28. Find the Index of the First Occurrence in a String

Easy Two Pointers String String Matching

Problem Description

The task is to find the first occurrence of the string `needle` within another string `haystack`. If `needle` is found within `haystack`, we should return the starting index of the first occurrence. If `needle` is not found, we return `-1`. It's important to note that an empty `needle` results in `0`, as an empty string is considered to be a substring of any string, including an empty string itself.

Intuition

To solve this problem, the intuitive approach is to scan through the `haystack` string and for each position, check whether the substring starting at that position matches the `needle`. We can do this in a linear scan, considering the length of `needle` is m and the length of `haystack` is n.

We only need to scan until $n - m + 1$ in `haystack`, since if we start matching any later than that, `needle` would overflow the bounds of `haystack`. For each position `i` starting from `0` to $n - m$, we take a substring of `haystack` from `i` to $i + m$ and compare it against `needle`. If it matches, we know we've found the first occurrence, and we return the index `i`. If we reach the end of the scan without finding `needle`, we return `-1`.

The time complexity of this approach is $O((n-m) \times m)$ since in the worst-case scenario, for each starting position, we might compare m characters until we find a mismatch. The space complexity is $O(1)$ as we are not using any extra space proportional to the input size; we are only using a few variables to store the indices and lengths.

Solution Approach

The implementation of the solution is straightforward, following the idea described in the intuition. Here's a step-by-step explanation of the algorithm and its practical realization in the given Python code:

1. First, we obtain the length of both `haystack` and `needle` to manage our loop and comparisons. Let's denote the length of `haystack` as n and the length of `needle` as m.
2. We use a single loop to iterate over the `haystack` string. The end condition for our loop is $n - m + 1$, which ensures that we don't attempt to match `needle` beyond the point where it could possibly fit inside `haystack`.
3. Inside the loop, we take a substring of `haystack` starting from the current index `i` and spanning m characters (the entire length of `needle`). In Python, the substring operation is `haystack[i : i + m]`.
4. We then compare this substring with `needle`. If they are equivalent, it means we have found the first occurrence of `needle` in `haystack`. In this case, we return the current index `i`.
5. If the loop completes without finding a match, it means `needle` is not a part of `haystack`. In this final case, we return `-1` to indicate the absence of `needle` in `haystack`.

In terms of algorithms, this approach uses a simple linear scan with a direct string comparison, making it easy to understand and implement. No additional data structures or complex patterns are used, keeping the space complexity to $O(1)$.

The key part of the code that performs the above logic is:

```
for i in range(n - m + 1):
    if haystack[i : i + m] == needle:
        return i
return -1
```

This code reflects directly the described steps, iterating through `haystack`, extracting substrings, and comparing them with `needle`. It's a simple yet effective solution that leverages Python's built-in ability to handle substrings and comparisons elegantly.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach.

Consider the following strings:

- `haystack = "hello"`
- `needle = "ll"`

Now, following the solution approach steps:

1. Obtain lengths of `haystack` and `needle`. Here, `n = 5` (length of `"hello"`) and `m = 2` (length of `"ll"`).
2. Begin a loop to iterate over the `haystack` from index `0` to `5 - 2 + 1 = 4`.
3. Inside the loop, extract substrings of `haystack` of length `m`. We will have the following comparisons:
   - `i = 0` -> compare `"he"` with `"ll"` -> not a match
   - `i = 1` -> compare `"el"` with `"ll"` -> not a match
   - `i = 2` -> compare `"ll"` with `"ll"` -> this is a match!
4. Since we found the match `"ll"` in `haystack` starting at index `2`, we can stop our search and return this index.
5. If no match was found by the end of the loop, we would return `-1`. However, in this case, we did find the `needle` in the `haystack`, so the loop ceases with a successful outcome, returning `2`.

The loop would have continued to `i = 3` and compared `"lo"` with `"ll"` if a match had not been found at `i = 2`.

Following the solution approach, this process efficiently finds the first occurrence of `needle` in the `haystack`, if it exists, and returns its starting index. If the `needle` is not present, `-1` would be the result.

Solution Implementation

Python

```
class Solution:
    def strStr(self, haystack: str, needle: str) -> int:
        # Length of the haystack and needle strings
        haystack_length, needle_length = len(haystack), len(needle)

        # Check all possible starting positions of needle in haystack
        for start in range(haystack_length - needle_length + 1):
            # If the substring matching the needle's length equals the needle, return the start index
            if haystack[start : start + needle_length] == needle:
                return start

        # If the needle is not found in haystack, return -1
        return -1

# The method strStr is intended to find the first occurrence of the needle string
# in the haystack string and return the index at which it begins.
# If needle is not part of haystack, it returns -1.
```

