828. Count Unique Characters of All Substrings of a Given String

Hard · Hash Table · String · Dynamic Programming · Leetcode Link

Problem Description

The problem is about counting unique characters within all possible substrings of a given string `s`. The goal is to compute the sum of the number of unique characters (`countUniqueChars`) for each substring in `s`. A unique character in a substring is a character that appears exactly once in it.

To clarify the requirement, let's consider if `s = "ABC"`, we then look at each substring:

- "A" has 1 unique character,
- "B" has 1 unique character,
- "C" has 1 unique character,
- "AB" has 2 unique characters,
- "BC" has 2 unique characters,
- "ABC" has 3 unique characters.

Adding these counts together, the sum would be 1 + 1 + 1 + 2 + 2 + 3 = 10.

The challenge of the problem is that a straightforward approach to finding and counting unique characters for each substring could be very inefficient, especially when `s` is long, because the number of substrings grows quadratically with the length of `s`.

Intuition

To efficiently solve this problem, we need an approach that avoids redundantly counting unique characters in overlapping substrings. This is where the intuition for the provided solution comes into play.

We create a dictionary, `d`, that maps each character in the string `s` to the list of indices where it appears. This allows us to quickly understand where each character contributes to the uniqueness in the substrings.

By iterating through the dictionary, we can consider each character independently and determine its contribution to the overall count. A key insight is that the contribution of a character at position `i` to the sum is determined by the distance to the previous occurrence of the same character and the distance to the next occurrence of the same character. This is because a character contributes to the uniqueness of all substrings that start after its previous occurrence and end before its next occurrence.

For example, if character 'A' appears at indices `3`, `5`, and `8` in `s`, then for the substring `s[4:6]` (which includes character at index 5), 'A' is unique. The number of such substrings is the product of the distance from the previous index (`5 - 3`) and the distance to the next index (`8 - 5`).

The solution efficiently calculates the contribution of each character by iterating through the list of indices in `v` (enhanced with start and end markers at `-1` and `len(s)`, respectively) for each character, multiplying the distances from the current index to its previous and next, and summing up these products to get the total count.

Overall, the intuition is to transform the original problem into calculating character contributions based on their positions, rather than evaluating each substring individually.

Solution Approach

The solution uses a combination of dictionary and list data structures alongside some clever indexing to solve the problem efficiently.

Here's a step-by-step explanation:

1. **Dictionary Creation**: Create a dictionary `d` where each key-value pair consists of a character from the string and a list of indices where that character appears. This is achieved through enumeration of string `s`.
2. **Initializing the Answer**: Start with a variable `ans` initialized to `0`. This variable will hold the sum of unique characters counts for all substrings.
3. **Iterate Over Each Character's Occurrences**:
   - For each character, we get its list of indices where it appears in the string and add two sentinel indices at the beginning and end `-1` and `len(s)`. These sentinels represent fictive occurrences before the start and after the end of the string to handle edge cases for the first and last characters.
   - Recursive Sub-problem Identification: Each unique occurrence of a character can be viewed as the center of a range extending to its previous and next occurrences. This character is unique in all substrings that start after its previous occurrence and end before its next occurrence.
4. **Accumulate Contribution**:
   - Iterate over the list of indices (now with added sentinels) for the given character. For index `i` that corresponds to a true occurrence of the character (not the `-1` or `len(s)`), calculate:
     - The distance to the previous index: `left = v[i] - v[i - 1]`.
     - The distance to the next index: `right = v[i + 1] - v[i]`.
   - The contribution of the character at this particular index to the overall unique character count is given by `left * right`. It represents the count of substrings where this character is unique, by combining the number of possible start points (left) with the number of possible endpoints (right).
5. **Sum Up Contributions**:
   - Add each character's contributions to the `ans` variable.
6. **Return the Result**: After processing all characters, return the final accumulated value in `ans`, which represents the sum of unique characters for all substrings of `s`.

This solution approach leverages the indexing pattern to avoid redundant calculations by using the distances between character occurrences to infer the number of substrings where those characters are unique. It's a great example of how considering the problem from a different angle can lead to an efficient approach that avoids brute-force inefficiency.

