1071. Greatest Common Divisor of Strings

Easy Math String

Problem Description

The provided problem involves finding the largest string `x` that can divide two input strings `str1` and `str2`. One string divides another if it can be repeated some number of times to obtain the other string. For example, if we have `str1 = "abcabc"` and `str2 = "abc"`, then `str2` divides `str1` because we can concatenate `str2` with itself twice to get `str1`. In other words, `str1` is the result of appending `str2` to itself multiple times. The challenge is to identify the largest common string that can divide both `str1` and `str2` in a similar fashion.

To illustrate further, if `str1` is "ababab" and `str2` is "abab", the string "ab" is the answer since it is the largest string that divides both `str1` and `str2`.

Intuition

The underlying concept used in the provided solution is based on the mathematical idea of finding the greatest common divisor (GCD), but rather than numbers, we're dealing with strings. The key insight is that if there exists such a common string `x` that divides both `str1` and `str2`, then the concatenation of `str1` with `str2` (`str1 + str2`) should be the same as the concatenation of `str2` with `str1` (`str2 + str1`). If this condition doesn't hold, it implies there is no such common string and the answer is an empty string.

Assuming the condition is true, we can find the length of the largest string `x` using the greatest common divisor of the lengths of `str1` and `str2`. This is because the repeating pattern of string `x` must be a factor of both string lengths in order for it to divide both strings completely. Therefore, we find the GCD of the lengths of `str1` and `str2` and then return the substring of `str1` up to that length. This substring is the largest common divider `x`.

Solution Approach

The solution provided takes advantage of Python's built-in `gcd` function from the `[math](/problems/math-basics)` library to calculate the greatest common divisor of the lengths of the two input strings `str1` and `str2`. This is crucial for the solution, as the `gcd` represents the length of the largest string that can divide both strings, if such a string exists.

Here is how the solution unfolds:

1. **Check for Compatibility**: The first step in the provided solution checks whether the strings are compatible to have a common divisor by concatenating `str1` with `str2` and comparing it to `str2` concatenated with `str1`. This is facilitated by the operation `if str1 + str2 != str2 + str1`, which ensures that the pattern of characters in both strings is compatible for them to have a common divisor string. If the check fails, the two strings cannot have a common divisor and the function immediately returns an empty string `''`.
2. **Find Length of the Common Divisor**: If the strings pass the compatibility check, the next step is to determine the length of the largest common divisor string. This is where we use the `gcd` function, by calling `n = gcd(len(str1), len(str2))`. The `gcd` function takes the lengths of `str1` and `str2` and returns the greatest common divisor of these two numbers.
3. **Extract the Common Divisor String**: With the length `n` obtained from the `gcd` function, the final step is to extract the common divisor string from `str1`. This is accomplished by the expression `return str1[:n]`. The `[:n]` slice operation on `str1` returns the substring of `str1` from the beginning up to the `n-th` character (exclusive), which is the required largest string that divides both `str1` and `str2`.

To summarize, the algorithm consists of concatenation for compatibility checking and the greatest common divisor calculation, both of which are simple, efficient operations. The lack of any complex data structures and the use of a mathematical pattern of common division make this solution both elegant and effective. It leverages the fact that if a common divisor string exists, the repeating pattern must align with the gcd of the string lengths.

Example Walkthrough

Let's use a small example to illustrate the solution approach. Suppose we have two strings `str1 = "abab"` and `str2 = "ab"`. We need to find the largest string `x` that can divide both `str1` and `str2`.

1. **Check for Compatibility**: We first check if concatenating `str1` with `str2` is equal to `str2` concatenated with `str1`.
   - `str1 + str2 = "abab" + "ab" = "ababab"`
   - `str2 + str1 = "ab" + "abab" = "ababab"`

   Since `str1 + str2` is equal to `str2 + str1`, the two strings are compatible, and therefore, it's possible they have a common divisor string.
2. **Find Length of the Common Divisor**: Next, we find the greatest common divisor of the lengths of `str1` (4) and `str2` (2).
   - `gcd(4, 2) = 2`

   The `gcd` tells us that the length of the largest string that possibly divides both `str1` and `str2` is 2.
3. **Extract the Common Divisor String**: Since the `gcd` is 2, we take the first 2 characters from `str1` to form our largest divisor string `x`.
   - `str1[:2] = "ab"`

   So, "ab" is the string we expect to divide both `str1` and `str2`. To verify, we can see that:
   - `str1` divided by "ab" is `"abab" / "ab" = "ab"`, which is the repetition of "ab" twice.
   - `str2` divided by "ab" is `"ab" / "ab" = ""`, which is the repetition of "ab" once.

The largest string that can divide both `str1` and `str2` is "ab", which matches our result from the solution approach. The solution is elegant and efficient, leveraging simple concatenation and the mathematical concept of greatest common divisor to determine the existence and length of a common divisor string.

