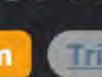
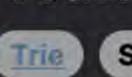
1698. Number of Distinct Substrings in a String



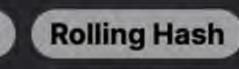












Leetcode Link

Problem Description

The problem requires calculating the total number of distinct substrings that can be formed from a given string s. A substring is defined as any sequence of characters that can be derived from the string by deleting zero or more characters from the beginning and zero or more characters from the end. For example, in the string "abc", some of the substrings include "a", "ab", "abc", and "b". The key here is to ensure that each substring is counted only once, even if it occurs multiple times in different positions of the string.

Intuition

The intuitive approach to solving this problem relies on generating all possible substrings of the given string and then counting the unique ones. To achieve this, two nested loops can be used where the outer loop iterates over the start position of a substring and the inner loop iterates over the end position. For each pair of start and end positions, a substring is extracted and stored in a set, which inherently eliminates duplicates due to its unordered collection of unique elements. After all possible substrings are added to the set, the number of distinct substrings can be determined by simply taking the size of the set.

This approach is easy to implement and understand, and although not the most efficient in terms of time complexity, it works well for strings of reasonable length.

Solution Approach

The solution's implementation relies on the concept of hash sets and comprehensions in Python. Here's a step-by-step explanation of the code provided:

2. A set is created using a set comprehension (1) that iterates through all possible start (1) and end (1) positions of the substrings.

1. Initialize the length of the string n = len(s). This is required to control the loops' range.

- In Python, sets store only unique elements which means any duplicate substrings generated will not be added to the set. 3. The outer loop begins at the first character (i starts at 0) and continues until the end of the string (i < n).
- 4. The inner loop begins at the character immediately following the start position (j starts at 1 + 1) and continues to the end of the string (j 🖛 n). This ensures that we consider all possible end positions for substrings starting at position 1.
- 5. During each iteration of the inner loop, a new substring s[i:j] is sliced from the original string s, with i inclusive and j exclusive. 6. The sliced substring is then added to the set. If the substring already exists in the set, it's ignored, as sets do not store
- duplicates. 7. After the looping, the set will contain all possible unique substrings of the original string.
- 8. The total number of distinct substrings is the size of the set, which is obtained using the len() function on the set.
- To summarize, the algorithm leverages a brute-force approach to generate all substrings and a data structure (set) that naturally

handles the uniqueness constraint. It's excellent for its simplicity and ease of understanding. However, for large strings, this approach may lead to time-consuming operations because the number of substrings grows quadratically with the length of the input string.

Let us consider a simple example using the string "abc" to illustrate the solution approach.

Example Walkthrough

1. First, we initialize the length of the string n = len(s). In our case, n would be 3, since "abc" is three characters long.

- 2. We then create a set to store our substrings. This set will only keep unique elements.
- 3. We start our outer loop, where i is our starting point of the substring. For our string "abc", i will take values 0, 1, and 2.
- 4. For each value of 1, the inner loop starts from 1 + 1 and continues until the end of the string. This is to ensure we cover all possible substrings starting at index i.
- 5. As the loops execute, we generate substrings using s[i:j] and add them to our set. For i = 0, j will go from 1 to 3, generating "a", "ab", and "abc".
- 6. We then move to i = 1, where j will go from 2 to 3, giving us substrings "b" and "bc".
- 8. All these substrings are added to our set: {"a", "ab", "abc", "b", "bc", "c"}.
- 9. After both loops have completed, we end up with a set containing all unique substrings.

7. Lastly, for i = 2, j will only be 3, which gives us the substring "c".

10. The total number of distinct substrings is simply the size of this set. Here, we have 6 unique substrings. Thus, the answer is 6.

// Function to count the number of distinct substrings in a given string

std::unordered_set<std::string_view> uniqueSubstrings;

// Iterating over all possible starting indexes for substrings in s

// Variable to represent the current substring

Through this example, we can see how the proposed solution methodically enumerates all the substrings while enforcing uniqueness

by utilizing the properties of a set.

