686. Repeated String Match

**String Matching** 

## **Problem Description**

String ]

Medium

The task is to determine how many times we need to repeat string a such that string b becomes a substring of the repeated string a.

 If it is not possible for b to be a substring of a no matter how many times a is repeated, we should return −1. • A string repeated 0 times is an empty string, repeated once remains the same, and so on.

Here are important things to note from the problem:

Intuition

The challenge lies in finding the minimum number of repetitions needed.

## Let's say a = "abcd" and b = "cdabcdab". The string b isn't immediately a substring of a, but if we repeat a a certain number of

We first calculate the minimum number of times a must be repeated such that the length of the resulting string is equal to or just exceeds the length of b. This is because, for b to be a substring of a, the repeated a must be at least as long as b.

times, it can become one.

To find out how many repetitions are needed:

- We start by repeating string a this minimum number of times and check if b is a substring of the resultant string. If not, we increment the number of repetitions by one and check again.
- o If b is not a substring of a repeated ans times (where ans is the initial calculated number), b must start near the end of a repeated ans times
- for it to possibly be included in a further repeated a.
- o If by adding one more a, b is still not a substring, then adding one more repetition on top of that (making it two more repetitions beyond the initial ans) will cover any possible overlap of b as a substring.
- If after three attempts (ans, ans+1, and ans+2) b is not a substring, it's concluded that b cannot be made a substring by repeating a.
- First, we measure the lengths of a and b using len(), storing the lengths in variables m and n respectively. We calculate the initial number of times a needs to be repeated, which we refer to as ans. This is done as follows:

return -1

ans = ceil(n / m)

```
Next, we begin testing if b is a substring of the repeatedly joined string a. We use a for loop that runs three times,
representing the maximum number of additional repeats we decided would be necessary:
```

If b was not found to be a substring, before going for the next loop iteration, we add one more a to our list of strings t,

''.join(t). If we find b, we immediately return the current number of times a has been repeated (ans).

This is outside our loop and is our default case if b was never found within the repeated a. Remember, the code doesn't explicitly check ans+2 repetitions within the loop because appending a to t inside the for loop

happens at the end of the iteration, which means when the loop exits, we would have already checked up to ans+2 repetitions.

The above code leverages the in operator in Python for substring checking, the join method for concatenation of strings, and a

simple list to handle the repetitions. It's a straightforward implementation with the focus on optimizing the number of repetition

Suppose we have strings a = "xyz" and b = "xyzxyzx". We want to determine how many repetitions of a we need so that b

Finally, if three attempts don't result in b becoming a substring, we return -1:

We calculate the minimum number of times a must be repeated to at least cover the length of b. Using ans = ceil(n / m):

After concatenating, we have: ''.join(t) = "xyzxyzxyz"

substring after the initial number of repetitions.

The initial number of repetitions of a required is 3.

becomes a substring of the repeated a.

ans = ceil(7 / 3) = ceil(2.33) = 3

Following the solution approach:

t = ["xyz", "xyz", "xyz"]

conclude with returning -1.

Solution Implementation

We concatenate strings in t and check if b is a substring.

Now, we check if b is a substring of this. Since "xyzxyzx" in "xyzxyzxyz" returns True, we've found that b is indeed a

As a result, we don't need to add more repetitions. The minimum number of repetitions of a needed is 3. We return ans:

"xyz", "xyz"] with the concatenated string being "xyzxyzxyzxyz". We would increment ans by 1 and check again.

substring of the repeated a. If the maximum considered repetitions (ans + 2) do not satisfy the condition, the method will

Since in this example b becomes a substring after the initial number of repetitions, the algorithm finishes early and returns 3.

This process efficiently checks the minimum and only necessary additional repetitions of a to determine if b can become a

# Calculate the length of the two strings

# Calculate the minimum number of times A has to be repeated

# Create an initial string by repeating A the calculated number of times

# Also allow for B to potentially overlap at the end and beginning of A

# If not found, add another A to the end and increment the count

# If B is found in the current string, return the current count of repetitions

# so that B can possibly be a substring of the repeated A.

# Check if B is a substring of the repeated A string

# by checking one and two additional repeats of A

public int repeatedStringMatch(String A, String B) {

// Calculate the lengths of strings A and B.

lenA, lenB = len(A), len(B)

repetitions = ceil(lenB / lenA)

repeatedA = A \* repetitions

if B in repeatedA:

repeatedA += A

int lengthA = A.length();

return repetitions

for i in range(3):

from math import ceil class Solution: def repeatedStringMatch(self, A: str, B: str) -> int:

```
int lengthB = B.length();
       // Calculate the potential minimum number of repetitions required
       // for string A so that string B becomes a substring of the repeated string A.
       int repetitions = (lengthB + lengthA - 1) / lengthA;
       // Build the repeated string by repeating string A as calculated.
       StringBuilder repeatedString = new StringBuilder(A.repeat(repetitions));
       // Check up to two additional concatenations of A,
       // because the substring B could straddle the join of A.
        for (int i = 0; i < 2; ++i) {
           // Check if the current repeated string contains string B.
            if (repeatedString.toString().contains(B)) {
               // If so, return the number of repetitions used so far.
               return repetitions;
           // Otherwise, increase the number of repetitions and append string A again.
            repetitions++;
            repeatedString.append(A);
       // If string B was not found after all the iterations, return -1.
       return -1;
C++
```

```
* @param pattern - The string to repeat.
* @param target - The string to search for within the repeated `pattern`.
* @returns The minimum number of repetitions of `pattern` needed, or -1 if impossible.
function repeatedStringMatch(pattern: string, target: string): number {
   // Length of the input strings 'pattern' and 'target'.
   const patternLength: number = pattern.length,
         targetLength: number = target.length;
   // Initial calculation to determine the least number of repetitions.
   let repetitions: number = Math.ceil(targetLength / patternLength);
   // `repeatedPattern` stores the repeated string of `pattern`.
   let repeatedPattern: string = pattern.repeat(repetitions);
```

