2168. Unique Substrings With Equal Digit Frequency

Medium Hash Table String Counting Hash Function Rolling Hash Leetcode Link

Problem Description

The problem provides a digit string `s` and asks us to find the number of unique substrings where every digit appears the same number of times. A digit string means the string contains only numeric characters between '0' and '9'.

For example, if the string is `1211`, there are several substrings where every digit appears an equal number of times: `1`, `2`, `11`, `21`, `12`. Note that `1211` as a whole does not count because '1' appears three times while '2' appears only once.

The challenge is to calculate the count of such substrings without counting any substrings more than once, no matter where they appear in the string.

Intuition

To solve this problem, the intuition is to consider every possible substring of `s` and check if all digits present in the substring appear with the same frequency. A brute force approach might involve repeatedly checking the frequency of digits within each potential substring, which would be inefficient.

To optimize this, we make use of prefix sums. A prefix sum array can help us quickly determine the frequency of each digit in any substring of `s`. With this, we can deduce the number of times a digit appears in any substring by subtracting the prefix sum up to its starting point from the prefix sum up to its ending point.

Here's the reasoning behind the solution steps:

1. Construct a prefix sum matrix `presum` where `presum[i][j]` gives the total count of digit `j` in the string `s[0...i-1]`. This is built for all digits from 0 to 9 and for each prefix of `s`.
2. Iterate through all possible substrings of `s` using two nested loops. Using indexes `i` and `j`, we can define a substring `s[i : j + 1]`.
3. For each substring, use the prefix sums to determine the count of each digit and add it to a set only if they all have the same count, ensuring that all digits must appear the same number of times for the substring to be valid. This is checked by the `check` function.
4. Since a set is used, duplicate substrings are inherently avoided.

Finally, the length of the set gives us the total count of unique valid substrings.

Solution Approach

The solution uses a few important concepts such as prefix sum arrays, set operations, and a check function to validate each substring. Here's a step-by-step implementation:

1. **Prefix Sum Array**: A 2D array `presum` is initialized, where `presum[i][j]` represents the total count of the digit `j` from index `0` to `i - 1` in the input string `s`. The prefix sum array greatly reduces the complexity of determining the count of each digit in a substring from `O(len(substring))` to `O(1)` time complexity since it can be computed by a simple subtraction.

2. **Iterating Through Substrings**: To generate all possible substrings of `s`, two nested loops are used. For all pairs of indices `(i, j)` such that `i <= j`, these indices represent the start and end of the substring `s[i:j+1]`.

3. **Checking the Equality of Digit Frequencies**: The key part of the solution is the `check` function, which takes two indices `(i, j)` and verifies if the digits in `s[i:j+1]` appear the same number of times. This is done by comparing the prefix sums for index `j + 1` and `i`. If, for every digit, the difference in prefix sums is either `0` (digit not present) or equal for all present digits, the substring meets our criteria.

4. **Set for Unique Substrings**: To ensure no substring is counted more than once, we use a set `vis`. A substring is added to the set only if it passes the check function, which implicitly handles the uniqueness constraint.

5. **Returning the Count**: The total number of unique valid substrings is the size of the set `vis`, which is returned as the output.

This algorithm efficiently checks all possible substrings and ensures unique counts. The usage of prefix sum arrays significantly reduces the time complexity for checking digit frequency equality in substrings. The overall complexity of the algorithm is `O(n^3)` due to the three nested loops - two for generating substrings and one inside the `check` function for iterating over the 10 possible digits.

Example Walkthrough

Let's illustrate the solution approach with a small example using the digit string `s = "1122"`. We want to find unique substrings where every digit appears the same number of times.

Step 1: Prefix Sum Array

First, we create a prefix sum matrix `presum` with dimensions corresponding to the length of `s` plus one (for simplicity), and with 10 columns for each digit.

The initialized `presum` for our string `s = "1122"` will look like this:

```
   0 1 2 3 4 5 6 7 8 9
0  0 0 0 0 0 0 0 0 0 0
1  0 1 0 0 0 0 0 0 0 0
2  0 2 0 0 0 0 0 0 0 0
3  0 2 1 0 0 0 0 0 0 0
4  0 2 2 0 0 0 0 0 0 0
```

Step 2: Iterating Through Substrings

Next, we iterate through all possible substrings using nested loops. For example:

- `s[0:1]` is '1', count of '1' is 1
- `s[0:2]` is '11', count of '1' is 2
- `s[0:3]` is '112', counts of '1' and '2' are not equal
- `s[1:2]` is '1', count of '1' is 1
- ... and so on

Step 3: Checking the Equality of Digit Frequencies

For each substring, we use the `check` function. Let's consider a check for `s[2:3]`, which is '22':

- We look at `presum[4] - presum[2]` to get `[0, 0, 1, 0, 0, 0, 0, 0, 0, 0]`.
- This shows us that '2' occurs once in this range.
- Since all digits either have a count of 0 or the same count (in this case, 1), this substring is valid.

