# 1668. Maximum Repeating Substring

Easy String String Matching

Leetcode Link

## Problem Description

The problem involves finding the highest number of times (`k`) a word can repeat itself as a contiguous subsequence within a larger string. For example, if the `sequence` is "ababc" and the `word` is "ab", then the word "ab" repeats once as a substring. However, it does not repeat twice in a row, so it cannot be considered 2-repeating. The task is to find the maximum `k`-repeating value, which means we want to find the highest number of consecutive repetitions of `word` within `sequence`.

If `word` does not appear in `sequence` even once, then the maximum `k`-repeating value is 0. The goal is to return this maximum `k` value.

## Intuition

The intuition behind the solution is a straightforward search approach. We know that the maximum `k` value cannot be more than the number of times `word` can fit into `sequence`, which is `len(sequence) // len(word)`. We start checking from the maximum possible repetition (k value) and work backwards. In each iteration, we construct a string by repeating `word` `k` times and check if that construction is a substring of `sequence`. If it is, that means `word` is `k`-repeating in `sequence`, and we have found our maximum `k`-repeating value.

The reason we start from the maximum and go backwards is efficiency; it saves us from checking all values of `k` from `1` upwards. As soon as we find the first `k` that satisfies the condition, we know that it's the maximum because all higher values would not fit the definition of a substring due to exceeding the length of `sequence`.

## Solution Approach

The given solution method `maxRepeating` defines a simple approach, utilizing a control structure to find the maximum `k`-repeating value. It uses a `for` loop and string multiplication operations to achieve this.

Here's a step-by-step overview of the algorithm:

1. First, we need to establish the upper bound of our search for `k`. The most number of times `word` can fit into `sequence` is found through integer division (`//`). This is calculated by `len(sequence) // len(word)`. It gives us the maximum number of times `word` can be repeated before it would exceed the length of `sequence`.

2. We then use a `for` loop to iterate from this maximum possible value of `k` down to `0`. The reason for iterating backwards is to find the maximum `k`-repeating value first. If we were to iterate forwards, we would need to check every possible value of `k` upto the maximum, which is less efficient.

3. In each iteration of the loop, we create a new string by multiplying `word` by `k`. The multiplication operator `*` used on strings in Python creates a new string by repeating the operand string `k` times.

4. We then check if this newly created string is a substring of `sequence` by using the `in` operator in Python, which returns `True` if the first operand is found within the second operand string.

5. If the condition `word * k in sequence` is true, the loop breaks and returns the current value of `k`, which is the maximum `k`-repeating value.

6. If the loop completes without finding any `k` value for which `word * k` is a substring of `sequence`, the function implicitly returns `None`, which is not relevant to our problem statement since we expect an integer.

7. The choice of using this specific pattern is due to its simplicity and effectiveness for the given problem. No additional data structures are needed, and the control flow along with built-in string operations is sufficient to arrive at the correct result.

With this approach, we minimize the number of checks we need to perform and ensure that we can return the highest `k` as soon as it is found.

### Example Walkthrough

Let's go through an example to illustrate the solution approach. For this example, let's take the `sequence` as "aabbabbabbaabbaabb" and the `word` as "abb".

1. **Establish the upper bound:** According to step 1, we find the maximum number of times `word` can fit into `sequence`. We get `len(sequence) // len(word)` which is `18 // 3 = 6`. Thus, the word "abb" can be repeated at most 6 times in a row without exceeding the length of the sequence.

2. **Iterate from the maximum possible value of `k` down to 0:** We start a for loop from `6` down to `1`.

3. **String multiplication and substring check:** In the first iteration, `k` is `6`, so we multiply "abb" by `6`. We check if "abbabbabbabbabbabb" (`word * k`) is a substring of the sequence "aabbabbabbaabbaabb". It isn't, so we continue the loop with `k` decremented by `1`.

4. Repeat step 3 with `k = 5, 4, 3, ...`, constructing the string "abb" * `k` each time and checking if it's a substring of "aabbabbabbaabbaabb".

5. When `k = 3`, we build "abbabbabb" (which is "abb" repeated 3 times) and check if it's a substring of the sequence. We find that "aabbabbabbaabbaabb" does contain "abbabbabb". Thus, `word` does `k`-repeat for `k = 3`.

6. The function then breaks from the loop and returns `3`, as this is the highest `k` value for which `word * k` is a subsequence of the `sequence`.

Using this approach, we efficiently found the highest number of contiguous subsequences of "abb" within the larger string "aabbabbabbaabbaabb" by starting from the largest possible repetition (k) and reducing it until we found a match.