Example Walkthrough

Let's go through an example to illustrate the solution approach using the string `s = "ABA"`.

1. **Dictionary Creation**:
   - Iterate over characters of the string: `A`, `B`, `A`.
   - Create a dictionary `d` where we record the indices of each character: `d = {'A': [0, 2], 'B': [1]}`.
2. **Initializing the Answer**:
   - Initialize `ans` to `0`.
3. **Iterate Over Each Character's Occurrences**:
   - We add sentinel values to the list of indices for each character in `d`. The updated lists will be `{'A': [-1, 0, 2, 3], 'B': [-1, 1, 3]}`.
4. **Accumulate Contribution**:
   - For character `A`:
     - For the first true occurrence at index `0`: The left distance is `0 - (-1) = 1` and the right distance is `2 - 0 = 2`. The contribution is `1 * 2 = 2`.
     - For the second true occurrence at index `2`: The left distance is `2 - 0 = 2` and the right distance is `3 - 2 = 1`. The contribution is `2 * 1 = 2`.
   - The total contribution for `A` is `2 + 2 = 4`.
   - For character `B`:
     - For the true occurrence at index `1`: The left distance is `1 - (-1) = 2` and the right distance is `3 - 1 = 2`. The contribution is `2 * 2 = 4`.
   - The total contribution for `B` is `4`.
5. **Sum Up Contributions**:
   - Sum up the contributions: `4 (from A) + 4 (from B) = 8`.
   - Update `ans` to `8`.
6. **Return the Result**:
   - After processing all characters, return the accumulated value in `ans`, which is `8`.

This example demonstrates the efficiency of the solution approach, which counts the unique characters in all substrings without directly examining each substring.

Python Solution

```python
from collections import defaultdict

class Solution:
    def unique_letter_string(self, input_string: str) -> int:
        # Create a default dictionary to store the indices of each character.
        index_map = defaultdict(list)

        # Iterate through the characters in the string, along with their indices.
        for index, character in enumerate(input_string):
            index_map[character].append(index)

        # Initialize the answer to 0.
        unique_count = 0

        # Iterate through the values in the index map dictionary.
        for indices in index_map.values():
            # Add pseudo-indices at the beginning and end.
            indices = [-1] + indices + [len(input_string)]

            # Iterate through the indices of the current character.
            for i in range(1, len(indices) - 1):
                # Calculate the contribution of each index to the unique count.
                # The idea is that for each index, we count contributions
                # from the previous occurrence to the current (v[i] - v[i-1]),
                # and from the current to the next occurrence (v[i+1] - v[i]).
                unique_count += (indices[i] - indices[i - 1]) * (indices[i + 1] - indices[i])

        # Return the total count of unique substrings.
        return unique_count

