370. Range Addition

July 18, 2016 | 11.1K views

Average Rating: 4.59 (22 votes)

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]
```

Solution

Approach 1: Naïve Approach

Algorithm

The algorithm is trivial. For each update query, we iterate over the required update range and update each element individually.

Each query of `updates` is a tuple of 3 integers: $start, end$ (the start and end indexes for the update range) and val (the amount by which each array element in this range must be incremented).

C++ Copy

```cpp
vector<int> getModifiedArray(int length, vector<vector<int> > updates)
{
    vector<int> result(length, 0);

    for (auto& tuple : updates) {
        int start = tuple[0], end = tuple[1], val = tuple[2];

        for (int i = start; i <= end; i++) {
            result[i] += val;
        }
    }

    return result;
}
```

Complexity Analysis

- Time complexity : $O(n \cdot k)$ (worst case) where k is the number of update queries and n is the length of the array. Each of the k update operations take up $O(n)$ time (worst case, when all updates are over the entire range).
- Space complexity : $O(1)$. No extra space required.

Approach 2: Range Caching

Intuition

- There is only one read query on the entire range, and it occurs at the end of all update queries. Additionally, the order of processing update queries is irrelevant.
- Cumulative sums or `partial_sum` operations apply the effects of past elements to the future elements in the sequence.

Algorithm

The algorithm makes use of the above intuition to simply store changes at the *borders* of the update ranges (instead of processing the entire range). Finally a single post processing operation is carried out over the entire output array.

The two steps that are required are as follows:

1. For each update query $(start, end, val)$ on the array arr, we need to do *only* two operations:
   - Update $start$ boundary of the range:
   $$arr_{start} = arr_{start} + val$$
   - Update just beyond the end boundary of the range:
   $$arr_{end+1} = arr_{end+1} - val$$
2. Final Transformation. The cumulative sum of the entire array is taken (`0` - based indexing)
$$arr_i = arr_i + arr_{i-1} \quad \forall \quad i \in [1, n)$$

C++ Copy

```cpp
vector<int> getModifiedArray(int length, vector<vector<int> > updates)
{
    vector<int> result(length, 0);

    for (auto& tuple : updates) {
        int start = tuple[0], end = tuple[1], val = tuple[2];

        result[start] += val;
        if (end < length - 1)
            result[end + 1] -= val;
    }

    // partial_sum applies the following operation (by default) for the parameters {x[0], x[n], y[0]}:
    // y[0] = x[0]
    // y[1] = y[0] + x[1]
    // y[2] = y[1] + x[2]
    // ...  ...  ...
    // y[n] = y[n-1] + x[n]

    partial_sum(result.begin(), result.end(), result.begin());

    return result;
}
```

Formal Explanation

For each update query $(start, end, val)$ on the array arr, the goal is to achieve the result:

$$arr_i = arr_i + val \quad \forall \quad i \in [start, end]$$

Applying the final transformation, ensures two things:

1. It carries over the $+val$ increment over to every element $arr_i \; \forall \; i \geq start$.
2. It carries over the $-val$ increment (equivalently, a $+val$ decrement) over to every element $arr_j \; \forall \; j > end$.

The net result is that:

$$\begin{aligned} arr_i &= arr_i + val & \forall \quad i \in [start, end] \\ arr_j &= arr_j + val - val = arr_j & \forall \quad i \in (end, length) \end{aligned}$$

which meets our end goal. It is easy to see that the updates over a range did not carry over beyond it due to the compensating effect of the $-val$ increment over the $+val$ increment.

It is good to note that this works for multiple update queries because the particular binary operations here (namely addition and subtraction):

- are *closed* over the entire domain of `Integer`s. (A counter example is division which is not closed over all `Integer`s).
- are *complementary* operations. (As a counter example multiplication and division are not always complimentary due to possible loss of precision when dividing `Integer`s).

Complexity Analysis

- Time complexity : $O(n + k)$. Each of the k update operations is done in constant $O(1)$ time. The final cumulative sum transformation takes $O(n)$ time always.
- Space complexity : $O(1)$. No extra space required.

Further Thoughts

An extension of this problem is to apply such updates on an array where all elements are **not** the same.

In this case, the second approach requires that the original configuration must be stored separately before applying the final transformation. This incurs an additional space complexity of $O(n)$.

@StefanPochmann suggested another method (see comment section) which does not require extra space, but requires an extra linear pass over the entire array. The idea is to apply *reverse* `partial_sum` operation on the array (for example, array $[2, 3, 10, 5]$ transforms to $[2, 1, 7, -5]$) as an initialization step and then proceed with the second method as usual.

Another general, more complex version of this problem comprises of multiple read and update queries over *ranges*. Such problems can be solved quite efficiently with Segment Trees by applying a popular trick called **Lazy Propogation**.

Analysis written by @babhishek21.

StefanPochmann 46873 July 18, 2016 6:38 PM

> An extension of this problem is to apply such updates on an array where all elements are not the same. In this case, the second approach **requires that the original configuration must be stored separately** before applying the final transformation. This incurs an additional space complexity of O(n).

Why? Can't you just apply "*reverse partial_sum*" to initialize? For example if given the array [2, 3

user760 8 November 20, 2018 1:56 AM

The "net result" equation in Formal Explanation section doesn't display correctly

ajDeViL 2 July 7, 2019 12:14 AM

So good. So neat. Hints are pretty useful. I am not saying because this I got the elegant solution.

its_dark 551 March 6, 2019 7:25 PM

Hi,

Can you please provide more explanation on the reverse_partial_sum method?

venkata-sai-krishn 19 July 20, 2017 12:22 AM

Yes, it works, and that is a brilliant thought! Thanks @StefanPochmann

kkzeng 174 March 27, 2020 12:54 PM

Beautiful solution!