1427. Perform String Shifts

Easy Array Math String

Leetcode Link

Problem Description

In this problem, we are given a string `s` that consists of lowercase English letters, and a matrix `shift` consisting of pairs `[direction, amount]`. We are tasked with performing a series of shift operations on the string according to the instructions provided by each element in the `shift` matrix. The `direction` indicates whether the shift should be to the left (`0`) or to the right (`1`) and `amount` tells us how many times the shift should occur.

A left shift (`direction == 0`) means removing the first character of `s` and appending it to the end. Conversely, a right shift (`direction == 1`) means removing the last character of `s` and adding it to the beginning. After applying all shifts in the order they appear in the `shift` matrix, our goal is to return the resultant string.

To illustrate, if we have string `s = "abcde"`:

- A left shift of `1` would change the string to `"bcdea"`.
- A right shift of `1` would change the string to `"eabcd"`.

We need to ensure that we carry out these operations in an efficient manner to determine the final state of the string `s`.

Intuition

To efficiently perform the shift operations on the string, we notice that shifts in opposite directions counteract each other. That means if we shift a string to the left by `2` positions and then to the right by `2` positions, we end up with the original string. Furthermore, shifting the string by its length or multiples of its length also results in the same string. Therefore, we can simplify the problem by combining all the shifts into a single effective shift.

To achieve this, we iterate over the `shift` matrix and maintain a variable `x` to calculate the net effect of all shifts. For left shifts, we can subtract the amount from `x`, and for right shifts, we can add the amount to `x`. By doing this for each shift and then simplifying the net shifts (`x`) modulo the length of the string, we arrive at a single shift value that represents the overall effect of all the given shifts.

Once the effective shift is determined, we can perform the shift in one step. If `x` is positive, it indicates a right shift; if negative, a left shift. The implementation in Python takes advantage of string slicing to perform this operation:

- `s[-x:]`: This slice takes the last `x` characters of `s`, which is what moves to the start of the string in a right shift.
- `s[:-x]`: This slice takes all characters of `s` except the last `x`, which remain in their original order after the shift.

Combining these two slices, we create the new string post shift, `s[-x:] + s[:-x]`, effectively performing the necessary left or right shift in a single operation. This approach effectively reduces the complexity of the problem from potentially having to do multiple shifts to simply calculating the net shift and applying it once. This not only simplifies the problem but also ensures that the solution is efficient, even when the number of shift operations is large.

Solution Approach

The provided solution follows a direct algorithmic approach to combine all the individual shift commands into one effective shift operation and then apply it to the string `s`. Here is a step-by-step walkthrough of the implementation:

1. Initialize a variable `x` to `0`. This will keep track of the net number of shifts needed. When we encounter a left shift, we will decrease `x`, and for a right shift, we will increase `x`.

2. Loop through each shift operation in the `shift` matrix. Each operation is a list containing two elements: `direction` and `amount`. We check the direction:

   - If the `direction` is `0` (left shift), we convert the `amount` to a negative number (by doing `-b`) and add it to `x`.
   - If the `direction` is `1` (right shift), we simply add the `amount` to `x`.

   In Python, this can be compactly written as:

```
for a, b in shift:
    x += b if a else -b
```

3. After processing all shift operations, we normalize the net total shifts `x` by taking the remainder when `x` is divided by the length of `s` (`x %= len(s)`). This is an essential step because shifting the string by its length (or multiples thereof) results in the same string, so shifts beyond that point are redundant.

   In mathematical form, this step is expressed as:

```
x %= len(s)
```

   If `x` is positive, it represents a net right shift; if negative, it represents a net left shift. In either case, since the remainder operation normalizes `x` within the range from `0` to `len(s) - 1`, we have a guaranteed valid index for the slicing operation in the next step.

4. Finally, we apply the effective shift to `s` using Python's string slicing feature:

   - `s[-x:]` generates a substring consisting of the last `x` characters in `s`.
   - `s[:-x]` creates a substring of all characters in `s` except for the last `x`.

   The result is then concatenated to get the shifted string. As `x` has been adjusted to be within the proper index range, these slices will always be valid. The Python code for this operation is:

```
return s[-x:] + s[:-x]
```

This method uses the string slicing capability of Python, which is very efficient, to perform the shifting operation in constant time, which is much more efficient than performing each shift individually. The space complexity of the solution is constant since it uses a fixed amount of extra space, regardless of the input size, and the time complexity is linear relative to the number of shift operations due to the single pass through the `shift` array.

Example Walkthrough

Let's consider a small example to illustrate the solution approach:

Suppose we have a string `s = "abcdef"` and a `shift` matrix with the following operations: `[[0, 1], [1, 2], [0, 3]]`.

1. We would initialize our variable `x` to `0`, which will keep track of the net number of shifts.
2. The first operation is a left shift ([0, 1]), so we decrease `x` by 1. Now `x` becomes `-1`.
3. The second operation is a right shift ([1, 2]), so we increase `x` by 2. Now `x` becomes `1` (`-1 + 2`).
4. The final operation is a left shift ([0, 3]), so we decrease `x` by 3. Now `x` becomes `-2` (`1 - 3`).
5. Once we've processed all operations, we will normalize `x` with the length of `s` which is 6. We compute `x %= len(s)`, which results in `x` becoming `4` because `-2 % 6` is `4`. It means we have a net right shift of `4`.
6. We now apply this effective shift to string `s` with a single slicing operation. The substring from the last `x` characters will be `s[-4:]`, which gives us `"cdef"`. The substring from the start to the point before the last `x` characters will be `s[:-4]`, giving us `"ab"`.
7. Combining these two substrings to perform the net right shift operation, we get the transformed string `"cdefab"`.