Step 4: Set for Unique Substrings

If a substring passes the check, we add it to the set `vis`. Continuing the process will give us all valid substrings:

- The substring '11' appeared twice, but because we are using a set, it will only be counted once.
- The set `vis` might look like: `{'1', '2', '11', '22'}`.

Step 5: Returning the Count

The size of the set `vis` gives us the number of unique valid substrings. In this example, the count would be 4.

This walkthrough with the small example of `s = "1122"` illustrates the mechanism of the algorithm and how it ensures that we count only the unique valid substrings where every digit appears the same number of times.

Python Solution

```python
class Solution:
    def equalDigitFrequency(self, s: str) -> int:
        # Helper function to check if the substring from index i to j has equal frequency of digits.
        def has_equal_frequency(i, j):
            digit_frequencies = set()
            # Check the frequency of each digit in the substring.
            for digit in range(10):
                # Calculate the frequency of the current digit.
                frequency = prefix_sum[j + 1][digit] - prefix_sum[i][digit]

                # If the frequency is greater than 0, add it to the set.
                if frequency > 0:
                    digit_frequencies.add(frequency)

                # If we have more than one frequency, they are not all equal.
                if len(digit_frequencies) > 1:
                    return False
            return True

        # Calculate the length of the string.
        length = len(s)
        # Create a prefix sum list to keep track of the frequency of each digit up to that index.
        prefix_sum = [[0] * 10 for _ in range(length + 1)]

        # Populate the prefix sum array.
        for i, char in enumerate(s):
            digit = int(char)
            prefix_sum[i + 1][digit] += 1
            # Add the previous prefix sums for each digit.
            for j in range(10):
                prefix_sum[i + 1][j] += prefix_sum[i][j]

        # Use a set to collect unique substrings with equal digit frequency.
        unique_substr = set()
        # Check all possible substrings.
        for i in range(length):
            for j in range(i, length):
                # If the substring has equal frequency of digits, add it to the set.
                if has_equal_frequency(i, j):
                    unique_substr.add(s[i:j+1])

        # Return the number of unique substrings with equal digit frequency.
        return len(unique_substr)

```

Java Solution

```java
class Solution {
    public int equalDigitFrequency(String s) {
        int length = s.length();
        int[][] prefixSum = new int[length + 1][10];

        // Build prefix sum array for each digit
        for (int i = 0; i < length; ++i) {
            // Increment the count of the current digit in the prefix sum
            prefixSum[i + 1][s.charAt(i) - '0']++;
            // Copy the previous counts to the current prefix
            for (int digit = 0; digit < 10; ++digit) {
                prefixSum[i + 1][digit] += prefixSum[i][digit];
            }
        }

        Set<String> uniqueSubstrings = new HashSet<>();

        // Check all possible substrings for equal frequency condition
        for (int start = 0; start < length; ++start) {
            for (int end = start; end < length; ++end) {
                // If the current substring meets the condition, add it to the set
                if (hasEqualDigitFrequency(start, end, prefixSum)) {
                    uniqueSubstrings.add(s.substring(start, end + 1));
                }
            }
        }

        // Return the number of unique substrings with equal digit frequency
        return uniqueSubstrings.size();
    }

    private boolean hasEqualDigitFrequency(int start, int end, int[][] prefixSum) {
        Set<Integer> frequencies = new HashSet<>();
        // Check the frequency of each digit in the substring
        for (int digit = 0; digit < 10; ++digit) {
            int count = prefixSum[end + 1][digit] - prefixSum[start][digit];
            if (count > 0) {
                // Add non-zero frequency to the set
                frequencies.add(count);
            }
            // If there are more than one unique frequencies, return false
            if (frequencies.size() > 1) {
                return false;
            }
        }
        // If there is only one frequency for all digits, return true
        return true;
    }
}

