RANGE UPDATES

Consider an array C of n integers, initially all equal to zero. We want to support the following operations:

- update(i, j, c), where $0 \leq i \leq j \leq n \leq 1$ and c is an integer: it changes C such that $C[k] := C[k] + c$ for every $i \leq k \leq j$.
- query(i), where $0 \leq i \leq n \leq 1$: it returns the values of$C[i]$.
- sum(i, j), where $0 \leq i \leq j \leq n \leq 1$: it returns $\sum_{k=i}^{j} C[k]$

Design a data structure that uses O(n) space and implements each operation above in O(log n) time. Note that query(i) = sum(i, i) but it helps to reason. [Hint to further save space: use an implicit tree such as the Fenwick tree (see wikipedia).]

SOLUTION
The solution can be implemented using a Segment tree or a Fenwick Tree. Both allowed the aforementioned operation in O(log n). Here we will present a solution that use Fenwick Trees also called Binary Indexed Tree (BIT).
Let's start with some considerations:

- integer can be represented as sum of powers of two. Therefore, I can represent the sum operation as sum of sets of sub-sums.
- each index, if it is a power of 2, will store the sum of all elements before that, and we will apply this repetitively so as to get what each index will store. Let's see an example:
  Suppose, we have an array of 16 elements, $[1 \dots 16]$. Powers of 2 are the index: 1, 2, 4, 8, 16. These index will store sum of all elements before them. Now, we divide this array in two halves: we get $[1 \dots 8]$ and $[9 \dots 16]$, and so on.
  Let's have the following array:

$$\begin{aligned} A &= [1, 2, 3, 4, 5, 6, 7, 8] \\ BIT &= [0, 0, 0, 0, 0, 0, 0, 0] \end{aligned} \tag{1}$$

  Let take the power 2 index (i.e. 1,2,4,8), and we store in BIT the value of the sum of the previous element. Then we have:

$$BIT = [1, 3, 0, 10, 0, 0, 0, 36]$$

  Now we divide the array in two: $A_1 = [1, 2, 3, 4]$ and $A_2 = [5, 6, 7, 8]$. Then we take the power 2 index of A_1 and A_2 (i.e. 1,2,4), and we update the value of BIT where there index that are not updated yet. Then we have:

$$BIT = [1, 3, 0, 10, 5, 11, 0, 36]$$

  Following the same procedure, the value doesn't change at any index if it has been already filled. Then we will the following.

$$BIT = [1, 3, 3, 10, 5, 11, 7, 36]$$

  If we consider our array as a binary tree, and we change the value of each node by adding the sum of nodes in its left sub-tree, we will obtain the same result.

- Now let's see how to implement sum(i, j). The idea is to keep a variable *ans* initialized to 0. Follow the path from *root* to the *index* node. Whenever we need to follow a right link, add the value of current node to *ans*, and once we reach the node with the searched index we add that value too. To get the sum of elements in range i to j, we get the sum from 0 to j and we subtract the sum from 0 to (i-1).

- Now let's see how to implement update(i, j, c). For now we focus on just the update of a single index. If we want to increment the value at index k by say c. Follow the path from root to the index node k. Whenever we need to follow a left link, add the value of c to current node. Once we reach the node, add c to that node too. This is because we will need to update the set of nodes in the tree that include that node in its left sub-tree, so that it will be consistent with our sum operation.

To implement this kind of idea we need to exploit some properties of the binary numbers. Since for the sum function we need the right-path we exploit the following fact: given an index if we reset the right-most SET-bit we will go up to the least node that took a RIGHT path. For example:

$5 \longrightarrow 101 \longrightarrow 100 \longrightarrow 4$

$7 \longrightarrow 111 \longrightarrow 110(6) \longrightarrow 100 \longrightarrow 4$

$3 \longrightarrow 011 \longrightarrow 010 \longrightarrow 2$

To implement this concept in to an algorithm we use the following formula $i-(i\ AND(-i))$, where i is the current index and $(-i)$ mean the 2's complement of i.

function GETSUM($index$)
  $ans \leftarrow 0$
  while $index\ != 0$ **do**
    $ans \leftarrow ans + BIT[index]$
    $index \leftarrow index - (index\ AND(-index))$
  end while
  return ans
end function

Now we should try to do the same for the updating function. Now given an index we should search the least node that took the LEFT path. Here, instead of stripping off the least-significant 1 bit (i.e. subtracting), we now add it on at each stage to get the next entry to adjust. Therefore we simply have:

function UPDATE($index$,val)
  while $index\ \leq length\_of\_array$ **do**
    $BIT[index] \leftarrow BIT[index] + val$
    $index \leftarrow index + (index\ AND(-index))$
  end while
  return ans
end function

Now to get the final functions we do the following:

- update(i, j, c) is equivalent to update(i, c) and update(j + 1, -c). Indeed update(p, v) will affect all the index $p' \geq p$. To limit the effect to a given range [i,j], we subtract -v from all the index $p' > j$ by performing the operation update(j+1, -v).
- query(i)= getSum(i)-getSum(i-1)
- sum(i, j)=getSum(j)-getSum(i-1)

Both getSum and update work in O(log n), where n is the length of the input. This because: the tree is always balanced and the two function they simply take a path in the tree. Furthermore, this algorithm take just O(n) space since it's just storing a flat array.