2546. Apply Bitwise Operations to Make Strings Equal

Medium Bit Manipulation String

Leetcode Link

Problem Description

You are provided with two binary strings, `s` and `target`, that have the same length, `n`. Your task is to transform string `s` into `target` by performing a specific type of operation as many times as necessary. The operation you are allowed to perform involves choosing two different indices `i` and `j` within the string `s` (where `0 <= i, j < n`) and then simultaneously applying the following changes:

- Change `s[i]` to be the result of `s[i] OR s[j]`.
- Change `s[j]` to be the result of `s[i] XOR s[j]`.

By repeatedly applying this operation, your goal is to determine whether it is possible to make string `s` identical to `target`. If it is possible, you should return `true`; otherwise, if it cannot be done, return `false`.

It is important to note that 'OR' and 'XOR' are binary operations. 'OR' assigns a 1 if at least one of the operands is 1, and 'XOR' assigns a 1 only if one of the operands is 1 and the other is 0.

An example of these operations: if `s = "0110"`, by choosing `i = 0` and `j = 2`, you would end up with `s[0] = "0" OR "1" = "1"` and `s[2] = "0" XOR "1" = "1"`, which would transform `s` to "1110".

Intuition

Upon reviewing the allowable operation, we notice something crucial: the 'OR' operation can only set bits to 1, and never to 0. Also, once a bit is set to 1, it can never be changed back to 0 because the 'OR' operation will always retain the 1, and the 'XOR' will either leave it as 1 or change it to 0 based on the other bit being 1 or 0, respectively. Considering this, it becomes clear that if the `target` contains a 1 in any position, `s` must also contain at least one 1 somewhere within it. Without the 1 in `s`, it would be impossible to match `target` through the available operation since you can't create a 1 from a string of all zeroes.

Upon further examination, we realize that the specific positions of the 1s in `s` and `target` do not matter. The key insight is that if `s` contains at least one 1, it can be propagated to any position using the operation. Conversely, if `s` contains no 1s, it is impossible to turn it into `target` if `target` contains at least one 1.

The solution relies on the simple check of whether there is a 1 present in both `s` and `target`, or absent in both. This straightforward approach works because the presence of at least one 1 in both strings confirms the possibility of manipulating `s` to match `target`, and the absence of 1s in both confirms that they are already equivalent in terms of the transformation capability.

Solution Approach

The implementation of the solution is as direct as the intuition suggests: we just need to check for the presence of the digit `1` in both strings `s` and `target`. The Solution class contains a single method, `makeStringsEqual`, which loops over each string to find whether `1` is present. However, since we're using Python, the implementation leveraging Python's expressiveness becomes really concise and doesn't even require a loop.

This is accomplished by the expression `("1" in s) == ("1" in target)`. In Python, the `in` keyword checks if the substring `'1'` exists within `s` and `target`, respectively, and returns a boolean value. This check is performed for both strings and then compared with `==`.

If both strings contain a `1`, or if both do not contain a `1`, then the expression evaluates to `True`, and otherwise, it evaluates to `False`.

There are no complicated data structures, algorithms, or patterns used here because the crux of the problem is a simple binary logic evaluation. There's no need for dynamically building up solutions, which would usually require more complex data structures like lists or dictionaries, nor are there any recursive or iterative processes that would necessitate control structures or specific algorithmic patterns.

In other words, the solution is highly optimized by taking advantage of the logical characteristics of the problem rather than computing power. Due to its simplicity and the absence of loops, the time complexity of this implementation is O(1), meaning it requires constant time to run regardless of the length of the input strings.

Example Walkthrough

Let's take two binary strings s = "0110" and target = "1101" and see how we can apply the solution approach to determine if we can transform `s` into `target` by applying the specified operation.

First, we need to check for the presence of the digit '1' in both strings. For string `s`, we see that '1' appears in position 1 and 2 (0-indexed), and for `target`, '1' appears in position 0, 2, and 3. So, yes, '1' is present in both strings. According to the intuition from the problem description, this means that it is potentially possible to transform `s` into `target`.

