**Sum Of Given Range:**

Now, it is better to solve this problem with segment tree.

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 tree. For any node which is represented by I, the left child is at index 2\*I+1. right child is at 2\*I+2. (if 0 indexing is followed. Since, we are handling array, 0 indexing is convenient). and parent is at (I-1)>>1.

**Note the Following things about segment tree:**

If height of level is calculated in 1 indexing, and the height or level or root is considered as 1, all the leave nodes are at either nth level or n-1th level.

Height of segment tree will be ceil(log2n) 

And size of the tree: 2\*2ceil(log2n)-1

**Prove that, the size of segment tree is 2\*2ceil(log2n)-1**

Now, remember the lemmas for acyclic graph(trees too):

**First,** the number of vertices with with odd degrees are even.

**Second,** the total degrees of of all nodes=2E. (where, E is the number of edges)

**Third,** the total number of vertices V=E+1

V consists of both internal nodes and leaf nodes.

Now, go back to the problems:

Now, there will be n leaf nodes. The actual values given.

Now, there is another thing which can be proven:

**In Binary tree, number of leaf nodes is always one more than nodes with two children. (now, in segment tree, either there are leaf nodes, or there is nodes with 2 children)**

**However, the general proof Is given below:**

**Case 1: There is only one node, the relationship holds**

**as T = 0, L = 1.**

**Case 2: Root has two children, i.e., degree of root is 2.**

**Sum of degrees of nodes with two children except root +**

**Sum of degrees of nodes with one child +**

**Sum of degrees of leaves + Root's degree = 2 \* (No. of Nodes - 1)**

**Putting values of above terms,**

**(T-1)\*3 + S\*2 + L + 2 = (S + T + L - 1)\*2**

**Cancelling 2S from both sides.**

**(T-1)\*3 + L + 2 = (S + L - 1)\*2**

**T - 1 = L - 2**

**T = L - 1**

**Case 3: Root has one child, i.e., degree of root is 1.**

**Sum of degrees of nodes with two children +**

**Sum of degrees of nodes with one child except root +**

**Sum of degrees of leaves + Root's degree = 2 \* (No. of Nodes - 1)**

**Putting values of above terms,**

**T\*3 + (S-1)\*2 + L + 1 = (S + T + L - 1)\*2**

**Cancelling 2S from both sides.**

**3\*T + L -1 = 2\*T + 2\*L - 2**

**T - 1 = L - 2**

**T = L - 1**

**Therefore, in all three cases, we get T = L-1.**

**So, you can get this, internal nodes of a segment tree is n-1 (where, segment tree has n leaf nodes)**

Total number of nodes= n + (n-1) = 2n-1 Now, we know its a full binary tree and thus the height is: ceil(Log2(n))

(what is a full binary tree? A binary tree where, every node either has 0 children or 2 children)

So, the atmost number of nodes a segment tree can contain is:

20+21+22+23+…2ceil(Log2(n))=**2\*2ceil(log2n)-1**

Approximately 4n since 2\*2log2n =4n

**Sum Of Given Range:  
  
// C program to show segment tree operations like construction, query**

**// and update**

**#include <stdio.h>**

**#include <math.h>**

**// A utility function to get the middle index from corner indexes.**

**int getMid(int s, int e) { return s + (e -s)/2; }**

**/\* A recursive function to get the sum of values in given range**

**of the array. The following are parameters for this function.**

**st --> Pointer to segment tree**

**si --> Index of current node in the segment tree. Initially**

**0 is passed as root is always at index 0**

**ss & se --> Starting and ending indexes of the segment represented**

**by current node, i.e., st[si]**

**qs & qe --> Starting and ending indexes of query range \*/**

**int getSumUtil(int \*st, int ss, int se, int qs, int qe, int si)**

**{**

**// If segment of this node is a part of given range, then return**

**// the sum of the segment**

**if (qs <= ss && qe >= se)**

**return st[si];**

**// If segment of this node is outside the given range**

**if (se < qs || ss > qe)**

**return 0;**

**// If a part of this segment overlaps with the given range**

**int mid = getMid(ss, se);**

**return getSumUtil(st, ss, mid, qs, qe, 2\*si+1) +**

**getSumUtil(st, mid+1, se, qs, qe, 2\*si+2);**

**}**

**/\* A recursive function to update the nodes which have the given**

**index in their range. The following are parameters**

**st, si, ss and se are same as getSumUtil()**

**i --> index of the element to be updated. This index is**

**in input array.**

**diff --> Value to be added to all nodes which have i in range \*/**

**void updateValueUtil(int \*st, int ss, int se, int i, int diff, int si)**

**{**

**// Base Case: If the input index lies outside the range of**

**// this segment**

**if (i < ss || i > se)**

**return;**

**// If the input index is in range of this node, then update**

**// the value of the node and its children**

**st[si] = st[si] + diff;**

**if (se != ss)**

**{**

**int mid = getMid(ss, se);**

**updateValueUtil(st, ss, mid, i, diff, 2\*si + 1);**

**updateValueUtil(st, mid+1, se, i, diff, 2\*si + 2);**

**}**

**}**

**// The function to update a value in input array and segment tree.**

**// It uses updateValueUtil() to update the value in segment tree**

**void updateValue(int arr[], int \*st, int n, int i, int new\_val)**

**{**

**// Check for erroneous input index**

**if (i < 0 || i > n-1)**

**{**

**printf("Invalid Input");**

**return;**

**}**

**// Get the difference between new value and old value**

**int diff = new\_val - arr[i];**

**// Update the value in array**

**arr[i] = new\_val;**

**// Update the values of nodes in segment tree**

**updateValueUtil(st, 0, n-1, i, diff, 0);**

**}**

**// Return sum of elements in range from index qs (quey start)**

**// to qe (query end). It mainly uses getSumUtil()**

**int getSum(int \*st, int n, int qs, int qe)**

**{**

**// Check for erroneous input values**

**if (qs < 0 || qe > n-1 || qs > qe)**

**{**

**printf("Invalid Input");**

**return -1;**

**}**

**return getSumUtil(st, 0, n-1, qs, qe, 0);**

**}**

**// A recursive function that constructs Segment Tree for array[ss..se].**

**// si is index of current node in segment tree st**

**int constructSTUtil(int arr[], int ss, int se, int \*st, int si)**

**{**

**// If there is one element in array, store it in current node of**

**// segment tree and return**

**if (ss == se)**

**{**

**st[si] = arr[ss];**

**return arr[ss];**

**}**

**// If there are more than one elements, then recur for left and**

**// right subtrees and store the sum of values in this node**

**int mid = getMid(ss, se);**

**st[si] = constructSTUtil(arr, ss, mid, st, si\*2+1) +**

**constructSTUtil(arr, mid+1, se, st, si\*2+2);**

**return st[si];**

**}**

**/\* Function to construct segment tree from given array. This function**

**allocates memory for segment tree and calls constructSTUtil() to**

**fill the allocated memory \*/**

**int \*constructST(int arr[], int n)**

**{**

**// Allocate memory for segment tree**

**//Height of segment tree**

**int x = (int)(ceil(log2(n)));**

**//Maximum size of segment tree**

**int max\_size = 2\*(int)pow(2, x) - 1;**

**// Allocate memory**

**int \*st = new int[max\_size];**

**// Fill the allocated memory st**

**constructSTUtil(arr, 0, n-1, st, 0);**

**// Return the constructed segment tree**

**return st;**

**}**

**// Driver program to test above functions**

**int main()**

**{**

**int arr[] = {1, 3, 5, 7, 9, 11};**

**int n = sizeof(arr)/sizeof(arr[0]);**

**// Build segment tree from given array**

**int \*st = constructST(arr, n);**

**// Print sum of values in array from index 1 to 3**

**printf("Sum of values in given range = %dn",**

**getSum(st, n, 1, 3));**

**// Update: set arr[1] = 10 and update corresponding**

**// segment tree nodes**

**updateValue(arr, st, n, 1, 10);**

**// Find sum after the value is updated**

**printf("Updated sum of values in given range = %dn",**

**getSum(st, n, 1, 3));**

**return 0;**

**}**

Now, check how segment tree is constructed:

**int constructSTUtil(int arr[], int ss, int se, int \*st, int si)**

**{**

**// If there is one element in array, store it in current node of**

**// segment tree and return**

**if (ss == se)**

**{**

**st[si] = arr[ss];**

**return arr[ss];**

**}**

**// If there are more than one elements, then recur for left and**

**// right subtrees and store the sum of values in this node**

**int mid = getMid(ss, se);**

**st[si] = constructSTUtil(arr, ss, mid, st, si\*2+1) +**

**constructSTUtil(arr, mid+1, se, st, si\*2+2);**

**return st[si];**

**}**

**/\* Function to construct segment tree from given array. This function**

**allocates memory for segment tree and calls constructSTUtil() to**

**fill the allocated memory \*/**

**int \*constructST(int arr[], int n)**

**{**

**// Allocate memory for segment tree**

**//Height of segment tree**

**int x = (int)(ceil(log2(n)));**

**//Maximum size of segment tree**

**int max\_size = 2\*(int)pow(2, x) - 1;**

**// Allocate memory**

**int \*st = new int[max\_size];**

**// Fill the allocated memory st**

**constructSTUtil(arr, 0, n-1, st, 0);**

**// Return the constructed segment tree**

**return st;**

**}**

Now, as we know, the height of a segment tree is going to be: ceil(Log2(n)). And, the atmost number of nodes that can be present:

20+21+22+23+…2ceil(Log2(n))=**2\*2ceil(log2n)-1**

Now, it creates an array of maximum size.

**After that we call:** constructSTUtil(arr, 0, n-1, st, 0);

Now, say {1, 3, 5, 7, 9, 11} is the given array. So, n=6

constructSTUtil(arr,0,5,st,0)

/ \

(arr,0,2,st,1) (arr,3,5,st,2)

/ \ / \

(arr,0,1,st,3) (arr,2,2,st,4) (arr,3,4,st,5) (arr,5,5,st,6)

/ \ / \

(arr,0,0,st,7) (arr,1,1,st,8) (arr,3,3,st,11) (arr,4,4,st,12)

36[0-5]

/ \

9[0-2] 27[3,5]

/ \ / \

4[0-1] 5[2,2] 16[3,4] 11[5,5]

/ \ / \

1[0,0] 3[1,1]7[3,3] 9[4,4]

This is how it is constructed.

How to get Sum of a particular range?

**int getSumUtil(int \*st, int ss, int se, int qs, int qe, int si)**

**{**

**// If segment of this node is a part of given range, then return**

**// the sum of the segment**

**if (qs <= ss && qe >= se)**

**return st[si];**

**// If segment of this node is outside the given range**

**if (se < qs || ss > qe)**

**return 0;**

**// If a part of this segment overlaps with the given range**

**int mid = getMid(ss, se);**

**return getSumUtil(st, ss, mid, qs, qe, 2\*si+1) +**

**getSumUtil(st, mid+1, se, qs, qe, 2\*si+2);**

**}**

**int getSum(int \*st, int n, int qs, int qe)**

**{**

**// Check for erroneous input values**

**if (qs < 0 || qe > n-1 || qs > qe)**

**{**

**printf("Invalid Input");**

**return -1;**

**}**

**return getSumUtil(st, 0, n-1, qs, qe, 0);**

**}**

36[0-5]

/ \

9[0-2] 27[3,5]

/ \ / \

4[0-1] 5[2,2] 16[3,4] 11[5,5]

/ \ / \

1[0,0] 3[1,1]7[3,3] 9[4,4]

This is the tree.

Now, check the initial conditions for getSum.

**if (qs < 0 || qe > n-1 || qs > qe)**

**{**

**printf("Invalid Input");**

**return -1;**

**}**

Where, qs is query starting range.

qe is query ending range.

Now, first, some invalid range cases are handled.

