<https://leetcode.com/problems/shifting-letters-ii/description/>

You are given a string s of lowercase English letters and a 2D integer array shifts where shifts[i] = [starti, endi, directioni]. For every i, **shift** the characters in s from the index starti to the index endi (**inclusive**) forward if directioni = 1, or shift the characters backward if directioni = 0.

Shifting a character **forward** means replacing it with the **next** letter in the alphabet (wrapping around so that 'z' becomes 'a'). Similarly, shifting a character **backward** means replacing it with the **previous** letter in the alphabet (wrapping around so that 'a' becomes 'z').

Return **the final string after all such shifts to**s**are applied**.

**Example 1:**

**Input:** s = "abc", shifts = [[0,1,0],[1,2,1],[0,2,1]]

**Output:** "ace"

**Explanation:**

Firstly, shift the characters from index 0 to index 1 backward. Now s = "zac".

Secondly, shift the characters from index 1 to index 2 forward. Now s = "zbd".

Finally, shift the characters from index 0 to index 2 forward. Now s = "ace".

**Example 2:**

**Input:** s = "dztz", shifts = [[0,0,0],[1,1,1]]

**Output:** "catz"

**Explanation:**

Firstly, shift the characters from index 0 to index 0 backward. Now s = "cztz".

Finally, shift the characters from index 1 to index 1 forward. Now s = "catz".

**Constraints:**

* 1 <= s.length, shifts.length <= 5 \* 10^4
* shifts[i].length == 3
* 0 <= starti <= endi < s.length
* 0 <= directioni <= 1
* s consists of lowercase English letters.

**Attempt 1: 2023-12-23**

**Solution 1: Prefix Sum (60min)**

class Solution {

    public String shiftingLetters(String s, int[][] shifts) {

        int n = s.length();

        // The extra '+1' is to handle the case where the end of

        // a shift range is the last character of the string

        // Note: not like other Line Sweep problems no need '+1'

        // because other problems only forward direction, but

        // here also have backward direction, the last character

        // will also be a start

        int[] delta = new int[n + 1];

        for(int[] shift : shifts) {

            // direction = 1 -> shift forward

            if(shift[2] == 1) {

                delta[shift[0]]++;

                delta[shift[1] + 1]--;

            // direction = 0 -> shift backward

            } else {

                delta[shift[0]]--;

                delta[shift[1] + 1]++;

            }

        }

        int[] presum = new int[n + 1];

        presum[0] = delta[0];

        for(int i = 1; i <= n; i++) {

            presum[i] = presum[i - 1] + delta[i];

        }

        char[] chars = s.toCharArray();

        for(int i = 0; i < n; i++) {

            if(presum[i] < 0) {

                presum[i] = 26 - (-presum[i] % 26);

            }

            chars[i] = (char)((chars[i] - 'a' + presum[i] % 26) % 26 + 'a');

        }

        return new String(chars);

    }

}

Time Complexity: O(N)

Space Complexity: O(N)

**Step by Step**

Style 1: Add extra position and no need to check range < n or not

int[] delta = new int[n + 1];

for(int[] shift : shifts) {

    // direction = 1 -> shift forward

    if(shift[2] == 1) {

        delta[shift[0]]++;

        delta[shift[1] + 1]--;

    // direction = 0 -> shift backward

    } else {

        delta[shift[0]]--;

        delta[shift[1] + 1]++;

    }

}

e.g s = "abc", shifts = [[0,1,0],[1,2,1],[0,2,1]]

         1   0   0  -1   => [0,2,1]

         0   1   0  -1   => [1,2,1]

        -1   0   1   0   <= [0,1,0]

delta    a   b   c   \_   -> extra position

=> final 0   1   1  -2

presum   0   1   2   0

result   a  b+1 c+2      => "ace"

e.g s = "abc", shifts = [[0,1,0],[1,2,1],[0,2,0]]

        -1   0   0   1   => [0,2,0]

         0   1   0  -1   => [1,2,1]

        -1   0   1   0   <= [0,1,0]

delta    a   b   c   \_   -> extra position

=> final-2   1   1   0

presum  -2  -1   0   0

result  a-2 b-1  c      => "yac"

==============================================================

Style 2: If no extra position and we handle same way as check range < n or not as before

int[] delta = new int[n];

for(int[] shift : shifts) {

    // direction = 1 -> shift forward

    if(shift[2] == 1) {

        delta[shift[0]]++;

        if(shift[1] + 1 < n) {

            delta[shift[1] + 1]--;

