686. Repeated String Match

Medium String String Matching

Leetcode Link

Problem Description

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`.

Here are important things to note from the problem:

- 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.

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

Intuition

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 times, it can become one.

To find out how many repetitions are needed:

1. 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`.
2. 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.
3. We only need to check up to two more repetitions of `a` beyond the initial calculated number of times. The reasoning is as follows:
    - 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`.
    - 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.
4. 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`.

Thus, the solution lies in trying at most three different numbers of repetitions and checking for the substring condition. If none meet the condition, then we return `-1`.

Solution Approach

The implementation closely follows the intuition:

1. First, we measure the lengths of `a` and `b` using `len()`, storing the lengths in variables `m` and `n` respectively.
2. We calculate the initial number of times `a` needs to be repeated, which we refer to as `ans`. This is done as follows:

```
ans = ceil(n / m)
```

   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.

3. 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
```

4. 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:

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

5. 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`, effectively repeating `a` one more time:

```
ans += 1
t.append(a)
```

   We increment `ans` by 1 for each additional repeat. Continuously checking every time after appending `a`.

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

```
return -1
```

   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 checks.

Example Walkthrough

Let's illustrate the solution approach with a small example:

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

Following the solution approach:

1. We measure the lengths of both strings. For `a`, the length, `m`, is `3` and for `b`, the length, `n`, is `7`.
2. We calculate the minimum number of times `a` must be repeated to at least cover the length of `b`. Using `ans = ceil(n / m)`:

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

   The initial number of repetitions of `a` required is `3`.

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

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

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

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

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

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

```
return 3
```

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

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 substring of the repeated `a`. If the maximum considered repetitions (`ans + 2`) do not satisfy the condition, the method will conclude with returning `-1`.

Python Solution

```python
from math import ceil

class Solution:
    def repeatedStringMatch(self, A: str, B: str) -> int:
        # Calculate the length of the two strings
        lenA, lenB = len(A), len(B)

        # Calculate the minimum number of times A has to be repeated
        # so that B can possibly be a substring of the repeated A.
        repetitions = ceil(lenB / lenA)

        # Create an initial string by repeating A the calculated number of times
        repeatedA = A * repetitions

        # Check if B is a substring of the repeated A string
        # Also allow for B to potentially overlap at the end and beginning of A
        # by checking one and two additional repeats of A
        for i in range(3):
            # If B is found in the current string, return the current count of repetitions
            if B in repeatedA:
                return repetitions

            # 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

```

Java Solution

```java
class Solution {
    public int repeatedStringMatch(String A, String B) {
        // Calculate the lengths of strings A and B.
        int lengthA = A.length();
        int lengthB = B.length();

        // Calculate the potential minimum number of repetitions required
        // for B to be a substring. This is determined by dividing the length of B by the length of A.
        int repetitions = (lengthB - lengthA > 1) ? lengthB / lengthA : 1;

        // 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 < 3; 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++ Solution

```cpp
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) ? lengthB / 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) {
            t += A;
        }

        // Check up to 2 more times of string A for the presence of B in t
        for (int i = 0; i < 3; ++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;
        }

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

```

Typescript Solution

```typescript
/**
 * Determines the minimum number of times `pattern` must be repeated such that `target` is a substring of the repeated `pattern`.
 * If such a repetition is not possible, it returns -1.
 *
 * @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;
    const 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);

    // 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;
}

```

Time and Space Complexity

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

- The variable assignments and the `ceil` operation take constant time, hence $O(1)$.
- 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 / m)`.
- 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 $O(len(t) * a)$ because it concatenates `len(t)` strings of length `a`. 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 $O(1)$ time. However, as the string concatenation inside the loop occurs in each iteration, it increases the total length of the concatenated string by `m` every time. So, by the third iteration, the `join` could be operating on a string of length up to $3 * m * (ceil(n / m) * m)$.

Given these considerations, we estimate the time complexity as:

- Worst-case time complexity is $O((ans + 2) * m + n)$, 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 $3 * m * (ceil(n / m) * m)$.
- Hence, the worst-case space complexity is $O(3 * m * (ceil(n / m) * m))$.

Time Complexity: $O((ans + 2) * m + n)$

Space Complexity: $O(3 * m * (ceil(n / m) * m))$