2955. Number of Same-End Substrings

Counting

String

Problem Description

<u>Array</u>

Hash Table

Medium

substring of s that starts at index 1 and ends at index r (both inclusive). The objective is to count how many substrings within this defined substring have the same character at the beginning and the end. These are called "same-end" substrings. It's important to include all substrings, even if they are just a single character, as a character is considered a "same-end" substring of itself. The final result should be an array where each element corresponds to the count for each query.

The task is to handle a series of queries on a given string s. For each query, you are given two indices 1 and r, which define a

Prefix Sum

Intuition To efficiently address multiple queries without re-calculating substring counts for all characters each time, the solution uses a Prefix Sum approach. A Prefix Sum is an array that helps to quickly calculate the sum of elements in a given range of an array. Here, it's adapted to count characters by building a prefix sum array for each unique character in string s. This array records how

many times each character has appeared up to every position in the string. Thus, with this pre-computed information, for any given range [I, r], you can quickly find out how often a character appears within that range. To get to the number of "same-end" substrings for a given range, we sum up the counts for each character. For any character that appears more than once in the range, we calculate the number of ways to pick any two instances of that character, as they will form a "same-end" substring. This count is given by the formula for combinations: for x instances of a character, there are x *

(x - 1) / 2 ways to pick two. This is combined with the count of single-character substrings (which is always a "same-end"

substring), which is just the length of the range. The sum of these calculations for each character gives the total count of "sameend" substrings for the range. **Solution Approach**

Character Set Creation: First, we create a set cs of all unique characters found in the string s. This set is used to identify

which characters we need to build prefix sums for. Prefix Sum Calculation: Next, we create a dictionary cnt where each key is a character from our set cs, and the value is an

array with n + 1 elements (where n is the length of the string s). The extra element at the beginning of the array is used to

The implementation of the solution is based on the following key steps:

handle cases where the substring starts at index 0. This array is used to store prefix sums, such that cnt[c][i] represents the count of occurrences of character c up to the i-th (0-indexed) position in the string. We populate this array with the

actual prefix sums by iterating through the string and updating the counts for each character at each position.

initialize the count t with the length of the query range (i.e., r - l + 1), as every single character is considered a "same-end" substring. Counting Same-End Substrings: We then enumerate each character c from our set cs and use the prefix sums to calculate the number of times c appears in the interval [1, r]. This number is calculated using the formula x = cnt[c][r + 1] cnt[c][1]. As mentioned previously, we calculate the total possible "same-end" substrings that can be formed using this

Evaluating Queries: With the <u>prefix sum</u> arrays ready, we can now efficiently evaluate each query. For every query [1, r], we

- character by the formula x * (x 1) / 2. We add this to our ongoing count t. Query Results Collection: After all characters have been evaluated for a given query, the final count t represents the total number of "same-end" substrings for this particular substring of s. We append this count to our answer list ans.
- without recalculating the entire substring each time, which significantly optimizes the performance for a large number of queries. In terms of the algorithmic pattern, this implementation uses a combination of prefix sums for precomputation and enumeration over a set to perform the actual calculation. This pattern reduces the time complexity from potentially O(n^2) per query to O(n +

k) preprocessing and O(u) per query, where n is the length of the string, k is the total number of queries, and u is the number of

The full enumeration for each character and swift calculations using prefix sums allow us to efficiently process multiple queries

Let's say we have a string s = "abaac" and we need to handle the following queries: [[0, 3], [1, 4], [2, 2]]. Character Set Creation: From the string s, we identify the set of unique characters: $cs = \{'a', 'b', 'c'\}$. Prefix Sum Calculation: We create prefix sum arrays for each character in cs: For a: cnt['a'] = [0, 1, 2, 2, 3] (There's one a by index 0, two by index 1, and so on.)

Query [0, 3]: We initialize count t with 3 - 0 + 1 = 4, since there are four single characters within this range.

Evaluating Queries:

unique characters.

Example Walkthrough

Counting Same-End Substrings: For each unique character:

- from the single character itself.
- Query [1, 4]: Initialize t with 4 1 + 1 = 4. For a: x = cnt['a'][4 + 1] - cnt['a'][1] = 3 - 2 = 1.

from the single character itself.

