1967. Number of Strings That Appear as Substrings in Word

Easy String

Leetcode Link

Problem Description

The problem gives us an array of strings called `patterns` and a single string called `word`. Our task is to count and return the number of strings from the `patterns` array that are also substrings of the `word`. A substring is defined as a sequence of characters that appear in unbroken succession within another string. For example, "cat" is a substring of "concatenate".

To solve this problem, we must check each pattern in the `patterns` array and determine whether it can be found within the `word`. Every time we find a pattern that is a substring of `word`, we increment our count by one. Once we have checked all the patterns, we return the total count.

Intuition

The solution leverages a simple yet efficient approach:

- Iterate through each string in the `patterns` array.
- Check if the current string pattern is a substring in `word`.
- Count the number of occurrences where a pattern is a substring of `word`.

In Python, this solution is very concise due to the language's concise syntax for string containment (`in` keyword) and list comprehensions (or generator expressions). The expression `p in word` returns `True` if pattern `p` is a substring of `word` and `False` otherwise.

A generator expression is used to iterate through all patterns, yielding a `True` (equivalent to 1) or `False` (equivalent to 0) for each check. The `sum` function is then used to add up these values, resulting in the total count of patterns that are substrings of `word`.

The efficiency of this approach lies in its simplicity—there are no explicit loops or complex logic required; the solution is a straightforward application of built-in Python features to match patterns in a string.

Solution Approach

The solution to this problem is a straightforward application of string manipulation and searching. No complex algorithms or additional data structures are necessary. This is because Python's inherent abilities to handle string operations make it an ideal language for such tasks.

Here's a step-by-step walkthrough of the implementation:

- The solution defines a class `Solution` with a method `numOfStrings` that takes in two arguments: a list of strings `patterns` and a single string `word`.
- The method `numOfStrings` returns the result of a `sum` function which is applied to a generator expression. The generator expression is the key part of this solution:

  ```
  sum(p in word for p in patterns)
  ```

  This line uses the `in` keyword, which in Python, checks for the existence of a substring within another string.
- The generator expression `(p in word for p in patterns)` goes through each pattern `p` in the `patterns` list, checks whether `p` is a substring of `word`, and yields `True` or `False` accordingly.
- Each `True` or `False` is implicitly converted to `1` or `0` as the `sum` function evaluates the expression.
- The `sum` function then adds up these `1`s and `0`s. The result is the total number of times a pattern from `patterns` is found as a substring in `word`.
- The complexity of the solution is O(n * m) where n is the number of patterns and m is the length of the string `word`, assuming the `in` keyword takes O(m) in the worst case (when the pattern is similar to the end part of `word` and has to be checked for each character).

No additional data structures are utilized, and the Python-specific `in` keyword optimizes the string searching, making the code concise and easy to understand.

Example Walkthrough

Let's say we have a list of patterns `["a", "abc", "bc", "d"]` and the word `"abc"`. To determine how many strings from the patterns array are substrings of the word, we proceed as follows:

1. We start with the pattern `"a"`. Is `"a"` a substring of `"abc"`? Yes, it is. The string `"abc"` does contain the substring `"a"`. So we can count that. Now our count is 1.
2. Next, we check the pattern `"abc"`. Is `"abc"` a substring of `"abc"`? Yes, the whole word is a match. We increment our count again. Now the count is 2.
3. Then, we check `"bc"`. Is `"bc"` a substring of `"abc"`? Yes, `"bc"` appears at the end of `"abc"` so it's a match. We update the count again. The count is now 3.
4. Finally, we check `"d"`. Is `"d"` a substring of `"abc"`? No, `"abc"` does not contain the substring `"d"`. The count remains the same.

At the end of this process, we have found that 3 of our patterns are also substrings of the word `"abc"`. Therefore, the method `numOfStrings` would return 3 for this example using the generator expression as described above.

Using the solution approach:

```python
class Solution:
    def numOfStrings(self, patterns, word):
        return sum(p in word for p in patterns)

# Our example patterns and word
patterns = ["a", "abc", "bc", "d"]
word = "abc"

# Create an instance of Solution
solution_instance = Solution()
# Call numOfStrings method and print the result
print(solution_instance.numOfStrings(patterns, word))  # Output: 3
```

This code snippet shows how to use the `Solution` class to solve our example. The output `3` matches our manual count from the walkthrough.

Python Solution

```python
from typing import List

class Solution:
    def num_of_strings(self, patterns: List[str], word: str) -> int:
        # Initialize the count of matches
        count_matches = 0

        # Iterate over each pattern in the list of patterns
        for pattern in patterns:
            # Check if the current pattern is a substring of the word
            if pattern in word:
                # If yes, increment the match count
                count_matches += 1

        # Return the total number of matches found
        return count_matches

# The class method num_of_strings() receives 'patterns', a list of strings, and 'word', a single string.
# It counts how many strings in 'patterns' are substrings of the 'word' parameter.

```

Java Solution

```java
class Solution {
    // Function to count the number of strings in 'patterns' that are substrings of 'word'
    public int numOfStrings(String[] patterns, String word) {
        int count = 0; // Variable to keep track of the number of substrings found

        // Iterate through each pattern in the 'patterns' array
        for (String pattern : patterns) {
            // Check if the current pattern is contained within 'word'
            if (word.contains(pattern)) {
                count++; // Increment the count if the pattern is found
            }
        }

        // Return the total count of patterns found within 'word'
        return count;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <string>

class Solution {
public:
    // Function that counts the number of patterns found within a given word.
    int numOfStrings(vector<string>& patterns, string word) {
        int count = 0; // Initialize the count of found patterns to 0

        // Iterate over each pattern in the patterns vector
        for (auto& pattern : patterns) {
            // Check if the current pattern exists within the word
            if (word.find(pattern) != string::npos) {
                count++; // Increment the count if the pattern is found
            }
        }

        return count; // Return the total count of patterns found in the word
    }
};

```

Typescript Solution

```typescript
// Counts the number of strings in 'patterns' that are substrings of 'word'
function numOfStrings(patterns: string[], word: string): number {
    // Initialize a counter for the number of substrings found
    let count = 0;

    // Iterate through each pattern in the patterns array
    for (const pattern of patterns) {
        // Check if the current pattern is a substring of 'word'
        if (word.includes(pattern)) {
            // Increment the count for each pattern found within 'word'
            count++;
        }
    }

    // Return the total count of substrings found
    return count;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given function primarily depends on two factors: the number of strings in the `patterns` list and the length of the `word` string. For each pattern, the function checks whether that pattern exists within the `word`, which is an O(n) operation where n is the length of the `word`. Assuming the average length of the patterns is `k`, and there are `m` patterns in total, the overall time complexity would be O(m * n).

Therefore, if `m` is the number of patterns and `n` is the length of the `word`, the time complexity is:

```
O(m * n)
```

Space Complexity

The space complexity of this function is O(1) because it uses only a constant amount of additional memory outside of the inputs. The `sum` operation with a generator expression does not create a new list in memory; it simply iterates over the `patterns` and accumulates the count. No additional data structures are used that would scale with the input size.

Hence, the space complexity is:

```
O(1)
```