2381. Shifting Letters II

Medium Array String Prefix Sum Leetcode Link

Problem Description

You are provided with a string `s` composed of lowercase English letters, and a list of shift operations represented by a 2D integer array `shifts`. Each shift operation is described by a triplet `[start, end, direction]`, and you are tasked with performing each of these operations on `s`.

When you perform a shift operation, you will be shifting every character in the string `s` starting at index `start` up to and including index `end`. If `direction` is `1`, each character will be shifted forward in the alphabet (cyclically, such that shifting 'z' forward wraps around to 'a'). Conversely, if `direction` is `0`, each character will be shifted backward in the alphabet (again cyclically, so shifting 'a' backward results in 'z').

The goal is to return the modified string after applying all the given shifts.

Intuition

To solve this problem, we need to efficiently apply each shift operation on the string without modifying the string character by character for each operation, which would lead to a large time complexity when many shifts are involved.

The solution approach utilizes the idea of a "difference array" to apply the shifts. The key insight is that shifting a range of characters forward or backward by a certain amount only affects the characters at the start and end boundaries of that range. Hence, we can increment or decrement the shift value at these boundaries and then later track the aggregated shift value for each character.

Here's the step-by-step breakdown of the process:

1. Initialize a "difference array" (`d`) with the same length as the string `s`, plus one extra element set to `0`.
2. Iterate over the `shifts` array:
   - Convert the direction `0` to `-1` for decrementing (backward shifting).
   - Increment the difference array at the `start` of the shift range by the shift value (which is `1` or `-1`).
   - Decrement the difference array at the element just after the `end` of the shift range to cancel the effect for subsequent characters beyond the shift range.
3. Process the difference array to convert it into an "aggregate shift array", where each element now represents the total number of shifts that character has gone through.
4. Iterate over the original string, applying the aggregate shift to each character:
   - Convert the character to its corresponding numerical value (`ord(s[i]) - ord('a')`).
   - Add the aggregate shift `d[i]`, wrap it with modulo 26 to keep the value within the alphabet range, and shift it relative to `ord('a')` to get the final character.
   - Join the transformed characters to form the final modified string.

By applying the difference array and only after accumulating all shift operations calculating the final characters, the algorithm optimizes the process and keeps the time complexity linear with respect to the length of the string and the number of shift operations.

Solution Approach

The solution to this problem involves a clever use of a technique called "Prefix Sum" or, in this specific case, a variation often referred to as the "difference array" approach. This allows us to perform multiple range update queries efficiently and then calculate the aggregate effect of all the updates. Here's a step-by-step explanation of how the solution is implemented:

1. **Initialization of the Difference Array (`d`)**: We start by creating a difference array `d` with the length of one more than the given string `s`. The extra element is to handle the case where the end of a shift range is the last character of the string. This array is initialized to `0` since initially, no shifts have been applied.

   ```
   d = [0] * (n + 1)
   ```

2. **Applying Shifts to the Difference Array**: The loop over the shifts array translates each shift operation into an efficient update on the difference array:

   ```
   for i, j, v in shifts:
       if v == 0:
           v = -1
       d[i] += v
       d[j + 1] -= v
   ```

   If `direction` is `0` (backward shift), it's converted to `-1`. The value `v` is added to the `start` index and subtracted from the index just after `end`. This setup means that when we later process the difference array, values from `start` to `end` get the proper shift effect applied.

3. **Accumulating Shift Values**: Next, by iterating over the difference array, we convert it from a range-effect array to an accumulation array where each element `d[i]` contains the total shift effect up to that point:

   ```
   for i in range(1, n + 1):
       d[i] += d[i - 1]
   ```

4. **Applying the Accumulated Shifts to the String**: Finally, we iterate through the original string and apply the accumulated shift effect to each character to obtain the final character after all shifts:

   ```
   return ''.join(
       chr(ord('a') + (ord(s[i]) - ord('a') + d[i] + 26) % 26) for i in range(n)
   )
   ```

   This is done by converting each character to its numerical equivalent (0 for 'a', 1 for 'b', etc.), adding the shift from `d[i]` (ensuring the result is still a valid character within the a-z range by using modulo 26), and converting it back to a character.

The use of a difference array allows us to apply complex range updates in `O(1)` time for each update and then apply the aggregate effect in `O(n)` time, giving the overall solution a linear time complexity relative to the size of the input. This makes the algorithm highly efficient for large strings and a large number of shift operations.