Solution Implementation

from typing import List

n = len(s)

Get the length of the string.

for i, char in enumerate(s, 1):

total_count = length

prefix_count[char][i] += 1

for unique_char in unique_chars:

class Solution:

Python

Query [2, 2]: Initialize t with 2 - 2 + 1 = 1.

For b: x = cnt['b'][4 + 1] - cnt['b'][1] = 1 - 1 = 0. For c: x = cnt['c'][4 + 1] - cnt['c'][1] = 1 - 0 = 1. There's one 'c' so no same-end substrings can be formed apart

For a: x = cnt['a'][3 + 1] - cnt['a'][0] = 2 - 1 = 1. There's only one 'a' so no same-end substrings can be formed apart

Since this is a single-character query, it only contains the character itself as a "same-end" substring.

without calculating from scratch each time thanks to the prefix sum arrays.

def sameEndSubstringCount(self, s: str, queries: List[List[int]]) -> List[int]:

Create a set with all unique characters in the string s.

Populate the prefix count for characters in the string.

Initialize a list to hold the answer for each query.

Total t = 1. There is 1 "same-end" substring for this query.

Total t = 4 + 0 + 0 = 4. There are 4 "same-end" substrings for this query.

For b: cnt['b'] = [0, 0, 1, 1, 1, 1] (The first b appears by index 1.)

For c: cnt['c'] = [0, 0, 0, 0, 1, 1] (The first c appears by index 4.)

For b: x = cnt['b'][3 + 1] - cnt['b'][0] = 1 - 0 = 1. Only one 'b' is present.

For c: x = cnt['c'][3 + 1] - cnt['c'][0] = 0 - 0 = 0. No 'c' is present in this range.

Adding them up gives t = 4 + 0 + 0 + 0 = 4. There are 4 "same-end" substrings for this query.

Query Results Collection: We collect the results for each query in our answer list: ans = [4, 4, 1]. The final result of the queries is [4, 4, 1], which efficiently gives us the count of same-end substrings in each query range

unique_chars = set(s) # Initialize a dictionary to store prefix counts for each character. prefix_count = {char: [0] * (n + 1) for char in unique_chars}

Initialize the total count of end-matching substrings with the length of substring.

prefix_count[unique_char][i] = prefix_count[unique_char][i - 1]

Process each query in the queries list. for left, right in queries: # Calculate the length of the substring. length = right - left + 1

#include <vector>

#include <string>

class Solution {

public:

#include <cstring>

using namespace std;

int strLength = s.size();

int charCount[26][strLength + 1];

for (auto& query : queries) {

vector<int> sameEndSubstringCount(string s, vector<vector<int>>& queries) {

// Populate the character count array, storing the cumulative

charCount[i][index] = charCount[i][index - 1];

// count of each character up to current index

charCount[s[index - 1] - 'a'][index]++;

vector<int> results; // Initializing the result vector

prefix_count = $\{char: [0] * (n + 1) for char in unique_chars\}$

prefix_count[unique_char][i] = prefix_count[unique_char][i - 1]

Initialize the total count of end-matching substrings with the length of substring.

Count the occurrences of each character within the current query's substring.

Calculate the frequency of the current character in the substring.

Include the count of substrings with the same ending character.

total_count += char_count * (char_count - 1) // 2

Combining both parts results in $O((n + m) * |\Sigma|)$ total time complexity.

char_count = prefix_count[char][right + 1] - prefix_count[char][left]

Populate the prefix count for characters in the string.

Initialize a list to hold the answer for each query.

for i, char in enumerate(s, 1):

for left, right in queries:

total_count = length

answers = []

prefix_count[char][i] += 1

length = right - left + 1

for char in unique_chars:

for unique_char in unique_chars:

Process each query in the queries list.

Calculate the length of the substring.

