459. Repeated Substring Pattern

Problem Description

The problem asks us to determine whether a given string `s` can be formed by taking a substring of itself and repeating that substring multiple times consecutively to create the string. For example, the string "abab" can be created by repeating the substring "ab" twice, so the output should be `True`. However, the string "aba" cannot be created by repeating any of its substrings, thus the output should be `False`.

Intuition

The intuition behind the solution is based on the idea that if the string `s` can be constructed by repeating a substring, then you would be able to find the beginning of the original string `s` somewhere in the middle of the string when the string is doubled. This is because the repeating pattern would continue in the doubled string and the start of the second copy of `s` in the doubled string would align with the start of some repeat of the pattern within the first copy.

The solution approach is to concatenate the string `s` with itself, forming `(s + s)`, and then searching for the original string `s` inside this new string starting from index 1. By doing this, we intentionally skip the very beginning of the first `s` to ensure we are not merely finding the original first occurrence of s. If `s` is a repeated pattern, we expect to find `s` starting at some position before we reach the end of the doubled string's first half. If `s` is found and the index is less than the length of `s`, it means `s` can indeed be constructed by repeating a substring.

For example, for `s = "abab"`:

- Doubling `s` gives us "abababab".
- Searching for "abab" inside the double string starting from the second character gives us an index of 2.
- Since 2 is less than the length of `s`, which is 4, the function returns `True`.

The Python code `(s + s).index(s, 1) < len(s)` implements this logic concisely. The method `.index()` will either return the index where `s` is found in `(s + s)`, or raise a ValueError if `s` is not found, which means the whole chain evaluates to `False`.

Solution Approach

The implementation of the solution involves a simple but clever use of string manipulation and searching. The algorithm doesn't explicitly use complex data structures or searching patterns but leverages the built-in methods in Python.

Here's a step-by-step breakdown of the solution approach:

1. Concatenate the input string `s` with itself using the `+` operator. This creates a new string which is twice the length of the original string. For an input string `s = "abab"`, after concatenation, we have a new string `"abababab"`.

2. Using the `.index()` method, search for the original string `s` within the concatenated string `(s + s)`, starting from index 1. Why index 1? Because we want to skip the first character of the concatenated string (which is where the first occurrence of `s` starts) and see if `s` appears again within what follows. We're looking to find the beginning of the second `s` in the doubled string.

3. If `s` is found and its starting index is less than the length of the original string, it confirms that the original string `s` can be created by repeating a substring. It means that the pattern started at least once more somewhere after the first character of the concatenated string, but before we reached the end of what would've been the first `s` in the doubled string.

4. If the `.index()` method returns an index that is less than the length of `s`, the method `repeatedSubstringPattern` returns `True`. Otherwise, it returns `False`.

The elegance of this solution lies in its use of string self-concatenation to simulate the process of repeating a substring—if `s` indeed can be constructed by such repetition, this pattern will reveal itself through the concatenation and be evident by the new occurrence of `s` within `(s + s)`.

In this Python code: `(s + s).index(s, 1) < len(s)`, `s` is doubled, we use the `.index()` method to search for the first occurrence of the original `s` starting from index 1, and we check if the index found is less than the length of the original string `s`. The comparison result (True or False) is then returned directly.

This method does not require extra space for another data structure, and it effectively utilizes the repeating nature of the string to check for the repeated substring pattern.

Example Walkthrough

Let's illustrate the solution approach with a small example using the string `s = "bcbcbc"`.

1. **Concatenation:** First, we concatenate the string to itself, resulting in `(s + s) = "bcbcbcbcbcbc"`. The length of the original string `s` is 6.

2. **Searching for s:** Next, we use the `.index()` method to search for the original string `s` within this new doubled string. We must start searching from position 1, i.e., from the second character, in order to skip the obvious match at the beginning of the doubled string.

3. **Index Found:** When we do this search, we find that the index where `s` appears again within the doubled string `(s + s)` is at position 2. This is because the substring "bcbc" from the original `s` can be seen starting at index 2 and again at index 4 and 6 in the doubled string.