```

Java Solution

```java
class Solution {
    public int uniqueLetterString(String s) {
        // Create a list of lists to keep track of the occurrences of each letter
        List<Integer>[] indexList = new List[26];
        // Initialize lists for each character 'A' to 'Z'
        Arrays.setAll(indexList, x -> new ArrayList<>());

        // Add a starting index -1 for each character,
        // representing the position before the start of the string
        for (int i = 0; i < 26; ++i) {
            indexList[i].add(-1);
        }

        // Iterate over the string and add the index of each character to the corresponding list
        for (int i = 0; i < s.length(); ++i) {
            indexList[s.charAt(i) - 'A'].add(i);
        }

        // Initialize a variable to hold the sum of unique letter strings
        int ans = 0;

        // Iterate through each list in indexList
        for (var occurences : indexList) {
            // Add the length of the string as the last index for each character
            occurences.add(s.length());

            // Calculate contributions for each index
            for (int i = 1; i < occurences.size() - 1; ++i) {
                // The count for a unique letter string is determined by the product of the distance to
                // the previous and next occurrence of the same character
                ans += (occurences.get(i) - occurences.get(i - 1)) * (occurences.get(i + 1) - occurences.get(i));
            }
        }

        // Return the total sum of unique letter strings
        return ans;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to calculate the sum of counts of unique characters in all substrings of the given string.
    int uniqueLetterString(string s) {
        // Create a 2D vector with 26 rows to store the indices of each character's occurrence.
        // Initialize the first index as -1, which is used as a dummy index for calculation.
        vector<vector<int>> index(26, {-1});

        // Loop through the given string to fill in the actual indices of each character.
        for (int i = 0; i < s.size(); ++i) {
            index[s[i] - 'A'].push_back(i);
        }

        // Initialize the counter for the answer.
        int count = 0;

        // Loop through the 2D vector to calculate the count for each character.
        for (auto& indices : index) {
            // Push the dummy index equal to the length of the string for the calculation.
            indices.push_back(s.size());

            // Loop through each group of indices for the character.
            for (int i = 1; i < indices.size() - 1; ++i) {
                // Calculate the contribution of the character at position 'i' and add to the answer.
                // The multiplier is the number of substrings where this character is unique.
                count += (indices[i] - indices[i - 1]) * (indices[i + 1] - indices[i]);
            }
        }

        // Return the total count of unique characters in all substrings of the string.
        return count;
    }
};

```

Typescript Solution

```typescript
// Function to calculate the sum of counts of unique characters in all substrings of the given string.
function uniqueLetterString(s: string): number {
    // Create a 2D array with 26 elements to store the indices of each character's occurrence.
    // Initialize the first index as -1, which is used as a dummy index for calculation.
    let indices: number[][] = Array.from({ length: 26 }, () => [-1]);

    // Loop through the given string to fill in the actual indices of each character.
    for (let i = 0; i < s.length; i++) {
        indices[s.charCodeAt(i) - 'A'.charCodeAt(0)].push(i);
    }

    // Initialize the counter for the answer.
    let count: number = 0;

    // Loop through the 2D array to calculate the count for each character.
    indices.forEach((charIndices: number[]) => {
        // Push the dummy index equal to the length of the string for the calculation.
        charIndices.push(s.length);

        // Loop through each group of indices for the character.
        for (let i = 1; i < charIndices.length - 1; i++) {
            // Calculate the contribution of the character at position 'i' and add to the answer.
            // The contribution is the number of substrings where this character is unique.
            count += (charIndices[i] - charIndices[i - 1]) * (charIndices[i + 1] - charIndices[i]);
        }
    });

    // Return the total count of unique characters in all substrings of the string.
    return count;
}

```

Time and Space Complexity

Time Complexity

The given Python code computes the number of substrings where each character occurs exactly once. To understand the time complexity, let's analyze the steps in the code:

1. A dictionary `d` is created using a `defaultdict` to store positions of each character in string `s`.
2. The string `s` is enumerated over once, so this is `O(n)` where `n` is the length of the string `s`.
3. For each character, the positions stored are iterated over in a nested loop to calculate the contribution of each character occurrence towards the unique substrings.
4. There are `m` characters in string `s`, and the nested loop iterates over `k_i` positions for the `i`-th character (`k_i` is the number of times the `i`-th character appears in `s`). This results in a time complexity of `O(m * k_i)`.

Hence, the overall time complexity is `O(n + m * k_i)`. However, since a character cannot appear more times than the string length `n`, and there are at most "26" English letters, the `m` and `k_i` can be bound by `n`, leading to a simplified bound of `O(n^2)`.

Space Complexity

For space complexity,

1. A dictionary `d` is used to store the index positions for each character in `s`. If `s` contains all unique characters, the space complexity of this operation would be `O(n)`.
2. Temporary lists are created for the positions of individual characters with two extra elements for boundary positions; however, their impact on space complexity is negligible as they don't grow with `n`.

The predominant factor is the space taken by `d` which is `O(n)`.

So, the space complexity of the code is `O(n)`.

Note that the auxiliary storage created within the loop is small and does not exceed the length of the string `s`. Since there are a fixed number of english characters, this does not affect the overall space complexity significantly.