end] and t[end] equal and add it to currentCost
            currentCost += abs(s[end] - t[end]);

            // If the currentCost exceeds maxCost, shrink the window from the start
            while (currentCost > maxCost) {
                currentCost -= abs(s[start] - t[start]); // Reduce the cost of the start character
                ++start; // Move the start pointer forward to shrink the window
            }

            // Calculate the length of the current window and update maxLength if necessary
            maxLength = max(maxLength, end - start + 1);
        }

        return maxLength; // Return the maximum length of equal substring within maxCost
    }
};
```

Typescript Solution

```typescript
function equalSubstring(s: string, t: string, maxCost: number): number {
    const length: number = s.length; // Stores the length of the input strings
    let maxLength: number = 0; // Stores the maximum length of equal substring within maxCost
    let currentCost: number = 0; // Current cost of making substrings equal
    // Two pointers for the sliding window approach
    let start: number = 0; // Start index of the current window
    let end: number; // End index of the current window

    // Iterate through the string with the end pointer of the sliding window
    for (end = 0; end < length; ++end) {
        // Calculate the cost of making s[end] and t[end] equal and add it to currentCost
        currentCost += Math.abs(s.charCodeAt(end) - t.charCodeAt(end));

        // If the currentCost exceeds maxCost, shrink the window from the start
        while (currentCost > maxCost) {
            currentCost -= Math.abs(s.charCodeAt(start) - t.charCodeAt(start)); // Reduce the cost of the start character
            ++start; // Move the start pointer forward to shrink the window
        }

        // Calculate the length of the current window and update maxLength if necessary
        maxLength = Math.max(maxLength, end - start + 1);
    }

    return maxLength; // Return the maximum length of equal substring within maxCost
}
```

Time and Space Complexity

Time Complexity

The time complexity of the given code is `O(n)`, where `n` is the length of the strings `s` and `t`. This linear time complexity arises from the single for loop that iterates over each character of the two strings exactly once. Inside the loop, there are constant-time operations such as calculating the absolute difference of character codes and updating the `sum`. The while loop inside the for loop does not add to the overall time complexity since it only moves the `j` pointer forward and does not result in reprocessing of any character — the total number of operations in the while loop across the entire for loop is proportional to `n`.

Space Complexity

The space complexity of the given code is `O(1)` because the extra space used by the algorithm does not grow with the input size `n`. The variables `sum`, `j`, and `ans` use a constant amount of space, as do the indices `i` and `n` which store fixed-size integer values. There are no data structures used that scale with the size of the input.