Solution Implementation

Python

```python
from math import gcd

class Solution:
    def gcdOfStrings(self, str1: str, str2: str) -> str:
        # Check if the concatenation of the two strings in different order results in the same string.
        # This is required as the strings should be made of the same substrings for them to have a common divisor.
        if str1 + str2 != str2 + str1:
            # If they don't form the same string when concatenated in different orders,
            # there is no common divisor string and hence return an empty string.
            return ''

        # Find the gcd of the lengths of the two strings.
        # The gcd of the lengths will give us the maximum length the common divisor string can have.
        length_gcd = gcd(len(str1), len(str2))

        # Return the substring from 0 to length_gcd from the first string,
        # which is the greatest common divisor string.
        return str1[:length_gcd]
```

Java

```java
class Solution {
    // Function to find the greatest common divisor of lengths of two strings.
    // This GCD can be used to find the longest substring that can construct
    // the given strings by repeated concatenation.
    public String gcdOfStrings(String str1, String str2) {
        // Check if the two strings can be constructed from a common substring
        // Only if str1+str2 equals str2+str1, they have a common divisor string
        if (!(str1 + str2).equals(str2 + str1)) {
            return ""; // If not, return an empty string as there is no common divisor
        }
        // Calculate the GCD of lengths of the two strings
        int len = gcd(str1.length(), str2.length());
        // The substring from the beginning of str1 with length 'len' is the gcd string
        return str1.substring(0, len);
    }

    // Helper function to calculate the greatest common divisor (GCD) of two integers
    // It uses the Euclidean algorithm to find the GCD
    private int gcd(int a, int b) {
        // Base case: if b is 0, then a is the GCD (as GCD(a, 0) = a)
        // Recursive step: GCD(a, b) = GCD(b, a mod b)
        return b == 0 ? a : gcd(b, a % b);
    }
}
```

C++

```cpp
#include <algorithm> // include algorithm header for std::gcd

class Solution {
public:
    // Function to find the greatest common divisor of strings str1 and str2
    string gcdOfStrings(string str1, string str2) {
        // check if concatenating the strings in both orders gives the same result
        // this is required because two strings can only be multiples of each other
        // if this condition is true
        if (str1 + str2 != str2 + str1) {
            return ""; // if they are not equivalent, return an empty string
        }

        // calculate the greatest common divisor (GCD) of the sizes of the two strings
        // std::gcd is available in C++17 and later. For C++14 and earlier, use a custom gcd function
        int gcdValue = std::gcd(str1.size(), str2.size());

        // return the common divisor string which is the substring from start of str1 to its GCD length
        return str1.substr(0, gcdValue);
    }
};
```

TypeScript

```typescript
// Function to calculate the greatest common divisor (GCD) of two numbers
// Uses Euclidean algorithm
function gcd(a: number, b: number): number {
    while (b !== 0) {
        let t = b;
        b = a % b;
        a = t;
    }
    return a;
}

// Function to find the greatest common divisor of strings str1 and str2
function gcdOfStrings(str1: string, str2: string): string {
    // Check if concatenating the strings in both orders gives the same result
    // This is required because two strings can only be multiples of each other
    // if this condition is true
    if (str1 + str2 !== str2 + str1) {
        return ""; // If they are not equivalent, return an empty string
    }

    // Calculate the greatest common divisor (GCD) of the lengths of the two strings
    const gcdValue = gcd(str1.length, str2.length);

    // Return the common divisor string which is the substring from start of str1 to its GCD length
    return str1.substring(0, gcdValue);
}
```

```python
from math import gcd

class Solution:
    def gcdOfStrings(self, str1: str, str2: str) -> str:
        # Check if the concatenation of the two strings in different order results in the same string.
        # This is required as the strings should be made of the same substrings for them to have a common divisor.
        if str1 + str2 != str2 + str1:
            # If they don't form the same string when concatenated in different orders,
            # there is no common divisor string and hence return an empty string.
            return ''

        # Find the gcd of the lengths of the two strings.
        # The gcd of the lengths will give us the maximum length the common divisor string can have.
        length_gcd = gcd(len(str1), len(str2))

        # Return the substring from 0 to length_gcd from the first string,
        # which is the greatest common divisor string.
        return str1[:length_gcd]
```

Time and Space Complexity

The code contains the `gcdOfStrings` method that finds the greatest common divisor (GCD) of lengths of two strings `str1` and `str2` to determine the largest string that can be repeatedly used to construct `str1` and `str2`.

Time Complexity:

The time complexity of this function mainly depends on two operations: the string concatenation operation (`str1 + str2` and `str2 + str1`) and the greatest common divisor computation (`gcd(len(str1), len(str2))`).

1. String Concatenation: If `str1` is of length `n` and `str2` of length `m`, the concatenation will take `O(n + m)` time as each character from both strings need to be combined once.
2. Checking String Equality: Comparing the concatenated strings takes `O(n + m)` time. If the strings differ, the method returns early.
3. Greatest Common Divisor: The `gcd` function typically employs Euclid's algorithm, which has a time complexity of `O(log(min(n, m)))` where `n` and `m` are the lengths of the strings.

Therefore, the overall time complexity combines these operations resulting in `O(n + m + log(min(n, m)))`. However, the dominating factor for large values of `n` and `m` will tend to be the concatenation and comparison, so we can approximate the time complexity as `O(n + m)`.

Space Complexity:

The space complexity of the `gcdOfStrings` function can be analyzed as follows:

1. Temporary Strings: The creation of concatenated strings `str1 + str2` and `str2 + str1` requires additional space of `O(n + m)`.
2. GCD Computation: The `gcd` function itself may use constant space, `O(1)`, if the Euclidean algorithm is implemented in an iterative manner.

As such, no additional space that grows with the input size is required except for the string concatenation. Hence, the space complexity is `O(n + m)`.