```

C++ Solution

```cpp
#include <string>
#include <vector>
#include <unordered_set>
using namespace std;

class Solution {
public:
    // Function to calculate the number of unique substrings with an equal digit frequency
    int equalDigitFrequency(string s) {
        int length = s.length();
        vector<vector<int>> prefixSum(length + 1, vector<int>(10, 0));

        // Build prefix sum array for each digit
        for (int i = 0; i < length; ++i) {
            // Increment the count of the current digit in the prefix sum
            prefixSum[i + 1][s[i] - '0']++;
            // Copy the previous counts to the current prefix
            for (int digit = 0; digit < 10; ++digit) {
                prefixSum[i + 1][digit] += prefixSum[i][digit];
            }
        }

        unordered_set<string> uniqueSubstrings;

        // Check all possible substrings for equal frequency condition
        for (int start = 0; start < length; ++start) {
            for (int end = start; end < length; ++end) {
                // If the current substring meets the condition, add it to the set
                if (hasEqualDigitFrequency(start, end, prefixSum)) {
                    uniqueSubstrings.insert(s.substr(start, end - start + 1));
                }
            }
        }

        // Return the number of unique substrings with equal digit frequency
        return uniqueSubstrings.size();
    }

private:
    // Helper function to check if a substring has equal digit frequency
    bool hasEqualDigitFrequency(int start, int end, const vector<vector<int>>& prefixSum) {
        unordered_set<int> frequencies;
        // Check the frequency of each digit in the substring
        for (int digit = 0; digit < 10; ++digit) {
            int count = prefixSum[end + 1][digit] - prefixSum[start][digit];
            if (count > 0) {
                // Add non-zero frequency to the set
                frequencies.insert(count);
            }
            // If there are more than one unique frequencies, return false
            if (frequencies.size() > 1) {
                return false;
            }
        }
        // If there is only one frequency for all digits, return true
        return frequencies.size() == 1;
    }
};

```

Typescript Solution

```typescript
// Helper function to check if the substring has equal digit frequency
function hasEqualDigitFrequency(start: number, end: number, prefixSum: number[][]): boolean {
    const frequencies: Set<number> = new Set();

    // Check the frequency of each digit in the substring
    for (let digit = 0; digit < 10; ++digit) {
        const count: number = prefixSum[end + 1][digit] - prefixSum[start][digit];
        if (count > 0) {
            // Add non-zero frequency to the set
            frequencies.add(count);
        }
        // If there are more than one unique frequencies, return false
        if (frequencies.size > 1) {
            return false;
        }
    }
    // If there is only one frequency for all digits, return true
    return true;
}

// Function to count unique substrings with equal digit frequencies
function equalDigitFrequency(s: string): number {
    const length: number = s.length;
    const prefixSum: number[][] = new Array(length + 1).fill(null).map(() => new Array(10).fill(0));
    const uniqueSubstrings: Set<string> = new Set();

    // Build prefix sum array for each digit
    for (let i = 0; i < length; ++i) {
        // Increment the count of the current digit in the prefix sum
        prefixSum[i + 1][s.charAt(i) - '0']++;
        // Copy the previous counts to the current prefix
        for (let digit = 0; digit < 10; ++digit) {
            prefixSum[i + 1][digit] += prefixSum[i][digit];
        }
    }

    // Check all possible substrings for equal frequency condition
    for (let start = 0; start < length; ++start) {
        for (let end = start; end < length; ++end) {
            // If the current substring meets the condition, add it to the set
            if (hasEqualDigitFrequency(start, end, prefixSum)) {
                uniqueSubstrings.add(s.substring(start, end + 1));
            }
        }
    }

    // Return the number of unique substrings with equal digit frequency
    return uniqueSubstrings.size;
}

```

Time and Space Complexity

Time Complexity

The code includes nested loops where `i` ranges from `0` to `n-1` and `j` ranges from `i` to `n-1`, leading to a `O(n^2)` complexity for this part. Within the innermost loop, there is a function `check()` that iterates once again through a constant `10` elements representing the digits `0` through `9`, which do not depend on `n` and hence contribute a `O(1)` to the inner complexity. Combining the nested loops with the constant-time `check()` function, we get a total time complexity of `O(n^2)`.

Before the loops, there is also another loop for constructing the `presum` array which takes `O(n)` time since it iterates over the length of the input string `s` and does constant work in updating the counts of digits.

Therefore, combining all these, the overall time complexity is `O(n^2)`.

Space Complexity

The space complexity is determined by the storage requirements of the `presum` array and the `vis` set.

1. `presum` an array of `n+1` elements where each element is an array of `10` integers, contributes a space complexity of `O(10n)` or simplifying `O(n)` because `10` is a constant.

2. `vis` is a set that holds unique substrings formed by the combinations of indices `i` and `j`. In the worst-case scenario, each pair (i, j) could potentially be unique, leading to `O(n^2)` space complexity.

By adding both `O(n)` for `presum` and `O(n^2)` for `vis`, the dominant term is `O(n^2)`, making the overall space complexity `O(n^2)`.