1 class Solution: def countDistinct(self, text: str) -> int: # Calculate the length of the input string

Python Solution

```
length_of_text = len(text)
           # Use a set comprehension to generate all unique substrings
           # The set will automatically eliminate duplicate substrings
           unique_substrings = {
               text[start:end] for start in range(length_of_text)
9
10
               for end in range(start + 1, length_of_text + 1)
11
12
13
           # The length of the set of unique substrings gives us the count of distinct substrings
           return len(unique_substrings)
14
15
Java Solution
```

// Create a HashSet to store the unique substrings Set<String> uniqueSubstrings = new HashSet<>(); int lengthOfString = s.length(); // Get the length of the input string

class Solution {

class Solution {

public:

9

10

11

12

13

9

10

11

12

public int countDistinct(String s) {

int countDistinct(std::string s) {

// Size of the input string

int length = s.size();

```
// Iterate over all possible starting points for the substrings
           for (int start = 0; start < lengthOfString; ++start) {</pre>
               // Iterate over all possible ending points for the substrings
                for (int end = start + 1; end <= lengthOfString; ++end) {</pre>
10
                    // Extract the substring from start to end index (exclusive) and add it to the Set
11
                    uniqueSubstrings.add(s.substring(start, end));
12
13
14
15
16
           // Return the size of the Set, which represents the count of distinct substrings
17
           return uniqueSubstrings.size();
18
20
C++ Solution
1 #include <string>
   #include <unordered_set>
```

// Using an unordered set to store unique substrings for O(1) average time complexity for insertions and lookups

```
14
           std::string_view currentSubstring;
15
           // String_view of the input string to optimize substring operations
           std::string_view stringView = s;
16
17
18
           // Iterate over all possible starting points for substrings in s
           for (int i = 0; i < length; ++i) {</pre>
19
20
               // Iterate over all possible ending points for substrings, starting from the next character
                for (int j = i + 1; j <= length; ++j) {
21
                   // Create a substring from the current starting and ending points
22
23
                    currentSubstring = stringView.substr(i, j - i);
24
                   // Insert the current substring into the set of unique substrings
25
                   uniqueSubstrings.insert(currentSubstring);
26
27
28
29
           // The size of the set represents the number of distinct substrings in s
           return uniqueSubstrings.size();
30
31
32 };
33
Typescript Solution
   // Importing Set from JavaScript's standard library
   import { Set } from 'core-js';
   // Function to count the number of distinct substrings in a given string
   function countDistinct(s: string): number {
     // Using a Set to store unique substrings for efficient insertions and uniqueness checks
     let uniqueSubstrings: Set<string> = new Set<string>();
 8
```

for (let i = 0; i < length; ++i) { 13 14 // Iterating over all possible ending indexes for substrings, starting from the next character 15 for (let j = i + 1; j <= length; ++j) { // Extracting a substring from the current starting and ending indexes 16

// Length of the input string

let length: number = s.length;

```
let currentSubstring: string = s.substring(i, j);
17
         // Inserting the current substring into the set of unique substrings
         uniqueSubstrings.add(currentSubstring);
20
21
22
23
     // The size of the set represents the number of distinct substrings in s
     return uniqueSubstrings.size;
24
25
26
   export { countDistinct }; // Exporting countDistinct for other modules to use
28
Time and Space Complexity
The given Python code snippet defines a function that counts the distinct substrings of a given string s. To analyze the complexity,
let's consider n to be the length of the input string s.
Time Complexity: 0 (n^3)
The time complexity of the given function is O(n^3) for the following reasons:
```

2. For each iteration, a substring is created using slicing s[i:j]. In Python, slicing a string is O(k) where k is the number of characters in the substring. In the worst case, k can be n leading to 0(n) for the slicing operation.

3. These two combined give us a total of $0(n^2) * 0(n) = 0(n^3)$ complexity since for each pair (i, j), a new substring of the string s can be created with O(n) complexity.

The space complexity of the given function is $O(n^2)$ for the following reasons:

Space Complexity: 0(n^2)

1. Two nested loops iterate over the string to generate all possible substrings, which leads to O(n^2) complexity.

- 1. A set is used to store the distinct substrings. In the worst case, the number of distinct substrings of a string of length n is the sum of the series 1 + 2 + 3 + ... + n, which results in n(n + 1)/2 substrings, equating to an $0(n^2)$ space complexity.
- 2. Even though string slicing itself does not take additional space since it's creating new strings that reference the same characters within the original string, in Python, slicing creates new objects and the total space in the set would be the sum of the length of
- all unique substrings. Considering these points, the complexities are as follows:
 - Time Complexity: 0(n^3)
 - Space Complexity: 0(n^2)