Segment Tree Algorithm

Project Team

- Mohamed Saeed El-Naggar
- Mohamed Refaat Bayoumi
- Mohamed Hassan Imam
- Mohamed Bahaa El-Din
- Mohamed Khaled Mohamed
- Mohamed Ebrahem Fatoh

Problem Definition

Let us consider the following problem to understand Segment Trees.

We have an array arr[0 . . . n-1]. We should be able to

1 → Find the sum of elements from index l to r where 0 <= l <= r <= n-1

2 → Change value of a specified element of the array to a new value x. We need to do arr[i] = x where 0 <= i <= n-1.

A **simple solution**

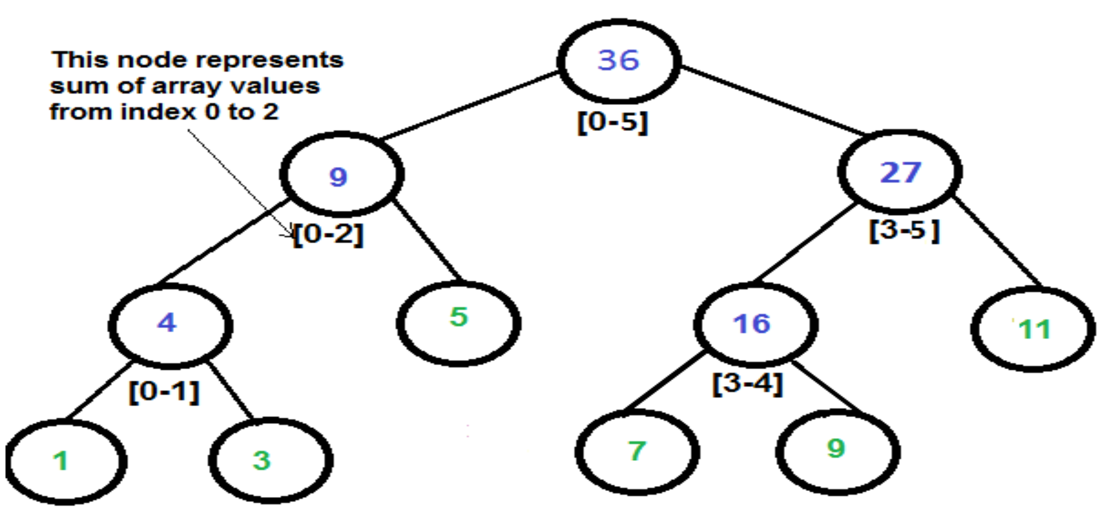
is to run a loop from l to r and calculate sum of elements in given range. To update a value, simply do arr[i] = x. The first operation takes O(n) time and second operation takes O(1) time.

Representation of SegmentTree

1. Leaf Nodes are the elements of the input array.

2. Each internal node represents some merging of the leaf nodes. The merging may be different for different problems. For this problem, merging is sum of leaves under a node.An array representation of tree is used to represent Segment Trees. For each node at index i, the left child is at index 2*i+1, right child at 2*i+2 and the parent

is at $\lfloor (i-1)/2 \rfloor$.



Segment Tree for input array {1, 3, 5, 7, 9, 11}

Construction of Segment Tree from given Array

We start with a segment arr[0 . . . n-1]. and every time we divide the current segment into two halves(if it has not yet become a segment of length 1), and then call the same procedure on both halves, and for each such segment we store the sum in corresponding node.

All levels of the constructed segment tree will be completely filled except the last level. Also, the tree will be a Full Binary Tree because we always divide segments in two halves at every level. Since the constructed tree is always full binary tree with n leaves, there will be n-1 internal nodes. So total number of nodes will be 2*n – 1.

Height of the segment tree will be $\lceil \log_2 n \rceil$. Since the tree is represented using array and relation between parent and child indexes must be maintained, size of memory allocated for segment tree will be $2 * 2^{\lceil \log_2 n \rceil} - 1$

Query for Sum of given Range

Once the tree is constructed, how to get the sum using the constructed segment tree. Following is algorithm to get the sum of elements.

int getMax(node, l, r) {

if range of node is within l and r return value in node

else if range of node is completely outside l and r return 0

elsereturn max(getSum(node's left child, l, r) , getSum(node's right child, l, r))

}

Update Value

Like tree construction and query operations, update can also be done recursively. We are given an index which needs to updated. Let *diff* be the value to be added. We start from root of the segment tree, and add *diff* to all nodes which have given index in their range. If a node doesn't have given index in its range, we don't make any changes to that node.