// If string B was not found, return -1

\* If such a repetition is not possible, it returns -1.

def repeatedStringMatch(self, A: str, B: str) -> int:

# Calculate the minimum number of times A has to be repeated

# Create an initial string by repeating A the calculated number of times

# Also allow for B to potentially overlap at the end and beginning of A

• The variable assignments and the ceil operation take constant time, hence 0(1).

# so that B can possibly be a substring of the repeated A.

# Check if B is a substring of the repeated A string

# by checking one and two additional repeats of A

# Calculate the length of the two strings

lenA, lenB = len(A), len(B)

repeatedA = A \* repetitions

if B in repeatedA:

return repetitions

for i in range(3):

m).

m + (ceil(n / m) \* m).

repetitions = ceil(lenB / lenA)

return -1;

**}**;

**/**\*\*

**TypeScript** 

```
# If not found, add another A to the end and increment the count
    repeatedA += A
    repetitions += 1
# If B is not found after the extra checks, return -1 indicating failure
return -1
```

# If B is found in the current string, return the current count of repetitions

O(len(t) \* m) because it concatenates len(t) strings of length m. Following this, the in operation has a worst-case time complexity of O((len(t) \* m) + n) as it needs to check substring b in the concatenated string. • The append operation inside the loop takes 0(1) time. However, as the string concatenation inside the loop occurs in each iteration, it increases the total length of the concatenated string by mevery time. So, by the third iteration, the join could be operating on a string of length up to 3 \*

- Worst-case time complexity is 0((ans + 2) \* m + n), reflecting the last iteration of the loop where ans could be incremented twice. The space complexity is determined by: • The space needed by the list t and the strings created by ''.join(t). The maximum length of the joined string can go up to 3 \* m + (ceil(n /
- Time Complexity: 0((ans + 2) \* m + n)Space Complexity: 0(3 \* m + (ceil(n / m) \* m))

m) \* m). • Hence, the worst-case space complexity is 0(3 \* m + (ceil(n / m) \* m)).

We only need to check up to two more repetitions of a beyond the initial calculated number of times. The reasoning is as follows:

**Solution Approach** The implementation closely follows the intuition:

Here, ceil is a mathematical function from the math module that takes a float and rounds it up to the nearest integer. This makes sure that if b is not completely covered by multiples of a, we round up to ensure complete coverage. Then we initialize a list t that will contain the repeated string a. We multiply the list [a] by ans to repeat the string a that many times initially: t = [a] \* ans

for \_ in range(3): if b in ''.join(t): return ans In this loop, we join the elements of t into one string using ''.join(t) and check if b is a substring of this string with b in

effectively repeating a one more time: ans += 1t.append(a) We increment ans by 1 for each additional repeat. Continuously checking every time after appending a.

checks. **Example Walkthrough** Let's illustrate the solution approach with a small example:

We measure the lengths of both strings. For a, the length, m, is 3 and for b, the length, n, is 7.

We prepare our list t to contain the repeated string a. Multiplying the list [a] by ans we get:

return 3 If b had not been a substring, we would add another a to t and check again. In this case, it would become ["xyz", "xyz",

**Python** 

repetitions += 1 # If B is not found after the extra checks, return -1 indicating failure return -1 Java class Solution {

class Solution { public: int repeatedStringMatch(string A, string B) { // Calculate the lengths of strings A and B int lengthA = A.size(), lengthB = B.size(); // Calculate the initial repeat count to cover the length of string B int repeatCount = (lengthB + lengthA - 1) / lengthA; // Create an empty string t for concatenation string t = ""; // Build the initial repeated string with repeatCount times of A for (int i = 0; i < repeatCount; ++i) {</pre> t += A;// Check up to 2 more times of string A for the presence of B in t for (int i = 0; i < 2; ++i) { // If string B is found in t, return the current repeat count if (t.find(B) != string::npos) { return repeatCount; // Increase repeat count and append string A to t ++repeatCount; t += A;

// We check up to 2 times beyond the initial calculated repetitions. // This is because the 'target' could start at the end of one repetition and end at the start of the following. for (let i = 0; i < 3; i++) { // Check if `target` is in the current `repeatedPattern`. if (repeatedPattern.includes(target)) { // If found, return the current count of repetitions. return repetitions; // If not found, increment the repetition count and append `pattern` to `repeatedPattern` again. repetitions++; repeatedPattern += pattern; // If the loop ends and `target` wasn't found in any of the repetitions, return -1. return -1; from math import ceil class Solution:

\* Determines the minimum number of times `pattern` must be repeated such that `target` is a substring of the repeated `pattern`.

Time and Space Complexity The time complexity of the given code can be analyzed based on the operations it performs:

• Creating t, which is a list of copies of string a, takes O(ans) time in the worst case, as it depends on the initial value of ans, which is ceil(n /

• The for loop will run at most 3 times, with each iteration including a ''.join(t) and testing if b in ''.join(t). The join operation takes

Given these considerations, we estimate the time complexity as: