Lazy propagation in Segment Tree

Bharat Kul Ratan Follow
Oct 15, 2017 · 5 min read

In the last last tutorial we learned how to build a Segment tree, query it and update point values to it. In this tutorial we will learn a useful technique **Lazy Propagation** for Segment trees. Using lazy propagation we can update a range in the tree efficiently. In this technique we update the value in a lazy manner, as the name suggests itself. Being lazy, values are not updated unless needed. Hence "lazy propagation" roughly translates to propagating values from root in a lazy manner.

Let's make it clear with an example. Suppose we've built a Segment tree and there are two types of queries available on the tree. First one is update, which adds some value to nodes in a particular range. Second type is query for sum, which asks for sum of node values for a given range. In lazy propagation, "Lazy" refers to the idea that we will not update the range down to the leaf nodes every time it is called. Because the query for sum is called in the last, we will accumulate the values in an another array every time update() is called and these accumulated values will provide sum only for query which asks for sum in a range. Let's take an example to make the concept clear.

Question
You are given an array of N elements, which are initially all 0. After that you will be given C commands. They are:
* 0 p q v — you have to add v to all numbers in the range of p to q (inclusive), where p and q are two indexes of the array.

* 1 p q — output a line containing a single integer which is the sum of all the array elements between p and q (inclusive)

Input:
In the first line you'll be given T, number of test cases.

Each test case will start with N (N<=100 000) and C (C<=100 000). After that you'll be given C commands in the format as mentioned above. 1 <= p,q <= N and 1 <= v <= 10^7.

Output:

Print the answers of the queries.

Source of the question : SPOJ — HORRIBLE

```
import java.io.BufferedReader;
import java.io.IOException;
import java.util.Arrays;
import java.io.InputStreamReader;
import java.util.StringTokenizer;
import java.io.FileInputStream;

class lazypropagation {
 long tree[];  //the Segment tree
 long helper[]; //this array accumulates value to be updated
 //if true, values are pending for this node to be updated
 boolean flag[];

 lazypropagation(){
  tree = new long [262145];
        helper = new long [262145];
        flag  = new boolean [262145];

        //initialize all node of tree and helper with zero
  Arrays.fill (tree, 0);
        Arrays.fill (helper, 0);

        //initially set all flags to false as their is no pending
value for any node
        Arrays.fill (flag, false);
 }


 /**
  * @param nodeToUpdate node in tree to be updated
  * @param valueToUpdate value to be updated
  * @param left is beginning index of array
  * @param right is ending index of array
  * @param start is beginning index of query
  * @param end is ending index of query
  **/
 private void update (int node, int value, int left, int right, int
start, int end){
  /**
```

```
     *   Legends:
     *   S = Start, E = End, L = Left, R = Right
     *   dotted lines ---- represents array
     *   square brackets [ ] denote range of query
     *
     **/

    // When range to be updated does not intersect
    // with the array indices [left, right]
    // ...E]  L------R [S...
     if ( end <left || start > right )
              return ;

         // left == right means we've reached to a single element of
array.
         if (left == right ){
             tree[node]+= value;

             //flag is true mean some value is pending in helper[node]
             //so we should add this to tree[node] and clear
helper[node]
             if (flag[node]==true){
                 tree[node] += helper[node];
                 helper[node]=0;
             }
             return;
         }

         int leftChild = 2*node;
         int rightChild = 2*node + 1;

         //Each of the below case finds a range by right-left+1 or
end-left+1
         //this range should be multiplied by value and should be
added to tree[node]

         //    [S   L------R E]
         if (start <= left && right <= end){
             helper[leftChild] += value;
             helper[rightChild] += value;

             flag[leftChild] = flag[rightChild] = true;

             tree[node] += (value *(right - left +1));
             return ;
         }
         //  ---[S---R   E]
         else if (start <= right &&  right < end)
             tree[node ] += (value*(right - start +1));

         //  [S    L----E]----
         else if (start < left && left <= end )
             tree[node] += (value*(end - left + 1));
```

```
        //  ----L----[S-----E]-----R-----
        else if (left <= start && end <= right)
            tree[node] += (value*(end - start + 1));

        update (leftChild, value, left, (left+right)/2, start, end );
        update (rightChild, value, ((left+right)/2)+1, right, start,
end);
 }

 private long sum_of_range (int node, int left, int right, int start,
int end){
  // query does not overlaps with array range
  // ---R---- [S  E]----L-----
        if (right < start || end < left)
            return 0;

        int leftChild = 2*node;
  int rightChild = 2*node+1;

  // true means value updates are pending for the node
        if (flag[node] == true){

   //update value in tree node
   // right-left+1 gives the range for which
   //values to be updated in parent node which here is tree[node]
            tree[node] += (helper[node] *(right-left+1));

            //after values are updated, set flag to false
            flag[node]=false;

            // if it's not a leaf node
            if (left != right){

    //push the flag downward to children of node because
    //after recursive call these children will act as parent
    //and then tree[node] would be calculated
                flag [ leftChild ] = flag [ rightChild ] = true;

                //push down the parent value to leftChild and
rightChild
                //hence updated values for child nodes would be the
sum of their own values + parent node value
                helper[ leftChild ] += helper[node];
                helper[ rightChild ] += helper[node];
            }

            //as value of helper[node] is already added to its
children
            // set helper[node] = 0 to receive fresh values
            helper[node]=0;
        }

  // return value of tree[node] as query completely overlaps the
array range
```

```
      // hence no need to go further on its children nodes.
      // [S---L---R---E]
            if (start <= left && right <= end){
                return tree[node];
            }

            //sum returned for left half of tree intersecting with query
            long s1 = sum_of_range(leftChild, left, (left+right)/2,
    start, end);

            //sum returned for right half of tree intersecting with query
            long s2 = sum_of_range(rightChild, ((left+right)/2)+1, right,
    start, end);
            return s1+s2;
        }

     public static void main (String [] args) throws IOException{
      // for reading input from console use new InputStreamReader
    (System.in)
            BufferedReader br = new BufferedReader (new InputStreamReader
    (new FileInputStream("input.txt")));
            StringTokenizer st =null;
            int t = Integer.parseInt(br.readLine());
            for(;t>0;t--){
                st = new StringTokenizer (br.readLine());
                int N = Integer.parseInt(st.nextToken());
                int C = Integer.parseInt(st.nextToken());
                lazypropagation segtree = new lazypropagation();
                for (int I=1;I<=C;I++){
                    st = new StringTokenizer(br.readLine());
                    if (st.nextToken().equals("0")){
                        int P = Integer.parseInt(st.nextToken());
                        int Q = Integer.parseInt(st.nextToken());
                        int V = Integer.parseInt(st.nextToken());
                        segtree.update (1, V, 1, N, P,Q);
                    }
                    else {
                        int P = Integer.parseInt(st.nextToken());
                        int Q = Integer.parseInt(st.nextToken());
                        long ans = segtree.sum_of_range (1,1,N, P,Q);
                        System.out.println(ans);
                    }
                }
            }
        }
    }
```

In the above code, update() is responsible for both building the tree and updating the values in it. In update(), we took array indices left to 1 and right to N as N is the number of elements in array. There are four possible cases regarding intersection of array range and query range. All those cases are represented in the comments of update(). Basic idea of update() is to accumulate the value to be updated for those node which are the intersection of query range and array range in helper[node]. These values will be accumulated for each update() and will be processed when sum_of_range() is called. I've put a lot of comments in the code to make it self-explanatory. Still for any doubts please drop a comment. Thanks for reading. Have a nice day!!!

Programming Data Structures Tutorial Java