Implementation Of Segment Tree (Classical + Lazy Propagation) C++

```
#include <iostream>

#include <bits/stdc++.h>


// Segment Tree : Solve Problems (Min , Max , Sum)

using namespace std ;


int A[1001] , NewA[4 * 1001] , Lazy[4 * 1001] ;

/*

 *  A : Ordinary Array That User Enter Value into

 *  NewA : Segment Tree Array That Will Be Built

 */


int getLeft(int i){ return (i << 1) ; } // get LeftChild in The
Tree

int getRight(int i){return (i << 1) + 1 ;} // get Right child in
the Tree
```

```
void BuiltTree(int l , int r , int p = 1){

    if(l == r) NewA[p] = A[l] ;

    else {

        int mid = l + (r - l) / 2 ;

        BuiltTree(l , mid , getLeft(p)) ; // go to LeftChild

        BuiltTree(mid + 1 , r , getRight(p)) ;

          // go to RightChild

        NewA[p] = NewA[getLeft(p)] + NewA[getRight(p)];
        //Backtracking

        /*

         *

         *  Put Sum of LeftChild and RightChild into Parent (P)

         *  if A : [ 2 3 4 5 ]

         *

         *          14

         *         /  \

         *        /    \

         *       5      9

         *      / \    / \

         *     2   3  4   5

         */

    }

}
```

```
// Update Range in Tree

void UpdateTree(int l , int r , int i , int j , int val , int p
= 1){ // Update Range(i , j)

    // Out of Range

    if(i > r || l > j) return ;

    if(l == r){

        NewA[p] += val ; // Update With Val

        return ; // There is No More Childs to Visit

    }

    int mid = l + (r - l) / 2 ; // get Mid Position

    UpdateTree(l , mid , i , j , val , getLeft(p)) ; // goto
LeftChild

    UpdateTree(mid + 1 , r , i , j , val , getRight(p)) ; //
goto RightChild

    NewA[p] = NewA[getLeft(p)] + NewA[getRight(p)] ; //
Backtracking

    /*

     * Update Value of Parent After Updating Left and Right
Child

     */

}


// Reply With Answer to User like this

// Ques : get Sum From Position 1 to Position 3

// Ans  : ....
```

```
int RangeSumQuery(int l , int r , int i , int j , int p = 1){ //
get Sum for Range(i , j)

    // Out of Range

    if(i > r || l > j) return 0 ; // Value to indicator that is
not in My Sum Range

    if(l >= i && j >= r) return NewA[p] ; // This is Exactly My
Range

    // Else i Will Browse Left and Right Child

    int mid = l + (r - l) / 2 ; // get Mid ;

    int FirstPart = RangeSumQuery(l , mid , i , j , getLeft(p))
; // goto Left Child

    int SecondPart = RangeSumQuery(mid + 1 , r , i , j ,
getRight(p)) ;  // goto Right


    return FirstPart + SecondPart ; // Return Sum Val of All My
Range

}


// More Optimization for Updating Function is Called Lazy
Propagation

/* Lazy Propagation Function Here*/


void UpdateTreeLazyPropagation(int l , int r , int i , int j ,
int val , int p = 1){

    if(l > r) return ; // Out of Range
```

```
if(Lazy[p]){
    NewA[p] += Lazy[p] ;
    if(l != r){ // Child are Not Update With New Val
        Lazy[getLeft(p)] += Lazy[p] ; // Update Left
        Lazy[getRight(p)] += Lazy[p]; // Update Right
    }
    Lazy[p] = 0 ; // This Range Was already Updated -->
}
// Out Of My Range ; // No OverLap
if(i > r || l > j) return ;

// Total OverLap
if(l >= i && j >= r){
    NewA[p] += val ;
    if(l != r){ // have 2 Child
        Lazy[getLeft(p)] += val ;
        Lazy[getRight(p)] += val ;
    }
    return ;
}

// Partial Overlap
int mid = l + (r - l) / 2 ;
```

```
    UpdateTreeLazyPropagation(l , mid , i , j , val ,
getLeft(p)) ; // goto Left

    UpdateTreeLazyPropagation(mid + 1 , r , i , j , val ,
getRight(p)) ; // goto Right


    // BackTracking Update Parent After making Changes in Left
and Right Child

    NewA[p] = NewA[getLeft(p)] + NewA[getRight(p)] ;

}


int RangeSumQueryLazyPropagation(int l , int r , int i , int j ,
int p = 1){

    if(l > r) return 0 ;


    if(Lazy[p]){

        NewA[p] += Lazy[p] ;

        if(l != r){

            Lazy[getLeft(p)] += Lazy[p] ;

            Lazy[getRight(p)] += Lazy[p] ;

        }

        Lazy[p] = 0 ;

    }


    // No OverLap

    if(l > j || i > r) return 0 ; // Indicator "This is Not My
```

```
Range"

    // Total OverLap
    if(l >= i && j >= r)
        return NewA[p] ;

    // Partial OverLap
    int mid = l + (r - l) / 2 ; // get Mid Position
    // goto LeftChild
    int First = RangeSumQueryLazyPropagation(l , mid , i , r ,
getLeft(p)) ;
    // goto RightChild
    int Second = RangeSumQueryLazyPropagation(mid + 1 , r , i ,
j , getRight(p)) ;

    return First + Second ; // Return My Range
}
```

```
int main(){

    ios_base::sync_with_stdio(0);

    cin.tie(0) ;


    int n ;

    cin >> n ;

    for(int i = 1 ; i <= n ; i++)

        cin >> A[i] ;


    BuiltTree(1 , n) ;

    /* Using LazyPropagation */

    cout << "Sum from " << 2 << " to " << 4 << " : " ;

    cout << RangeSumQueryLazyPropagation(1 , n  , 2 , 4) <<
endl;

    UpdateTreeLazyPropagation(1 , n , 2 , 4 , 1) ;

    cout << "Sum from " << 2 << " to " << 4 << " : " ;

    cout << RangeSumQueryLazyPropagation(1 , n , 2 , 4) << endl;


    return 0 ;

}
```

Thanks