Example Walkthrough

Let's consider a small example to illustrate the solution approach step by step. Assume we have a string `s` which is "abcd", and two shift operations given by the 2D array `shifts`: `[[1, 2, 1], [1, 3, 0]]`.

The first operation shifts characters from index 1 to 2 ('b' and 'c') forward, and the second operation shifts characters from index 1 to 3 ('b', 'c', and 'd') backward.

1. **Initialization of the Difference Array (`d`)**: For string "abcd", we initialize the difference array `d` of length 5 (since "abcd" has length 4).

   ```
   d = [0, 0, 0, 0, 0]
   ```

2. **Applying Shifts to the Difference Array**:
   - Apply the first shift: `shifts[0] = [1, 2, 1]` which means to shift forward the characters at indices 1 and 2.

     ```
     d[1] += 1  # d = [0, 1, 0, 0, 0]
     d[3] -= 1  # d = [0, 1, 0, -1, 0]
     ```

   - Apply the second shift: `shifts[1] = [1, 3, 0]` which translates into shifting backward, hence `direction` is `0` and `v = -1`.

     ```
     d[1] -= 1  # d = [0, 0, 0, -1, 0]
     d[4] += 1  # d = [0, 0, 0, -1, 1]
     ```

3. **Accumulating Shift Values**: Iterate over `d` to accumulate shift values:

   ```
   # Starting from index 1, accumulate the shift values
   for i in range(1, len(d)):
       d[i] += d[i - 1]

   # d becomes [0, 0, 0, -1, 0]
   ```

4. **Applying the Accumulated Shifts to the String**: Iterate through the string, applying the accumulated shifts:

   ```
   # "a" does not change as d[0] is 0
   s[0] = 'a'

   # "b" would stay as 'b' because d[1] is 0
   s[1] = 'b'

   # "c" would also stay as 'c' because d[2] is 0
   s[2] = 'c'

   # "d" needs to be shifted backward once as d[3] is -1
   s[3] = chr(ord('a') + (ord('d') - ord('a') - 1 + 26) % 26)  # 'd' -> 'c'
   ```

The final modified string after applying all the given shifts is "abcc". The process demonstrates how the shifts are efficiently applied using the difference array approach.

Python Solution

```python
class Solution:
    def shiftingLetters(self, string: str, shifts: List[List[int]]) -> str:
        # Length of the input string
        length_of_string = len(string)

        # Initialize a difference array with an extra space for ease of calculation
        difference_array = [0] * (length_of_string + 1)

        # Apply each shift operation on the difference array
        for start, end, direction in shifts:
            # Convert the shift direction to -1 for left shift, and 1 for right shift
            value = 1 if direction == 1 else -1

            # Apply the shift direction at the start
            difference_array[start] += value

            # Reverse the shift direction at the end + 1 to nullify the effect past the end
            difference_array[end + 1] -= value

        # Convert the difference array to the prefix sum array representing actual shifts
        for i in range(1, length_of_string + 1):
            difference_array[i] += difference_array[i - 1]

        # Shift the characters of the original string according to the calculated shifts
        # and construct the resulting shifted string
        shifted_string = ''.join(
            # Shift each character using its ASCII value
            chr(
                # Ensure we start counting from 'a' and wrap around using modulo 26 (the alphabet size)
                ord('a') + (
                    # Find ASCII of current character and add the shift value
                    ord(string[i]) - ord('a') + difference_array[i] + 26
                ) % 26  # Mod to keep within alphabet range
            ) for i in range(length_of_string)
        )

        return shifted_string