## Python Solution

```python
class Solution:
    def max_repeating(self, sequence: str, word: str) -> int:
        """
        Find the maximum number of times 'word' can be consecutively repeated
        in 'sequence' as a substring.

        :param sequence: The string in which to search for repeating 'word'.
        :param word: The word to look for in 'sequence'.
        :return: The maximum number of times 'word' can be repeated.
        """

        # Calculate the maximum possible repetitions of 'word' within 'sequence'
        max_possible_repeats = len(sequence) // len(word)

        # Iterate over the possible repetitions, starting from the most and descending
        for k in range(max_possible_repeats, -1, -1):
            # If the repeat sequence of 'word' is within 'sequence', return that repeat count
            if word * k in sequence:
                return k

        # If no repetition of 'word' is found, we return 0
        return 0  # Technically this line is not necessary due to the loop's range

```

## Java Solution

```java
class Solution {
    // Defines a method to find the maximum number of times 'word' repeats in 'sequence'
    public int maxRepeating(String sequence, String word) {
        // Start from the maximum possible repetitions and decrement
        for (int k = sequence.length() / word.length(); k > 0; --k) {
            String repeatedWord = word.repeat(k); // Construct the word repeated 'k' times
            if (sequence.contains(repeatedWord)) { // Check if 'sequence' contains the repeated 'word'
                return k; // If found, return the current repetition count 'k'
            }
        }
        return 0; // If no repetition is found, return 0
    }
}

```

## C++ Solution

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

class Solution {
public:
    // This function finds the maximum number of times 'word' can be repeated
    // consecutively in the string 'sequence'.
    int maxRepeating(string sequence, string word) {
        int maxCount = 0; // Stores the maximum count of repeating 'word'
        string repeatedWord = word; // Starts with one 'word' and we'll append more
        int possibleRepeats = sequence.size() / word.size(); // Calculates the maximum possible times 'word' could repeat

        // Loop repeats until 'word' can no longer fit into 'sequence'
        for (int k = 1; k <= possibleRepeats; ++k) {
            // Check if the current 'repeatedWord' is a substring of 'sequence'
            if (sequence.find(repeatedWord) != string::npos) {
                maxCount = k;  // Update the maxCount to the current repetition number
            } else {
                // If the 'repeatedWord' is not in 'sequence', break out of the loop
                break;
            }

            // Append 'word' to 'repeatedWord' for the next iteration to check for the next number of repeats
            repeatedWord += word;
        }

        // Return the maxCount, which is the maximum number of times 'word' repeats consecutively in 'sequence'
        return maxCount;
    }
};

```

## Typescript Solution

```typescript
/**
 * Find the maximum number of times the word can be repeated consecutively in the sequence.
 * @param {string} sequence - The string to search within.
 * @param {string} word - The word to look for in the sequence.
 * @returns {number} The maximum number of times the word is repeated.
 */
function maxRepeating(sequence: string, word: string): number {
    // Determine the lengths of the sequence and the word
    let sequenceLength: number = sequence.length;
    let wordLength: number = word.length;

    // Start from the maximum possible repetition (sequence length divided by word length)
    // and go down to find the maximum repeating consecutive occurrences
    for (let repeatCount: number = Math.floor(sequenceLength / wordLength); repeatCount > 0; repeatCount--) {

        // Generate the word repeated 'repeatCount' times
        let repeatedWord: string = word.repeat(repeatCount);

        // Check if the sequence includes the repeated word
        if (sequence.includes(repeatedWord)) {
            // If the sequence contains the word repeated 'repeatCount' times, return 'repeatCount' as the result
            return repeatCount;
        }
    }

    // If no repetition is found, return 0
    return 0;
}

```

## Time and Space Complexity

**Time Complexity:**

The given code finds the maximum number of times the word `word` can be repeated in the `sequence` string. It checks for every `k` from `len(sequence) // len(word)` down to `0` to see if `word * k` is a substring of the `sequence`.

The time complexity of checking if a string is a substring of another string is O(n), where n is the length of the string. Here, it is checking a substring of maximum length `len(word) * k` in a string of length `len(sequence)`. This needs to be done for each k from `len(sequence) // len(word)` to `0`.

Therefore, the time complexity can be approximated as O((n/m) * (n + n - m + n - 2m + ... + m)), where n is `len(sequence)` and m is `len(word)`. This simplifies to O((n^2/m) * (n/m)) = O(n^3/m^2).

**Space Complexity:**

The space complexity of the code is O(n), with n being the maximum length of `word * k` which can be generated for the comparison with `sequence`. Essentially, the space required is for the substring that is created during `word * k`. The memory used grows linearly with the size of this generated string, which is at most the length of `sequence` itself.