2950. Number of Divisible Substrings

String

Counting

Prefix Sum

since 6 (the sum of its mapped values) is divisible by 3 (the length of the substring).

Problem Description

Hash Table

Medium

1 to 9. To determine if a substring of s is "divisible", we sum the mapped values of all its characters, and if this sum is divisible by the length of the substring, then the substring is considered divisible. Our task is to find the total count of these divisible substrings in the string s. Remember, substrings are contiguous sequences of

For example, if | s = "abc", and the mapping is { 'a': 1, 'b': 2, 'c': 3 }, then "abc" (summing to 6) is a divisible substring

In this problem, we have a string s composed of lowercase English letters. Each letter is mapped to a digit according to a given

keyboard layout that resembles an old phone keypad. Under this mapping, certain groups of letters correspond to the digits from

characters, and they must be non-empty.

Intuition

the sum of the numeric values of its characters according to the given mapping, and checking if this sum is divisible by the substring's length. Here's a step-by-step breakdown of the intuition behind this approach:

To solve this problem, we can use a brute force approach, which involves enumerating over every possible substring, calculating

Create a mapping from characters to digits based on the provided image or keyboard layout. This will be our reference to

Iterate over all possible starting indices of substrings in the string s. For each starting index, iterate over all possible ending indices that follow it to generate every possible substring starting

convert characters to their corresponding numerical values.

- from that index. For each substring, calculate the sum of its character's numerical values.
- Check if the sum is divisible by the length of the substring. If it is, we've found a divisible substring, and we increment our counter.
- This enumeration algorithm ensures we do not miss any possible substrings while systematically checking each one. While this is
- not the most efficient method, given the problem's constraints, it is an approach that is easy to understand and implement.
- The implementation of the solution is straightforward once the brute force approach has been understood. The solution uses a couple of basic concepts in Python: loops and hash tables (dictionaries). Hash Table for Character to Number Mapping: We use a dictionary mp to map each character to its corresponding number.

The mapping is taken from an image that displays characters on an old phone keypad layout. Nested Loops to Enumerate Substrings: Two nested loops are utilized to consider every possible substring of the given

Sum of Mapped Characters: Inside the inner loop, as we extend our substring one character at a time, we keep adding the

Checking for Divisibility: After including each character, we check if the current sum s is divisible by the length of the

string s. The outer loop sets the starting index of the substring, while the inner loop sets the ending index.

current substring, which is j - i + 1 (as i is the starting index and j is the ending index).

mapped numerical value of the recently included character to a running sum s.

then returned.

 $mp = \{\}$

ans = 0

return ans

Map each character to a digit

Count of divisible substrings

"a" is a divisible substring.

ans += s % (j - i + 1) == 0

Return the total count of divisible substrings

Solution Approach

Incrementing the Count: Each time we find that the sum is divisible by the substring's length, we increment ans by 1. Returning the Result: After all substrings have been considered, ans contains the total count of divisible substrings, which is

- The algorithm's time complexity is O(n^2) in the worst case, where n is the length of string s. This is due to the fact that we are checking every substring possible. The space complexity is O(1) or O(n) depending on whether we consider the space taken by
- for i, s in enumerate(d, 1): for c in s: mp[c] = i

n = len(word)# Enumerating all substrings for i in range(n): s = 0for j in range(i, n): s += mp[word[j]]

We repeat this process, incrementing i and j to explore all substrings.

Check if sum is divisible by the length of the substring

the hash table which has a fixed size (there are 26 lowercase English letters).