```

Java Solution

```java
class Solution {
    public String shiftingLetters(String s, int[][] shifts) {
        int stringLength = s.length();
        // Difference array to hold the net shift values after performing all shift operations.
        int[] netShifts = new int[stringLength + 1];

        // Iterate over each shift operation and update the difference array accordingly.
        for (int[] shift : shifts) {
            int direction = (shift[2] == 0) ? -1 : 1;  // If the shift is left, make it negative.
            netShifts[shift[0]] += direction;          // Apply the shift to the start index.
            netShifts[shift[1] + 1] -= direction;      // Negate the shift after the end index.
        }

        // Apply the accumulated shifts to get the actual shift values.
        for (int i = 1; i <= stringLength; ++i) {
            netShifts[i] += netShifts[i - 1];
        }

        // Construct the result string after applying the shift to each character.
        StringBuilder resultStringBuilder = new StringBuilder();
        for (int i = 0; i < stringLength; ++i) {
            // Calculate the new character by shifting the current character accordingly.
            // The mod operation keeps the result within the range of the alphabet,
            // and the addition of 26 before mod ensures the number is positive.
            int shiftedIndex = (s.charAt(i) - 'a' + netShifts[i] % 26 + 26) % 26;
            resultStringBuilder.append((char) ('a' + shiftedIndex));
        }
        // Convert the StringBuilder to a String and return the result.
        return resultStringBuilder.toString();
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    string shiftingLetters(string str, vector<vector<int>>& shifts) {
        int length = str.size();
        vector<int> delta(length + 1); // Use 'delta' to represent the change in shift for each character

        // Process the shifts
        for (auto& shift : shifts) {
            // Additionally, adjust shift[2] to -1 for left shifts instead of 0
            if (shift[2] == 0) {
                shift[2] = -1;
            }
            delta[shift[0]] += shift[2]; // Apply the shift to the start index
            delta[shift[1] + 1] -= shift[2]; // Reverse the shift after the end index
        }

        // Accumulate the shifts to apply
        for (int i = 1; i <= length; ++i) {
            delta[i] += delta[i - 1];
        }

        // Construct the result string with the new shifts applied
        string result;
        for (int i = 0; i < length; ++i) {
            // Calculate new character by adding the shift (wrap around with modulo 26),
            // also ensure the result is non-negative with an additional +26 before modulo
            int newCharIndex = (str[i] - 'a' + delta[i] % 26 + 26) % 26;
            result += ('a' + newCharIndex);
        }
        return result;
    }
};

```

Typescript Solution

```typescript
function shiftingLetters(str: string, shifts: number[][]): string {
    const length: number = str.length;
    const delta: number[] = new Array(length + 1).fill(0); // Use 'delta' to represent the net shift for each character position af

    // Process the shifts
    for (const shift of shifts) {
        const start: number = shift[0];
        const end: number = shift[1];
        let direction: number = shift[2] === 0 ? -1 : 1; // Left shifts are negative; right shifts are positive

        delta[start] += direction; // Apply the shift at the start index
        delta[end + 1] -= direction; // Reverse the shift effect after the end index
    }

    // Apply the cumulative shifts to each character
    for (let i = 1; i <= length; i++) {
        delta[i] += delta[i - 1];
    }

    // Construct the final shifted string
    let result: string = '';
    for (let i = 0; i < length; i++) {
        // Calculate the new character index accounting for wrapping around the alphabet;
        // Ensure non-negative index with +26 before modulo 26
        const newCharIndex: number = (str.charCodeAt(i) - 'a'.charCodeAt(0) + delta[i] + 26) % 26;
        result += String.fromCharCode('a'.charCodeAt(0) + newCharIndex);
    }

    return result;
}

```

Time and Space Complexity

Time Complexity

The given code's time complexity can be analyzed as follows:

1. Initializing the difference array `d` with `n+1` zeroes takes `O(n)` time.
2. The shifts loop runs for each element in the `shifts` list (let's say there are `m` shift operations). For each shift operation, only constant-time operations are performed (two additions/subtractions), resulting in a time complexity of `O(m)` for this loop.
3. The loop for accumulating the shift values in the difference array `d` takes `O(n)` time as it iterates through the array once and performs constant-time operations at each index.
4. The final loop to construct the shifted string also runs in `O(n)` time, as it iterates through the string once and performs constant-time operations for each character (arithmetic and modulo operations).

The overall time complexity is the sum of all these steps: `O(n) + O(m) + O(n) + O(n)`, which can be simplified to `O(n + m)` since `n` and `m` could be of different sizes and each contributes linearly to the total runtime.

Space Complexity

1. The difference array `d` has a length of `n + 1`, giving a space complexity of `O(n)`.
2. The final string construction does not use additional space relative to the input size, except for the returned string, which is also of size `n`. However, as this is the output, it is typically not counted towards the space complexity.
3. No additional space is used that is dependent on the size of `shifts`.

Therefore, the space complexity is `O(n)`, determined by the size of the difference array `d`.