The operation we can perform is choosing two different indices `i` and `j` such that we apply:

- `s[i] = s[i] OR s[j]`.
- `s[j] = s[i] XOR s[j]`.

Since we've determined that both strings contain at least one '1', we know that we can spread this '1' throughout the string `s` by strategically choosing indices for our operation.

In our example, we can start by choosing `i = 1` and `j = 3`:

- After the operation `s[i] OR s[j]`, `s[1]` remains `1` since `1 OR 0` is `1`.
- After the operation `s[i] XOR s[j]`, `s[3]` becomes `1` since `1 XOR 0` is `1`.

So now `s` becomes "0111". We can see that we're able to propagate the '1' to the third position without altering the other '1's in `s`.

The solution can be abstracted away from such specific steps because all we need to know is that the presence of a '1' in `s` guarantees that we can use the operation to replicate `1`s as needed across the entire string to match the `target`. Therefore, for the strings given above, we use the fact that both `s` and `target` include the digit '1' and the solution method returns `True`.

The Python expression `("1" in s) == ("1" in target)` evaluates to `True` because both strings contain the digit '1', confirming the possibility of their transformation.

Indeed, using the logic from the problem description, for any pair of strings where this expression is `True`, the transformation is possible, and for any pair where this is not `True`, the transformation is not possible. This keeps the implementation very concise and the complexity to O(1).

Python Solution

```python
class Solution:
    def make_strings_equal(self, s: str, target: str) -> bool:
        # The function checks whether both input strings 's' and 'target'
        # either contain the character '1' or both do not.
        # The comparison ('1' in s) == ('1' in target) evaluates to True in two cases:
        # 1. '1' is in both 's' and 'target'.
        # 2. '1' is in neither 's' nor 'target'.
        # The function returns True if the strings match the condition above, otherwise False.

        # Check if '1' is in both strings or absent in both
        return ('1' in s) == ('1' in target)

```

Java Solution

```java
class Solution {
    // Method to check if two strings can be made equal by removing characters.
    public boolean makeStringsEqual(String s, String target) {
        // The method checks if both strings contain the character "1"
        // It returns true if both strings either contain or do not contain "1"
        // This implies both strings can be made equal by removing all other characters
        // as the existence of "1" is the only condition checked.
        return s.contains("1") == target.contains("1");
    }
}

```

C++ Solution

```cpp
#include <algorithm> // Include algorithm header for count function
#include <string>    // Include string header for string type

class Solution {
public:
    // Function to determine if two strings can be made equal by rearranging
    // Assumes strings consist of '0's and '1's
    // Returns true if both strings can be made equal, false otherwise
    bool makeStringsEqual(string s, string target) {
        // Count the number of '1's in the first string
        // Convert the count to a boolean indicating the presence of '1'
        bool hasOneInS = std::count(s.begin(), s.end(), '1') > 0;

        // Count the number of '1's in the target string
        // Convert the count to a boolean indicating the presence of '1'
        bool hasOneInTarget = std::count(target.begin(), target.end(), '1') > 0;

        // Check if both strings have '1's or both do not have '1's
        // This is because we can only rearrange the characters, not modify them
        // If one string has a '1' and the other does not, they cannot be made equal
        return hasOneInS == hasOneInTarget;
    }
};

```

Typescript Solution

```typescript
/**
 * Determines if string `s` can be made equal to string `target` by only changing 0's to 1's
 * @param {string} s - The original string to compare.
 * @param {string} target - The target string to compare against.
 * @returns {boolean} - A boolean value indicating whether the strings can be made equal.
 */
function makeStringsEqual(s: string, target: string): boolean {
    // Check if both strings `s` and `target` contain the character '1'.
    // Since we can only change 0's to 1's, the presence of '1' in both strings is a prerequisite for equality.
    return s.includes('1') === target.includes('1');
}

```

Time and Space Complexity

The time complexity of the code provided is `O(n)`, where `n` is the length of the string `s`. This is because checking for the existence of the character "1" in the string involves iterating over each character of the string in the worst case.

The space complexity of the code is `O(1)` since it only uses a constant amount of space regardless of the size of the input strings. The boolean operation does not depend on the length of the string and does not require additional space proportional to the size of the input.