Here is the key portion of the solution code, which represents the described logic:

```
for-loops for checking each substring.
Example Walkthrough
  Let's go through a small example using the string s = "abcde", with the mapping { 'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5
  . According to this mapping, the numeric values for each character are their respective positions in the English alphabet. We aim
  to enumerate all substrings of s and determine which ones are divisible.
  Let's visualize the process of iterating over substrings and checking their divisibility:
     We start with the outer loop at i = 0 (character 'a').
      The inner loop starts at j = i.
     For the first iteration, i = 0 and j = 0, the substring is "a" with a sum of 1. Since 1 % 1 (length of the substring) is 0,
```

The implementation exploits the elegance of Python's dictionaries for fast look-ups of character mappings and the efficiency of

Next, i = 0 and j = 1, the substring is "ab". The sum is 1 + 2 = 3. Since 3 % 2 is not 0, "ab" is not divisible. Continuing this, i = 0 and j = 2, the substring is "abc" with sum 1 + 2 + 3 = 6. Since 6 % 3 is 0, "abc" is divisible.

• i = 4, "e" (5) From this, we can count that the divisible substrings are: "a", "abc", "b", "c", "cde", "d", "e". There are a total of 7.

Now for a visualization of the nested loops in action:

• i = 0, "a" (1), "ab" (3), "abc" (6), "abcd" (10), "abcde" (15)

• i = 1, "b" (2), "bc" (5), "bcd" (9), "bcde" (14)

• i = 2, "c" (3), "cd" (7), "cde" (12)

• i = 3, "d" (4), "de" (9)

1. Map characters to numbers.

2. Initialize the count (ans) to 0.

Solution Implementation

Python

Java

class Solution {

class Solution:

3. Use nested loops to extract all substrings.

for char in substr:

for i in range(word length):

int[] mapping = new int[26];

int length = word.length();

for (int i = 0; i < length; ++i) {</pre>

for i in range(i, word length):

Return the total count of divisible substrings

// Array of strings representing groups of characters

// Mapping for characters to their respective group values

// Iterate over all possible starting points of substrings

// Initialize the mapping for each character to its group value

public int countDivisibleSubstrings(String word) {

// Initialize count of divisible substrings

for (int j = i; j < length; ++j) {</pre>

sum += mappings[word[j] - 'a'];

// If it is, increment the count

// Return the total count of divisible substrings

function countDivisibleSubstrings(word: string): number {

const letterWeights: number[] = Array(26).fill(0);

for (const char of groups[groupIndex]) {

// Initialize the count of valid substrings

return count;

// by the length of the substring

// Get the length of the word

let count = 0;

const wordLength = word.length;

};

TypeScript

count += sum % (j - i + 1) == 0 ? 1 : 0;

word_length = len(word)

return total_count

- In our Python implementation, we would follow these steps:
- 5. Check if this sum is divisible by the length of this substring; if so, increment the count (ans). 6. After considering all substrings, ans will represent the total count of divisible substrings.

sum of numbers = 0 # Sum of numbers corresponding to characters in the current substring

Increase the count if the sum is divisible by the length of the substring

String[] groups = {"ab", "cde", "fgh", "iik", "lmn", "opg", "rst", "uvw", "xyz"};

// 'sum' will hold the sum of the group values for the current substring

// Increment the sum with the mapping of the current character

// Check if the sum is divisible by the length of the substring

// Function to count the number of substrings where the sum of the mapped values of letters is divisible

// Define an array that represents the groups of letters, each group has the same weight

const groups: string[] = ['ab', 'cde', 'fgh', 'ijk', 'lmn', 'opq', 'rst', 'uvw', 'xyz'];

letterWeights[char.charCodeAt(0) - 'a'.charCodeAt(0)] = groupIndex + 1;

// Populate the letterWeights with the corresponding group index + 1 (as weights)

for (let groupIndex = 0; groupIndex < groups.length; ++groupIndex) {</pre>

// Check all substrings starting from each character in the word

// Initialize a mapping array with 26 elements set to 0 to store the weight of each letter

sum of numbers += char to number[word[j]] # Add the number corresponding to current character

Define substrings where each character maps to a number (1-9)substrings = ["ab", "cde", "fqh", "iik", "lmn", "opq", "rst", "uvw", "xvz"] char to number = {} # Dictionary to map characters to their respective numbers for index. substr in enumerate(substrings, 1):

