<https://leetcode.ca/all/370.html>

Assume you have an array of length ***n*** initialized with all **0**'s and are given ***k*** update operations.

Each operation is represented as a triplet: **[startIndex, endIndex, inc]** which increments each element of subarray **A[startIndex ... endIndex]** (startIndex and endIndex inclusive) with **inc**.

Return the modified array after all ***k*** operations were executed.

Example:

Input: length = 5, updates = [[1,3,2],[2,4,3],[0,2,-2]]

Output: [-2,0,3,5,3]

**Explanation:**

Initial state:

[0,0,0,0,0]

After applying operation [1,3,2]:

[0,2,2,2,0]

After applying operation [2,4,3]:

[0,2,5,5,3]

After applying operation [0,2,-2]:

[-2,0,3,5,3]

**Attempt 1: 2023-12-03**

**Solution 1: Sweep Line (10 min)**

class Solution {

public int[] getModifiedArray(int length, int[][] updates) {

int[] result = new int[length];

for(int[] update : updates) {

int start = update[0];

int end = update[1];

int increment = update[2];

result[start] += increment;

if(end + 1 < length) {

result[end + 1] -= increment;

}

}

for(int i = 1; i < length; i++) {

result[i] += result[i - 1];

}

return result;

}

}

Time Complexity: O(N)

Space Compleixty: O(N)

**Refer to**

<https://algo.monster/liteproblems/370>

## Problem Description

In this problem, we are given an integer length which refers to the length of an array initially filled with zeros. We are also given an array of update operations called updates. Each update operation is described as a tuple or list with three integers: [startIdx, endIdx, inc]. For each update operation, we are supposed to add the value inc to each element of the array starting at index startIdx up to and including the index endIdx.

The goal is to apply all the update operations to the array and then return the modified array.

For instance, if length = 5 and an update action specifies [1, 3, 2], then after this update, the array will have 2 added to its 1st, 2nd, and 3rd positions (keeping zero-based indexing in mind), resulting in the array [0, 2, 2, 2, 0] after this single operation.

After applying all updates, we need to return the final state of the array.

## Intuition

The intuitive brute force approach would be to go through each update and add inc to all elements ranging from startIdx to endIdx for every update in updates. However, this would be time-consuming, especially for a large number of updates or a large range within the updates.

This is where the prefix sum technique comes into play. It is a very efficient way for handling operations that involve adding some value to a range of elements in an array. The main intuition behind prefix sums is that we can record changes at the borders – at the start index, we begin to add the increment, and just after the end index, we cancel it out.

In more detail, for each update [startIdx, endIdx, inc], we add inc to the position startIdx and subtract inc from endIdx + 1. This marks the range where the increment is valid. When we compute the prefix sum of this array, it will apply the inc increment to all elements since startIdx, and the subtraction at endIdx + 1 will counterbalance it, returning the array to its original state beyond endIdx.

The accumulation step goes through the array and adds each element to the sum of all previous elements, effectively applying the increments and decrements outlined in the updates.

In the presented solution, the Python accumulate function from the itertools module takes care of the accumulation step for us, summing up the differences and giving us the array after all updates have been applied.

## Solution Approach

The implementation of this solution is straightforward once we understand the intuition behind using prefix sums. Here’s a step-by-step rundown of the algorithm:

* We initialize an array d with a length of length filled with zeros. This array will serve as our difference array which records the difference of each position compared to the previous one.
* We then iterate over each update in the updates array. Each update is in the format [startIdx, endIdx, inc].
* For each update, we add inc to d[startIdx]. This signifies that from startIdx onwards, we have an increment of inc to be applied.
* We then check if endIdx + 1 < length, which is to ensure we do not go out of bounds of the array. If we are still within bounds, we subtract inc from d[endIdx + 1]. This effectively cancels out the previous increment beyond the endIdx.
* After processing all updates, d now contains all the changes that are needed to be applied in the form of a difference array.
* Finally, we use the accumulate function of Python to calculate the prefix sum array from the difference array d. This step goes through the array adding each element to the sum of all the previous elements and thus applies the increments tracked in d at their respective starting indices and cancels them after their respective ending indices.
* The returned value from the accumulate function gives us the modified array arr after all updates have been applied, and this is returned as the final result.

This approach effectively reduces the time complexity of the problem, as we only need to make a constant-time update for each range increment, rather than incrementing all elements within the range for each update which could lead to a much higher time complexity.

Here's the mentioned code for the described solution approach that uses these steps:

from itertools import accumulate

class Solution:

def getModifiedArray(self, length: int, updates: List[List[int]]) -> List[int]:

d = [0] \* length

for l, r, c in updates:

d[l] += c

if r + 1 < length:

d[r + 1] -= c

return list(accumulate(d))

In this code example, accumulate is an in-built Python function from the itertools module that computes the cumulative sum of the elements. This effectively does the last step of our prefix sum implementation for us.

### Example Walkthrough

Let's illustrate the solution approach with a small example.

Consider an array of length = 5 which is initially [0, 0, 0, 0, 0]. Suppose we have the following updates = [[1, 3, 2], [2, 4, 3]] which includes two update operations.

* Initialize the difference array d which is the same size as our initial array: d = [0, 0, 0, 0, 0]
* Apply the first update [1, 3, 2]:

1. Add the increment 2 to d[startIdx] which is d[1], so now d = [0, 2, 0, 0, 0].
2. Subtract the increment 2 from d[endIdx + 1] which is d[4], but d[4] is out of bounds for the first update's end index, so d remains unchanged here.

* Apply the second update [2, 4, 3]:

1. Add the increment 3 to d[startIdx] which is d[2], now d = [0, 2, 3, 0, 0].
2. Subtract the increment 3 from d[endIdx + 1] which is not applicable here as endIdx + 1 equals 5 which is out of bounds, hence no subtraction is done.

* With all updates applied, the difference array d is: d = [0, 2, 3, 0, 0]
* Now use the accumulate function to calculate the prefix sum array from the difference array d: Final array = list(accumulate(d)) = [0, 2, 5, 5, 5]

The final array after applying all updates will be [0, 2, 5, 5, 5].

This example confirms that the prefix sum technique updates the initial zero-filled array efficiently with the given range update operations.

class Solution {

// Method to compute the modified array after a sequence of updates

public int[] getModifiedArray(int length, int[][] updates) {

// Create an array 'difference' initialized to zero, with the given length

int[] difference = new int[length];

// Apply each update in the updates array

for (int[] update : updates) {

int startIndex = update[0]; // Start index for the update

int endIndex = update[1]; // End index for the update

int increment = update[2]; // Value to add to the subarray

// Apply increment to the start index

difference[startIndex] += increment;

// If the end index is not the last element,

// apply the negation of increment to the element after the end index

if (endIndex + 1 < length) {

difference[endIndex + 1] -= increment;

}

}

// Convert the 'difference' array into the actual array 'result'

// where each element is the cumulative sum from start to that index

for (int i = 1; i < length; i++) {

difference[i] += difference[i - 1];

}

// Return the resultant modified array

return difference;

}

}