        }

    // direction = 0 -> shift backward

    } else {

        delta[shift[0]]--;

        if(shift[1] + 1 < n) {

            delta[shift[1] + 1]++;

        }

    }

}

e.g s = "abc", shifts = [[0,1,0],[1,2,1],[0,2,1]]

         1   0   0   => [0,2,1] (2 + 1 >= n(=3)) -> ignore -1 on extra position

         0   1   0   => [1,2,1] (2 + 1 >= n(=3)) -> ignore -1 on extra position

        -1   0   1   <= [0,1,0]

delta    a   b   c   -> no extra position

=> final 0   1   1

presum   0   1   2

result   a  b+1 c+2  => "ace"

e.g s = "abc", shifts = [[0,1,0],[1,2,1],[0,2,0]]

        -1   0   0   => [0,2,0] (2 + 1 >= n(=3)) -> ignore 1 on extra position

         0   1   0   => [1,2,1] (2 + 1 >= n(=3)) -> ignore -1 on extra position

        -1   0   1   <= [0,1,0]

delta    a   b   c   -> no extra position

=> final-2   1   1

presum  -2  -1   0

result  a-2 b-1  c   => "yac"

==============================================================

Wrong Solution:

Compare with correction solution style 2, the problem is not on we must have

extra position, because even no extra position, we can handle with check < n

or not like we usually do, the problem is on when shift backward, we should

not turn delta calculation also from backward, it has to keep align with same

start / end position like forward, which means 'shift[0]' still for delta range

'start' position, 'shift[1] + 1' still for delta range 'end' position, NOT swap

as 'shift[1]' for delta range 'start' position, 'shift[0] - 1' for delta range

'end' position, the only change will be the delta value we add on 'start' and

'end' position, in forward, 'start' position delta '+1', 'end' position delta

'-1', in backward, 'start' position delta '-1', 'end' position delta '+1'

int[] delta = new int[n];

for(int[] shift : shifts) {

    // direction = 1 -> shift forward

    if(shift[2] == 1) {

        delta[shift[0]]++;

        if(shift[1] + 1 < n) {

            delta[shift[1] + 1]--;

        }

    // direction = 0 -> shift backward

    } else {

        // We swap 'shift[1]' for delta range 'start' position,

        // 'shift[0] - 1' for delta range 'end' position, it is

        // totally wrong, for below statement the effect is

        // on 'shift[0] - 1' we '+1' and on shift[1] we '-1'

        //

        // The correct solution:

        //

        //     shift[0] '+1'           shift[1] + 1 '-1'

        //       |--------------------|--|  [forward]

        //

        //     shift[0] '-1'           shift[1] + 1 '+1'

        //       |--------------------|--|  [backward]

        //

        // The wrong statement below will create:

        //

        //  shift[0] - 1 '+1'       shift[1] '-1'

        //    |--|--------------------|     [backward]

        //

        // Not only the index shift is wrong, but also the delta

        // value is wrong, in backward, we want original end '+1'

        // and original start '-1' but in wrong solution that is

        // still original end '-1' and original start '+1' same

        // as correction solution forward

        delta[shift[1]]--;

        if(shift[0] - 1 >= 0) {

            delta[shift[0] - 1]++;

        }

    }

}

e.g s = "abc", shifts = [[0,1,0],[1,2,1],[0,2,1]]

         1   0   0   => [0,2,1] (2 + 1 >= n(=3)) -> ignore -1 on extra position

         0   1   0   => [1,2,1] (2 + 1 >= n(=3)) -> ignore -1 on extra position

         0  -1   0   <= [0,1,0] (0 - 1 < 0) -> ignore 1 on ahead-extra position

delta    a   b   c   -> no extra/ahead-extra position

=> final 1   0   0

presum   1   1   1

result   a  b+1 c+1  => "acd" (wrong, expect "ace")

e.g s = "abc", shifts = [[0,1,0],[1,2,1],[0,2,0]]

         0   0  -1   => [0,2,0] (0 - 1 < 0) -> ignore 1 on ahead-extra position

         0   1   0   => [1,2,1] (2 + 1 >= n(=3)) -> ignore -1 on extra position

         0  -1   0   <= [0,1,0] (0 - 1 < 0) -> ignore 1 on ahead-extra position

delta    a   b   c   -> no extra position

=> final 0   0  -1

presum   0   0  -1

result   a   b  c-1   => "abb" (wrong, expect "yac")

==============================================================

**Refer to**

<https://algo.monster/liteproblems/2381>

**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.

1. 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.
2. 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]

**3.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]

**4.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] 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.

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();

    }

}

**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.