Iterate over the word to check each possible substring

char_to_number[char] = index # Populate the mapping

total_count += sum_of_numbers % (j - i + 1) == 0

total count = 0 # Variable to keep track of divisible substrings count

def countDivisibleSubstrings(self, word: str) -> int:

4. Calculate the sum of the numerical values of characters in the current substring.

for (int i = 0; $i < groups.length; ++i) {$ for (char c : groups[i].toCharArray()) { mapping[c - 'a'] = i + 1;

int count = 0;

int sum = 0;

```
// Iterate over all possible ending points of substrings
            for (int i = i; i < length; ++i) {</pre>
                // Add group value of the current character to 'sum'
                sum += mapping[word.charAt(i) - 'a'];
                // Increment the count if sum is divisible by the length of the substring
                count += sum % (j - i + 1) == 0 ? 1 : 0;
        // Return the total count of divisible substrings
        return count;
C++
class Solution {
public:
    // Function to count divisible substrings
    int countDivisibleSubstrings(string word) {
        // An array to hold strings that represent the respective remainders
        string divisors[9] = {"ab", "cde", "fqh", "iik", "lmn", "opq", "rst", "uvw", "xyz"};
        int mappings[26] = {}; // Array to map characters to their divisor group number
        // Populate the mappings array with corresponding group numbers
        for (int i = 0; i < 9; ++i) {
            for (char& c : divisors[i]) {
                mappings[c - 'a'] = i + 1;
        int count = 0: // Variable to keep the count of divisible substrings
        int length = word.size(); // The length of the input word
        // Iterate over all starting positions of the substrings in the word
        for (int i = 0; i < length; ++i) {</pre>
            int sum = 0; // To keep the sum of divisor group numbers
            // Iterate over all possible ending positions of the substring
```

```
for (let startIndex = 0: startIndex < wordLength: ++startIndex) {</pre>
       // Initialize the sum of letter weights for the current substring
        let sum = 0;
       // Iterate over the substring from the current startIndex
        for (let endIndex = startIndex; endIndex < wordLength; ++endIndex) {</pre>
            // Add the corresponding letter weight to the sum
            sum += letterWeights[word.charCodeAt(endIndex) - 'a'.charCodeAt(0)];
           // If the sum is divisible by the substring length, increase the count
           if (sum % (endIndex - startIndex + 1) === 0) {
                count++;
   // Return the final count of valid substrings
   return count;
class Solution:
   def countDivisibleSubstrings(self. word: str) -> int:
       \# Define substrings where each character maps to a number (1-9)
       substrings = ["ab", "cde", "fqh", "ijk", "lmn", "opq", "rst", "uvw", "xyz"]
       char to number = {} # Dictionary to map characters to their respective numbers
        for index, substr in enumerate(substrings, 1):
            for char in substr:
                char_to_number[char] = index # Populate the mapping
       total count = 0 # Variable to keep track of divisible substrings count
       word_length = len(word)
       # Iterate over the word to check each possible substring
       for i in range(word length):
           sum of numbers = 0 # Sum of numbers corresponding to characters in the current substring
           for j in range(i, word length):
               sum of numbers += char to number[word[i]] # Add the number corresponding to current character
               # Increase the count if the sum is divisible by the length of the substring
               total_count += sum_of_numbers % (j - i + 1) == 0
       # Return the total count of divisible substrings
       return total_count
```

The time complexity of the given code is $0(n^2)$. This is because there are two nested loops. The outer loop runs for n iterations (n being the length of the word), and for each iteration of the outer loop, the inner loop runs for at most n iterations - starting

Time Complexity

Time and Space Complexity

not change with the size of the input string and is hence constant.

from the current index of the outer loop to the end of the word. During each iteration of the inner loop, a constant number of operations are executed. So, for each element, we potentially loop through every other element to the right of it, leading to the n * (n-1) / 2 term, which simplifies to $0(n^2)$. **Space Complexity** The space complexity of the code is O(C) where C is the size of the character set. In this case, C=26 as there are 26 lowercase English letters. The space is used to store the mapping of each character to its associated integer, which in this instance does