4. **Evaluation:** The index 2 is less than the length of the original string `s` (which is 6), indicating that the string `s` can indeed be constructed by repeating a substring ("bcbc" in this case).

5. **Conclusion:** Since the `.index()` method returned an index (2) less than the length of the original string `s`, according to our solution approach, the function `repeatedSubstringPattern` should return `True`. This correctly reflects that the string "bcbcbc" can be formed by repeating the substring "bcbc" multiple times consecutively.

Python Solution

```python
class Solution:
    def repeatedSubstringPattern(self, s: str) -> bool:
        # Duplicate the string 's' by concatenating it with itself.
        # This is a common trick to identify if the string contains a repeating pattern.
        double_s = s + s

        # Try to find the original string 's' within this doubled string,
        # but starting the search from index 1 (not from the beginning)
        # to bypass the first occurrence of the string 's'.
        index_in_double_s = double_s.index(s, 1)

        # If 's' is found in double_s before the end of the original string 's',
        # then it means 's' has a repeating pattern within itself.
        # In the original string, the second occurrence must start before the string ends
        # if it is a repeated pattern.
        return index_in_double_s < len(s)

# The method 'repeatedSubstringPattern' is part of the 'Solution' class, and it
# checks if the given string 's' consists of one or more repetitions of a substring.
```

Java Solution

```java
class Solution {

    // Function to check if the string contains a repeating substring pattern.
    public boolean repeatedSubstringPattern(String str) {
        // Concatenate the string with itself.
        String doubledString = str + str;

        // Check if the concatenated string, with the first and last characters removed,
        // still contains the original string.
        // This technique works because if the original string is composed of repeated
        // of patterns, then by removing the first and the last characters, and still being able
        // to find the original string ensures that the pattern is indeed repeating.
        return doubledString.substring(1, doubledString.length() - 1).contains(str);
    }
}
```

C++ Solution

```cpp
#include <string>
using std::string;

class Solution {
public:
    // Function to check if the input string is composed of one or more repetitions of a substring.
    bool repeatedSubstringPattern(string str) {
        // Concatenate the input string with itself.
        string doubledString = str + str;

        // Try to find the original string inside the doubled string, starting from index 1.
        // The idea is that if the original string is composed a repeating pattern, the doubling
        // will cause overlaps of this pattern, and the original string should reoccur before
        // the end of the first instance of itself in the concatenation.
        size_t position = doubledString.find(str, 1); // Start the search at index 1

        // If the original string is found in the concatenated string at a position that is less
        // than the size of the original string, it means there is a repeating pattern.
        // We also exclude the trivial case of finding the first occurrence of the string
        // in the doubled string, which will always happen at index str.size().

        // Return true if we find the pattern, which indicates the input is a repetition.
        // Otherwise, return false.
        return position < str.size();
    }
};
```

Typescript Solution

```typescript
function repeatedSubstringPattern(str: string): boolean {
    // Double the input string and remove the first and last characters
    const doubledString = (str + str).slice(1, -1);

    // Check if the modified (doubled and trimmed) string includes the original string
    // If it does, it implies the original string is composed of repeated substrings
    return doubledString.includes(str);
}

// Usage of the function
// const result = repeatedSubstringPattern("abab"); // Should return true
// console.log(result);
```

Time and Space Complexity

The time complexity of the code is `O(n)` where `n` is the length of string `s`. This is because string concatenation `(s + s)` takes `O(n)` time and the subsequent `.index()` method call will in the worst-case scan the entire length of the concatenated string which is `2n`. Still, since the index is being found for a substring that we know exists (and is actually the prefix of the concatenated string), it will typically be found near the start of the second half of the concatenated string, often making the practical runtime much lower. However, for worst-case time complexity estimation, we consider it as `O(n)`.

The space complexity of the code is also `O(n)` because a new string `(s + s)` is being created that is twice the length of the original string `s`.