// Iterate through each query to calculate results

int left = query[0], right = query[1];

for (int i = 0; i < 26; ++i) {

for (int index = 1; index <= strLength; ++index) {</pre>

memset(charCount, 0, sizeof(charCount)); // Initialize all elements to 0

// Directly adding the length of the substring as part of the result

answers = []

```
# Count the occurrences of each character within the current query's substring.
            for char in unique_chars:
                # Calculate the frequency of the current character in the substring.
                char_count = prefix_count[char][right + 1] - prefix_count[char][left]
                # Include the count of substrings with the same ending character.
                total_count += char_count * (char_count - 1) // 2
           # Add the total count of end-matching substrings to the answers list.
            answers.append(total_count)
       # Return the list of answers for each query.
        return answers
Java
class Solution {
    public int[] sameEndSubstringCount(String s, int[][] queries) {
       int strLength = s.length();
       // Initialize the count array to keep track of character frequencies up to each index
       int[][] charCount = new int[26][strLength + 1];
       // Precompute the number of each character up to each index j
        for (int j = 1; j <= strLength; ++j) {</pre>
           // Copy counts from the previous index
            for (int i = 0; i < 26; ++i) {
                charCount[i][j] = charCount[i][j - 1];
            // Update the count of the current character
            charCount[s.charAt(j - 1) - 'a'][j]++;
        int numQueries = queries.length;
        int[] answer = new int[numQueries];
        // Process each query in queries array
        for (int k = 0; k < numQueries; ++k) {</pre>
           int left = queries[k][0], right = queries[k][1];
           // Initial count is the length of the substring
           answer[k] = right - left + 1;
           // Count the number of substrings that start and end with the same character
            for (int i = 0; i < 26; ++i) {
                int countInSubstring = charCount[i][right + 1] - charCount[i][left];
                // Add the combination of two same characters to the answer
                answer[k] += countInSubstring * (countInSubstring - 1) / 2;
       return answer;
C++
```

```
results.push_back(right - left + 1);
              // Iterate through each character in the alphabet
              for (int i = 0; i < 26; ++i) {
                  // Compute the count of the current character within the given range
                  int charInRange = charCount[i][right + 1] - charCount[i][left];
                  // Use the count to calculate combinations and add to current result
                  results.back() += charInRange * (charInRange - 1) / 2;
          return results; // Return the final computed result vector
  };
  TypeScript
  function sameEndSubstringCount(s: string, queries: number[][]): number[] {
    const lengthOfString: number = s.length;
    // Initialize a 2D array to store counts of characters up to each index
    const characterCount: number[][] = Array.from({ length: 26 }, () => Array(lengthOfString + 1).fill(0));
    // Fill the character count array with the cumulative counts of each character
    for (let endIndex = 1; endIndex <= lengthOfString; endIndex++) {</pre>
      for (let i = 0; i < 26; i++) {
        characterCount[i][endIndex] = characterCount[i][endIndex - 1];
      characterCount[s.charCodeAt(endIndex - 1) - 'a'.charCodeAt(0)][endIndex]++;
    // Initialize an array to store the answers to the queries
    const answer: number[] = [];
    // Process each query to count substrings ending with the same letter
    for (const [leftIndex, rightIndex] of queries) {
      // Start with the length of the substring as the base count
      answer.push(rightIndex - leftIndex + 1);
      // Count the pairs of identical characters within the query range
      for (let i = 0; i < 26; i++) {
        const countInRange: number = characterCount[i][rightIndex + 1] - characterCount[i][leftIndex];
        answer[answer.length - 1] += (countInRange * (countInRange - 1)) / 2;
    return answer;
from typing import List
class Solution:
   def sameEndSubstringCount(self, s: str, queries: List[List[int]]) -> List[int]:
       # Get the length of the string.
       n = len(s)
       # Create a set with all unique characters in the string s.
       unique_chars = set(s)
       # Initialize a dictionary to store prefix counts for each character.
```

Add the total count of end-matching substrings to the answers list. answers.append(total_count) # Return the list of answers for each query. return answers

Time and Space Complexity

The time complexity of the given code is $O((n + m) * |\Sigma|)$, where n is the length of the string s and m is the number of queries, while $|\Sigma|$ represents the size of the set of letters appearing in the string s.

Time Complexity

character set).

This complexity arises because for each of the n characters in the string s, we update a counter for each character in the character set Σ . This process has a complexity of $O(n * |\Sigma|)$. Additionally, for each query, we calculate a number of substrings based on character counts, which involves iterating over the set of characters again, leading to a complexity of $0(m * |\Sigma|)$.

Space Complexity The space complexity of the code is $O(n * |\Sigma|)$ because we store a count for each character of Σ at each index of the string,

leading to a two-dimensional data structure with dimensions n (length of the string) by | [X] (number of unique characters in the