796. Rotate String

Easy String String Matching

Leetcode Link

Problem Description

The problem presents a scenario where we have two strings, `s` and `goal`, and we are tasked with determining whether `s` can be transformed into `goal` through a series of shifts. A shift operation is defined as taking the first character from `s` and moving it to the end of the string. The task is to perform as many shifts as needed to check if `s` can match `goal`. We need to return `true` if it's possible and `false` otherwise.

Intuition

The intuition behind the solution is founded on the idea of concatenation and sub-string checking. By appending string `s` to itself, we create a new string that contains all possible shifts of `s`. This is because any shift of a string of length `n` is a sub-string starting from some index `i` to index `i + n` in the double-length string. If `goal` is a sub-string of this doubled string, then it must be possible to shift the original `s` to match `goal`. It is essential, however, to ensure that `s` and `goal` are of the same length before checking for a sub-string match, as having different lengths automatically implies that no amount of shifting will make the two strings equal.

Solution Approach

The implementation of the solution is quite straightforward and elegant due to the simplicity of the logic involved. It leverages both the Python string methods and the concept of string concatenation and sub-string search. Here's how it's done using the given solution code:

1. First, the solution checks if `s` and `goal` have the same length, as different lengths would make it impossible for `s` to be shifted into `goal`. This is done with the expression `len(s) == len(goal)`. If they are not of equal length, the function will return `False`.
2. Next, if the lengths are equal, the solution concatenates string `s` with itself using `s + s`. This double-length string contains all the possible shifts of string `s`. For example, for `s = "abcde"`, `s + s = "abcdeabcde"` would include `"eabcd"`, `"deabc"`, `"cdeab"`, and so on, as its substrings.
3. The final check is to see if `goal` is a sub-string within this concatenated version of `s`. This is achieved with the expression `goal in s + s`. In Python, the `in` operator checks if one string is contained within another, returning `True` if it is found and `False` if not.

By combining these steps, the function `rotateString` assesses whether there exists a sequence of shifts that transform `s` into `goal`. If `goal` exists as a sub-string in `s + s`, that means `goal` could be reached by shifting `s` a certain number of times.

In terms of algorithms and data structures, this approach uses a linear search underneath the `in` operation, and the extra space used is directly proportional to the size of string `s` due to the concatenation. However, since string concatenation and searching are highly optimized in Python, this solution is both space and time-efficient for the problem at hand.

Example Walkthrough

Let's walk through an example to illustrate the solution approach for the problem of determining if one string can be transformed into another through cyclic shifts.

Imagine we have `s = "abc"` and `goal = "cab"`. We want to find out if by cyclically shifting `s`, we can obtain `goal`.

1. First of all, we check if both `s` and `goal` have the same length. Here, both `s` and `goal` have a length of 3, so we pass this check.
2. We then concatenate `s` with itself. So, `s + s = "abcabc"`. Notice that this new string, `"abcabc"`, contains all possible shifts of `s`:
   - The original string "abc" (starting at index 0)
   - The first shift "bca" (starting at index 1)
   - The second shift "cab" (starting at index 2)
3. Finally, we look for `goal` in this new doubled string. `goal = "cab"` is a substring of `s + s = "abcabc"`, starting from index 2 to index 4.

Because we found `goal` within `s + s`, we can confidently return `True`, which signifies that it is possible to get `goal` by shifting `s` two places to the right.

Had `goal` not been found within the double string `s + s`, we would have returned `False`, meaning no amount of shifting would have matched `s` to `goal`.

By following these steps with Python's concise string operations, we have a solution that is both elegant and efficient.

Python Solution

```python
class Solution:
    def rotateString(self, s: str, goal: str) -> bool:
        # Compare the lengths of the strings to ensure they are the same,
        # which is a prerequisite for one to be a rotation of the other.
        if len(s) != len(goal):
            return False

        # Concatenate the string 's' with itself, which covers all possible rotations.
        # If 'goal' is a rotated version of 's', it will be a substring of 's+s'.
        # Check if 'goal' is present within this concatenated string.
        return goal in (s + s)

# Example usage:
# solution = Solution()
# result = solution.rotateString("abcde", "cdeab")
# print(result)  # Output would be True since "cdeab" is a rotation of "abcde"

```

Java Solution

```java
class Solution {
    // Method to check if the string 'goal' can be obtained by rotating 's'.
    public boolean rotateString(String s, String goal) {
        // Check if the two strings are of the same length.
        // If they aren't, they cannot be rotations of each other.
        if (s.length() != goal.length()) {
            return false;
        }

        // Concatenate 's' with itself. In such a string, any rotation of 's'
        // would appear as a substring.
        String doubledS = s + s;

        // Check if the concatenated string contains 'goal' as a substring.
        // If it does, 'goal' is a rotation of 's'.
        return doubledS.contains(goal);
    }
}

```

C++ Solution

```cpp
#include <string> // Include necessary library for using strings
#include <cstring> // Include C-string library for strstr function

class Solution {
public:
    // Function to check if string 's' after rotation can become string 'goal'
    bool rotateString(std::string s, std::string goal) {
        // First check if the sizes of both strings are equal, an essential condition for rotation
        if (s.size() != goal.size()) {
            return false; // If sizes differ, one cannot be a rotation of the other
        }

        // Double the string 's' so that it contains all possible rotations
        std::string doubled_s = s + s;

        // Use the C library function 'strstr' to check if 'goal' is a substring of 'doubled_s'
        // 'strstr' returns a pointer to the first occurrence of 'goal' in 'doubled_s', or a null pointer if 'goal' is not present
        // 'data()' method provides a pointer to the start of the string data
        return strstr(doubled_s.data(), goal.data()) != nullptr;
    }
};

```

Typescript Solution

```typescript
// Checks whether the string 's' can be rotated to match the string 'goal'
function rotateString(s: string, goal: string): boolean {
    // Check if the lengths of both strings are equal, as a necessary condition for rotation
    if (s.length !== goal.length) {
        return false;
    }

    // Concatenate the goal string with itself
    const doubledGoal = goal + goal;

    // Check if the original string 's' is a substring of the concatenated string 'doubledGoal'
    // This would imply that 's' can be rotated to form 'goal'
    return doubledGoal.includes(s);
}

```

Time and Space Complexity

The given Python function `rotateString` checks if `s` rotated can be equal to `goal` by appending `s` to itself and checking if `goal` is a substring of the result.

Time Complexity

The time complexity of the function is determined primarily by the `in` operation where `goal` is checked to be a substring of `s + s`. In the worst-case scenario, the `in` statement takes `O(n*m)` time, where `n` is the length of the string `s` and `m` is the length of the string `goal`. In this code, since we ensure that `len(s) == len(goal)`, we can simplify this to `O(n^2)`.

However, most modern Python implementations use optimized substring search algorithms like the Boyer-Moore or the Knuth-Morris-Pratt algorithm, which on average can provide sub-linear performance in practice, but for the worst-case analysis, we can consider `O(n^2)`.

Space Complexity

Space complexity is determined by the additional space used by the program beyond the input and output. For this code, creating `s + s` generates a temporary string that requires `O(n)` space, where `n` is the length of `s` (since `s` and `goal` are the same length).

Thus, the space complexity of the function is `O(n)`.