Array Manipulation

Isaac DeJager

March 15, 2023

Click here to view the HackerRank page

1 Problem Description

Starting with a 1-indexed array of zeros and a list of operations, for each operation add a value to each array element between two given indices, inclusive. Once all operations have been performed, return the maximum value in the array.

Example
$n = 10$
$queries = [[1, 5, 3], [4, 8, 7], [6, 9, 1]]$
Queries are interpreted as follows:

a b k
1 5 3
4 8 7
6 9 1

Add the values of k between the indices a and b inclusive:

index → 1 2 3 4 5 6 7 8 9 10
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[3, 3, 3, 3, 3, 0, 0, 0, 0, 0]
[3, 3, 3, 10, 10, 7, 7, 7, 0, 0]
[3, 3, 3, 10, 10, 8, 8, 8, 1, 0]

The largest value is 10 after all operations are performed

Function Description
Complete the function arrayManipulation in the editor below.
arrayManipulation has the following parameters:

- int n - the number of elements in the array
- int $queries[q][3]$ - a two dimensional array of queries where each $queries[i]$ contains 3 integers, a, b, and k

Returns

- int - the maximum value in the resultant array

Input Format

The first line contains two space-separated integers n and m, the size of the array and the number of operations.

Each of the next m lines contains three space-separated integers a, b, and k, the left index, right index and summand.

Constraints

- $3 \leq n \leq 10^7$
- $1 \leq m \leq 2 * 10^5$
- $1 \leq a \leq b \leq n$
- $0 \leq k \leq 10^9$

Sample Input

5 3
1 2 100
2 5 100
3 4 100

Sample Output

200

Explanation

After the first operation the list is 100 100 0 0 0
After the second operation the list is 100 200 100 100 100
After the third operation the list is 100 200 200 200 100
The maximum value is 200

2 Solution

```
long arrayManipulation(int n, vector<vector<int>> queries) {

    vector<int> arr(n, 0);

    //Create the difference array
    for (int i = 0; i < queries.size(); i++) {
        arr[queries[i][0] - 1] += queries[i][2];
        arr[queries[i][1]] -= queries[i][2];
    }

    long sum = arr[0];
    long max = sum;

    //Search the difference array for the largest value
    for (int i = 1; i < n; i++) {
        sum += arr[i];
        if (sum > max) {
            max = sum;
        }
    }

    return max;

}
```

3 Analysis

The streamline solution to this is explained the problem description. Create a new array fully initialized at 0, then loop through each operation, perform them on the new array, and then find the largest value in the result. While this solution yields the correct answer, it is far from efficient.

Consider creating an array of size n with m operations where each operation has $a = 1$ and $b = n$. This means that will take:

$$O(m * n) = O(n^2)$$

time to yield a result, since for every operation in *queries*, you have to add k n times. There is a far more efficient solution.

Instead of drafting a solution where at each operation you add k to every index between a and b, instead we create what is called a difference array and only adjust the values at indexes a and b. A difference array simply means that each index i equals the difference between $arr[i]$ and $arr[i-1]$. For example, suppose you have the array:

$$arr = [1, 4, 4, 3, 5, 5, 5, 9]$$

The corresponding difference array is:

$$dArr = [1, 3, 0, -1, 2, 0, 0, 4]$$

So, the first step in the solution is to create a difference array based on the queries. We start by creating a vector arr instead of a simple integer array to avoid a segmentation fault. We fully initialize the vector with 0s and then enter a for loop where we loop through every row in *queries*.

Recall $queries[i][0] = a$ is the starting index (counting start at 1), $queries[i][1] = b$ is the ending index and $queries[i][2] = k$ is the value to be added.

So, we add k to $arr[i][a-1]$ and subtract k from $arr[i][b]$. Since all indices between $a-1$ and b also have k added to it, for our difference array we do not need to touch them because the difference between $arr[a-1]$, $arr[a]$, $arr[a+1]$, ... $arr[b-2]$, and $arr[b-1]$ is all the same as it was before since k was added to all of them. And k is subtracted from $arr[b]$ since b is the first index that k is not added to. And all indices after b will also have the same difference as before.

After looping through *queries*, we have finished our difference array. For every index i, $arr[i]$ equals the difference between indices i and $i-1$ in the actual array. And now, to find the maximum value in the actual array is very simple. We just loop through our difference array and sum up all the values, keeping track of the highest sum. This is because for all i:

$$\sum_{n=1}^{i} dArr[i] = arr[i]$$

.

We have a difference array. That means every time we add the next value what we are really calculating is the actual value in the resultant array. We basically create the requested array using the difference array, but instead of saving every value we only keep track of the highest because that is all we need to return. This procedure is done in $O(n)$ time.

The time complexity of this algorithm is far simpler:

$$O(3 * m) + O(n) = 2 * O(n) = O(n)$$

.

The space complexity does require a vector of size n, so:

$$O(n)$$

One final note on this solution is the use of longs instead of integers. Even though the difference array works fine as each index being an integer, once we start summing up each element to find the largest value in the resultant array, the numbers become to high for integers to handle. So, to ensure the data types can handle the potentially very high values, we use longs as our variables when calculating the sum.