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 the 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 `n` 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 - n`, 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) x 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:

```
1  for i in range(n - m + 1):
2      if haystack[i : i + m] == needle:
3          return i
4  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.

Python Solution

```python
1  class Solution:
2      def strStr(self, haystack: str, needle: str) -> int:
3          # Length of the haystack and needle strings
4          haystack_length, needle_length = len(haystack), len(needle)
5
6          # Check all possible starting positions of needle in haystack
7          for start in range(haystack_length - needle_length + 1):
8              # If the substring matching the needle's length equals the needle, return the start index
9              if haystack[start : start + needle_length] == needle:
10                 return start
11
12         # If the needle is not found in haystack, return -1
13         return -1
14
15 # The method strStr is intended to find the first occurrence of the needle string
16 # in the haystack string and return the index at which it begins.
17 # If needle is not part of haystack, it returns -1.
18
```

Java Solution

```java
1  class Solution {
2      public int strStr(String haystack, String needle) {
3          // If needle is empty, the index to return is 0 (as per the problem statement).
4          if (needle.isEmpty()) {
5              return 0;
6          }
7
8          // Get the lengths of haystack and needle.
9          int haystackLength = haystack.length();
10         int needleLength = needle.length();
11
12         // Initialize pointers for haystack and needle.
13         int haystackPointer = 0;
14         int needlePointer = 0;
15
16         // Iterate through the haystack.
17         while (haystackPointer < haystackLength) {
18             // Check if the current characters in the haystack and needle are the same.
19             if (haystack.charAt(haystackPointer) == needle.charAt(needlePointer)) {
20                 // Special case: if needle length is 1 and characters match, we found the match.
21                 if (needleLength == 1) {
22                     return haystackPointer;
23                 }
24                 // Move both pointers forward.
25                 haystackPointer++;
26                 needlePointer++;
27             } else {
28                 // Current characters do not match. Reset haystackPointer back by the amount
29                 // of needlePointer had advanced, then move forward by one to search from next position.
30                 haystackPointer -= needlePointer - 1;
31                 // Reset needlePointer back to the start of the needle.
32                 needlePointer = 0;
33             }
34
35             // Check if the needle has been found within the haystack.
36             if (needlePointer == needleLength) {
37                 // The start of the substring in haystack that matches the needle
38                 // is at the difference between current haystackPointer and the length of the needle.
39                 return haystackPointer - needleLength;
40             }
41         }
42
43         // Needle was not found in the haystack. Return -1 as specified in the problem statement.
44         return -1;
45     }
46 }
47
```

C++ Solution

```cpp
1  class Solution {
2  private:
3      // Constructs the 'next' array used in the KMP algorithm to optimize matching
4      vector<int> buildNextArray(string pattern) {
5          vector<int> next(pattern.length());
6          next[0] = -1; // Initialization with -1 for the first character
7          int index = 0; // Index in the pattern string
8          int prefixIndex = -1; // Index of the longest prefix that is also a suffix
9          int patternLength = pattern.length();
10         while (index < patternLength) {
11             // When there is a mismatch or it's the first iteration
12             while (prefixIndex >= 0 && pattern[index] != pattern[prefixIndex]) {
13                 prefixIndex = next[prefixIndex];
14             index++, prefixIndex++;
15             // If we have not reached the end of the pattern
16             if (index < patternLength) {
17                 // Record the length of the longest prefix which is also suffix
18                 if (pattern[index] == pattern[prefixIndex]) {
19                     next[index] = next[prefixIndex];
20                 } else {
21                     next[index] = prefixIndex;
22                 }
23             }
24         }
25         return next;
26     }
27
28 public:
29     // Function to find the first occurrence of `needle` in `haystack`
30     int strStr(string haystack, string needle) {
31         if (needle.empty()) // If the needle is empty, return 0 as per convention
32             return 0;
33
34         vector<int> nextArray = buildNextArray(needle);
35
36         int haystackLength = haystack.length(); // Length of the haystack string
37         int needleLength = needle.length(); // Length of the needle string
38         int len = haystackLength - needleLength + 1; // Compute the limit of searching
39         for (int i = 0; i < len; ++i) {
40             int needleIndex = 0; // Index for the needle string
41             int haystackIndex = i; // Starting index in the haystack string
42             // Search while the characters match and we are within both strings
43             while (needleIndex < needleLength && haystackIndex < haystackLength) {
44                 if (haystack[haystackIndex] != needle[needleIndex]) {
45                     needleIndex = nextArray[needleIndex];
46                     continue; // Use the next array to skip some comparisons
47                 } else {
48                     break; // Mismatch without a valid sub-prefix match
49                 }
50                 ++haystackIndex, ++needleIndex;
51             }
52             // When the whole needle string has been traversed, return the starting index
53             if (needleIndex == needleLength)
54                 return haystackIndex - needleIndex;
55         }
56
57         return -1; // If the needle has not been found, return -1
58     }
59 };
60
```

Typescript Solution

```typescript
1  /**
2   * Finds the first occurrence of the 'needle' in 'haystack', and returns its index.
3   * If 'needle' is not found in 'haystack', returns -1.
4   *
5   * @param haystack - The string to be searched within.
6   * @param needle - The string to find in the haystack.
7   * @returns The index at which the needle is found in the haystack, or -1 if not found.
8   */
9  function strStr(haystack: string, needle: string): number {
10     // Length of the haystack and needle strings
11     const haystackLength: number = haystack.length;
12     const needleLength: number = needle.length;
13
14     // Early return if the needle is an empty string
15     if (needleLength === 0) return 0;
16
17     // Loop through each character in the haystack until there's no room left for the needle
18     for (let i = 0; i <= haystackLength - needleLength; i++) {
19         // Assume that the needle is found unless a mismatch is found
20         let isMatch: boolean = true;
21
22         // Check each character of the needle against the haystack
23         for (let j = 0; j < needleLength; j++) {
24             if (haystack[i + j] !== needle[j]) {
25                 // If characters do not match, set isMatch to false and break out of the inner loop
26                 isMatch = false;
27                 break;
28             }
29         }
30
31         // If the needle was found in the haystack, return the starting index `i`
32         if (isMatch) {
33             return i;
34         }
35     }
36
37     // If the needle was not found in the haystack, return -1
38     return -1;
39 }
40
```

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.