Documentation for the "Count The Repetitions" problem.

Table of Contents

1. Problem Statement

We are given two strings s1 and s2, and two integers n1 and n2.

- s1 is repeated n1 times to form a long string str1.
- s2 is repeated n2 times to form a long string str2.

The task is to determine the maximum number of times str2 can be obtained by removing some characters from str1.

Example:

Input:

s1 = "acb", n1 = 4
s2 = "ab", n2 = 2

Output: 2

2. **Intuition**

To solve this problem efficiently, we need to simulate extracting s2 from str1, which is s1 repeated n1 times. Given that both n1 and n2 can be as large as 10^6, we must avoid directly constructing the strings.

The key is to keep track of how many times s2 can be matched within str1 using a cycle detection approach. Instead of iterating over all characters, we can speed up the process by recognizing repeating patterns.

3. **Key Observations**

- We are not interested in constructing the full strings str1 or str2, as this would be inefficient given the constraints.
- The challenge is to find how many times s2 can be matched in str1 without explicitly building the large strings.
- Once a cycle in the matching process is detected, we can use it to quickly jump over repeated operations.

4. **Approach**

We will iterate through the repeated instances of s1, matching characters with s2 as we go. Key steps include:

- Simulate the Matching: As we go through s1 (repeated n1 times), match characters from s2 by skipping non-matching characters.
- Cycle Detection: If the index of the character in s2 being matched repeats, it indicates a cycle. We can then calculate how many times we can skip the cycle to accelerate the matching process.
- Final Count: Once we know how many times s2 can be matched, divide it by n2 to get the final result.

The solution uses a dictionary to store the index of s2 after each iteration through s1, and whenever we encounter the same index, we know a cycle has occurred.

5. **Edge Cases**

- Case 1: s1 and s2 are identical:
  - If s1 and s2 are identical, then each repetition of s1 counts as one occurrence of s2.
- Case 2: One or both strings are extremely short or long:
  - The solution should handle small strings and large repetitions efficiently without constructing huge strings.
- Case 3: s2 is not a subsequence of s1:
  - If s2 cannot be formed from s1, then the result should be 0.
- Case 4: n1 or n2 is 1:
  - If either n1 or n2 is 1, we will need to handle this case where we don't have many repetitions of either string.

6. **Complexity Analysis**

Time Complexity

- The time complexity is O(n1 + length of s1) due to the iteration over n1 repetitions of s1. Each iteration checks each character of s1 once.
- The cycle detection mechanism ensures that we avoid reprocessing the same part of the string multiple times.

Thus, the time complexity is linear in terms of the total length of str1.

Space Complexity

- The space complexity is O(m), where m is the length of s2. We store the index of s2 positions as we process s1, which requires only a small amount of additional memory.

7. **Alternative Approaches**

i. Brute Force Approach:

   a. A brute-force solution would involve constructing str1 and str2 explicitly, which is inefficient for large inputs (up to 10^6 repetitions).

ii. Greedy Approach:

a. A greedy approach could match characters from s1 to s2 without explicitly checking for cycles, but would also suffer performance issues as n1 and n2 grow.

8. Test Cases

Test Case 1:

Input:

- s1 = "acb", n1 = 4
- s2 = "ab", n2 = 2

Output: 2

Test Case 2:

Input:

- s1 = "acb", n1 = 1
- s2 = "acb", n2 = 1

Output: 1

Test Case 3:

Input:

- s1 = "abc", n1 = 1000000
- s2 = "ab", n2 = 500000

Output: 500000

Test Case 4:

Input:

- s1 = "xyz", n1 = 1
- s2 = "abc", n2 = 1

Output: 0

9. Final Thoughts

This solution efficiently handles the problem using cycle detection to avoid brute-force string construction. It scales well even for large inputs (n1, n2 up to 10^6), making it suitable for competitive programming and real-world applications.

The use of a cycle detection mechanism makes the solution both time and space-efficient, ensuring that we avoid unnecessary computations while maintaining clarity and correctness.