Java

```
class Solution {
    public int strStr(String haystack, String needle) {
        // If needle is empty, the index to return is 0 (as per the problem statement).
        if (needle.isEmpty()) {
            return 0;
        }

        // Get the lengths of haystack and needle.
        int haystackLength = haystack.length();
        int needleLength = needle.length();

        // Initialize pointers for haystack and needle.
        int haystackPointer = 0;
        int needlePointer = 0;

        // Iterate through the haystack.
        while (haystackPointer < haystackLength) {
            // Check if the current characters in the haystack and needle are the same.
            if (haystack.charAt(haystackPointer) == needle.charAt(needlePointer)) {
                // Special case: if needle length is 1 and characters match, we found the needle.
                if (needleLength == 1) {
                    return haystackPointer;
                }
                // Move both pointers forward.
                haystackPointer++;
                needlePointer++;
            } else {
                // Current characters do not match. Reset haystackPointer back by the amount
                // needlePointer had advanced, then move forward by one to search from next position.
                haystackPointer -= needlePointer - 1;
                // Reset needlePointer back to the start of the needle.
                needlePointer = 0;
            }

            // Check if the needle has been found within the haystack.
            if (needlePointer == needleLength) {
                // The start of the substring in haystack that matches the needle
                // is at the difference between current haystackPointer and the length of the needle.
                return haystackPointer - needlePointer;
            }
        }

        // Needle was not found in the haystack. Return -1 as specified in the problem statement.
        return -1;
    }
}
```

C++

```
class Solution {
private:
    // Constructs the 'next' array used in the KMP algorithm to optimize matching
    vector<int> buildNextArray(string pattern) {
        vector<int> next(pattern.length());
        next[0] = -1; // Initialization with -1 for the first character
        int index = 0, // Index in the pattern string
        int prefixIndex = -1; // Index of the longest prefix that is also a suffix
        int patternLength = pattern.length();
        while (index < patternLength) {
            // When there is a mismatch or it's the first iteration
            while (prefixIndex >= 0 && pattern[index] != pattern[prefixIndex])
                prefixIndex = next[prefixIndex];
            index++, prefixIndex++;
            // If we have not reached the end of the pattern
            if (index < patternLength) {
                // Record the length of the longest prefix which is also suffix
                if (pattern[index] == pattern[prefixIndex])
                    next[index] = next[prefixIndex];
                else
                    next[index] = prefixIndex;
            }
        }
        return next;
    }

public:
    // Function to find the first occurrence of `needle` in `haystack`
    int strStr(string haystack, string needle) {
        if (needle.empty()) // If the needle is empty, return 0 as per convention
            return 0;

        vector<int> nextArray = buildNextArray(needle);

        int haystackLength = haystack.length(); // Length of the haystack string
        int needleLength = needle.length(); // Length of the needle string
        int len = haystackLength - needleLength + 1; // Compute the limit of searching
        for (int i = 0; i < len; ++i) {
            int needleIndex = 0; // Index for the needle string
            int haystackIndex = i; // Starting index in the haystack string
            // Search while the characters match and we are within both strings
            while (needleIndex < needleLength && haystackIndex < haystackLength) {
                if (haystack[haystackIndex] != needle[needleIndex]) {
                    if (nextArray[needleIndex] == 0) {
                        needleIndex = nextArray[needleIndex];
                        continue; // Use the next array to skip some comparisons
                    } else
                        break; // Mismatch without a valid sub-prefix match
                }
                ++haystackIndex, ++needleIndex;
            }
            // When the whole needle string has been traversed, return the starting index
            if (needleIndex == needleLength)
                return haystackIndex - needleIndex;
        }

        return -1; // If the needle has not been found, return -1
    }
};
```

TypeScript

```
/**
 * Finds the first occurrence of the `needle` in `haystack`, and returns its index.
 * If `needle` is not found in `haystack`, returns -1.
 *
 * @param haystack - The string to be searched within.
 * @param needle - The string to find in the haystack.
 * @returns The index at which the needle is found in the haystack, or -1 if not found.
 */
function strStr(haystack: string, needle: string): number {
    // Length of the haystack and needle strings
    const haystackLength: number = haystack.length;
    const needleLength: number = needle.length;

    // Early return if the needle is an empty string
    if (needleLength === 0) return 0;

    // Loop through each character in the haystack until there's no room left for the needle
    for (let i = 0; i <= haystackLength - needleLength; i++) {
        // Assume that the needle is found unless a mismatch is found
        let isMatch: boolean = true;

        // Check each character of the needle against the haystack
        for (let j = 0; j < needleLength; j++) {
            if (haystack[i + j] !== needle[j]) {
                // If characters do not match, set isMatch to false and break out of the inner loop
                isMatch = false;
                break;
            }
        }

        // If the needle was found in the haystack, return the starting index `i`
        if (isMatch) {
            return i;
        }
    }

    // If the needle was not found in the haystack, return -1
    return -1;
}


class Solution:
    def strStr(self, haystack: str, needle: str) -> int:
        # Length of the haystack and needle strings
        haystack_length, needle_length = len(haystack), len(needle)

        # Check all possible starting positions of needle in haystack
        for start in range(haystack_length - needle_length + 1):
            # If the substring matching the needle's length equals the needle, return the start index
            if haystack[start : start + needle_length] == needle:
                return start

        # If the needle is not found in haystack, return -1
        return -1

# The method strStr is intended to find the first occurrence of the needle string
# in the haystack string and return the index at which it begins.
# If needle is not part of haystack, it returns -1.
```

Time and Space Complexity

The time complexity of the provided code is $O((n - m + 1) * m)$, where n is the length of the haystack string and m is the length of the needle string. The `for` loop iterates up to $(n - m + 1)$ times for the worst-case scenario, and the `==` operation inside the loop takes $O(m)$ time to compare the substring to the needle.

The space complexity of the code is $O(1)$ since only a few variables are used and there is no additional space allocated that is dependent on the input size.