Thus, after applying an algorithmic approach, we were able to reduce multiple shifts to a single operation and efficiently determine the final state of `s` after all the shifting instructions.

Python Solution

```
from typing import List

class Solution:
    def stringShift(self, s: str, shifts: List[List[int]]) -> str:
        # Initialize a variable to keep track of the net shift amount
        net_shift = 0

        # Loop through each shift command in the shifts list
        for direction, amount in shifts:
            # If direction is 0, shift left; otherwise, shift right.
            # Left shift can be considered as a negative shift in calculations,
            # so we negate the amount for left shifts.
            if direction == 0:
                net_shift -= amount
            else:
                net_shift += amount

        # The effective shift is the net shift modulo the string's length
        # This accounts for shifts that exceed the string length.
        effective_shift = net_shift % len(s)

        # Apply the effective shift on the string and return the result.
        # We use negative indexing to wrap around the string.
        return s[-effective_shift:] + s[:-effective_shift]

# Example usage:
# solution = Solution()
# result = solution.stringShift("abcdefg", [[1, 1], [1, 1], [0, 2], [1, 3]])
# print(result)  # Expected output would be the final shifted string.
```

Java Solution

```
class Solution {

    // This method takes a String 's' and performs a series of shifts according to 'shifts' array.
    public String stringShift(String s, int[][] shifts) {
        int totalShifts = 0; // This variable will accumulate the effective number of shifts.

        // Iterate through each shift operation described by the 'shifts' array.
        for (int[] shiftOperation : shifts) {
            // The direction of the shift (0 for left shift, 1 for right shift).
            int direction = shiftOperation[0];
            int amount = shiftOperation[1]; // The amount to shift.

            // Convert left shifts into negative values to simplify the calculation.
            if (direction == 0) {
                amount = -amount;
            }

            // Accumulate the effective shift amount.
            totalShifts += amount;
        }

        int stringLength = s.length();
        // Normalize the effective shift to be within the range of the string length.
        // This also accounts for the shifts that go beyond the string length.
        totalShifts = (totalShifts % stringLength + stringLength) % stringLength;

        // Perform the shift by using substring.
        // Extract the end of the string that comes to the front and concatenate with
        // the beginning of the string that goes to the end.
        return s.substring(stringLength - totalShifts) + s.substring(0, stringLength - totalShifts);
    }
}
```

C++ Solution

```
#include <string>
#include <vector>

using std::string;
using std::vector;

class Solution {
public:
    string stringShift(string s, vector<vector<int>>& shifts) {
        int totalShift = 0; // Will hold the net number of shifts to apply.

        // Loop through each shift operation in the array.
        for (const auto& shiftOp : shifts) {
            int direction = shiftOp[0]; // Direction of shift: 0 for left, 1 for right.
            int amount = shiftOp[1];    // Amount to shift.

            // If the direction is 'left' (0), we convert amount to a negative value.
            if (direction == 0) {
                amount = -amount;
            }

            // Accumulate the effective shift amount.
            totalShift += amount;
        }

        int n = s.size();                // Size of the string to help modulate the shift amount.
        totalShift = (totalShift % n + n) % n; // Normalize totalShift within the bounds of the string length.

        // Perform the actual shift operation. The substring from (n - totalShift) to the end is moved to the front,
        // concatenated by the beginning of the string up to (n - totalShift).
        return s.substr(n - totalShift) + s.substr(0, n - totalShift);
    }
};
```

Typescript Solution

```
function stringShift(s: string, shifts: number[][]): string {
    // Holds the net number of shifts to apply.
    let totalShift: number = 0;

    // Loop through each shift operation in the array.
    for (let shiftOp of shifts) {
        let direction: number = shiftOp[0]; // Direction of shift: 0 for left, 1 for right
        let amount: number = shiftOp[1];    // Amount to shift

        // If the direction is 'left' (0), we convert amount to a negative value.
        if (direction === 0) {
            amount = -amount;
        }

        // Accumulate the effective shift amount.
        totalShift += amount;
    }

    let n: number = s.length; // Size of the string to help modulate the shift amount.
    // Normalize totalShift within the bounds of the string length.
    totalShift = (totalShift % n + n) % n;

    // Perform the actual shift operation. The substring from (n - totalShift) to the end is moved to the front,
    // concatenated by the beginning of the string up to (n - totalShift).
    return s.substring(n - totalShift) + s.substring(0, n - totalShift);
}
```

Time and Space Complexity

Time Complexity

The time complexity of the given code is $O(n + m)$, where n is the length of the string `s`, and m is the number of shifts in the `shift` list. This is because the code loops once over the list of shifts (m operations) and then applies a single string operation that has a complexity of $O(n)$.

Let's break it down:

- The `for` loop processes each element in the `shift` array of size m, which means it operates in $O(m)$ time.
- The modulo operation (`x %= len(s)`) is done in constant time ($O(1)$) since the length of the string is calculated just once.
- The string concatenation (`s[-x:] + s[:-x]`) is $O(n)$ because it involves creating two substrings and then concatenating them together, which requires building a new string of length n.

Therefore, when combined we get $O(n + m)$ for the time complexity of the entire function.

Space Complexity

The space complexity of the code is $O(n)$. This is due to the space needed for the output string in the final concatenation step. While `x` is calculated in place and does not need additional space proportional to the input, the slicing operation creates new strings which are then concatenated to form the final result, requiring a buffer that can hold the entire string `s`.

Here's why:

- The integer `x` is a single counter variable and thus takes up $O(1)$ space.
- The space required to store the input `s` and the shift instructions does not count towards the space complexity as they are considered input.
- The final line (`return s[-x:] + s[:-x]`) creates substrings and concatenates them, requiring $O(n)$ space since the resulting string is of the same length as the original string `s`.

In summary, the space complexity is $O(n)$.