After that, we call:

**return getSumUtil(st, 0, n-1, qs, qe, 0);**

Now, suppose, we want the sum of range 2,5

[0-5] (range needs to be shortened)

/ \

[0-2] [3,5]

([0-2]need to be checked) ([3,5]. it will return the range sum of [3,5] )  
 (as, [qs,qe] is [2,5] [ss,se] is [3,5] So, qs<=ss and qe>=se we will return the sum of [ss,se]

/ \ / \

[0-1] [2-2] [3,4] [5,5]

([0,1] will be out of range.) ([2,2] will be returned)

/\

[0,0] [1,1]

UpdateValue:

**void updateValueUtil(int \*st, int ss, int se, int i, int diff, int si)**

**{**

**// Base Case: If the input index lies outside the range of**

**// this segment**

**if (i < ss || i > se)**

**return;**

**// If the input index is in range of this node, then update**

**// the value of the node and its children**

**st[si] = st[si] + diff;**

**if (se != ss)**

**{**

**int mid = getMid(ss, se);**

**updateValueUtil(st, ss, mid, i, diff, 2\*si + 1);**

**updateValueUtil(st, mid+1, se, i, diff, 2\*si + 2);**

**}**

**}**

**// The function to update a value in input array and segment tree.**

**// It uses updateValueUtil() to update the value in segment tree**

**void updateValue(int arr[], int \*st, int n, int i, int new\_val)**

**{**

**// Check for erroneous input index**

**if (i < 0 || i > n-1)**

**{**

**printf("Invalid Input");**

**return;**

**}**

**// Get the difference between new value and old value**

**int diff = new\_val - arr[i];**

**// Update the value in array**

**arr[i] = new\_val;**

**// Update the values of nodes in segment tree**

**updateValueUtil(st, 0, n-1, i, diff, 0);**

**}**

Now, first, we calculate the difference between previous value and new value in the actual position, Then reflect the value in actual array position.

Now, suppose, we are updating the value of index 4. Now, every range which contains 4 in it’s index range, will be incremented by the difference.

**Range Minimum Query:**

**Range GCM Query:**

**Min Max Range Query:**

I can do these things.

**Lazy Propagation On Segment Tree:**

If a range of element is updated together.

void update\_value\_util(int \*segment\_tree,int starting\_index,int ending\_index,int pos,int diff,int curr\_index)

{

if(starting\_index<=pos&&pos<=ending\_index)

{

segment\_tree[curr\_index]+=diff;

}

if(starting\_index!=ending\_index)

{

int mid=get\_mid(starting\_index,ending\_index);

update\_value\_util(segment\_tree,starting\_index,mid,pos,diff,2\*curr\_index+1);

update\_value\_util(segment\_tree,mid+1,ending\_index,pos,diff,2\*curr\_index+2);

}

}

**Now, how to do range update:**

void update\_value\_of\_a\_range\_util(int \*segment\_tree,int start,int end,int range\_start,int range\_end,int added\_value,int curr\_index)

{

if(start>end||end<range\_start||start>range\_end)

{

return;

}

if(start==end)

{

segment\_tree[curr\_index]+=added\_value;

return;

}

int mid=get\_mid(start,end); update\_value\_of\_a\_range\_util(segment\_tree,start,mid,range\_start,range\_end,added\_value,2\*curr\_index+1); update\_value\_of\_a\_range\_util(segment\_tree,mid+1,end,range\_start,range\_end,added\_value,2\*curr\_index+2);

segment\_tree[curr\_index]=segment\_tree[2\*curr\_index+1]+segment\_tree[2\*curr\_index+2];

}

Now, lazy propagation keeps a lazy array.

When there are many updates and updates are done on a range, we can postpone some updates (avoid recursive calls in update) and do those updates only when required.

Please remember that a node in segment tree stores or represents result of a query for a range of indexes. And if this node’s range lies within the update operation range, then all descendants of the node must also be updated. For example consider the node with value 27 in above diagram, this node stores sum of values at indexes from 3 to 5. If our update query is for range 2 to 5, then we need to update this node and all descendants of this node. With Lazy propagation, we update only node with value 27 and postpone updates to its children by storing this update information in separate nodes called lazy nodes or values. We create an array lazy[] which represents lazy node. Size of lazy[] is same as array that represents segment tree, which is tree[] in below code.

The idea is to initialize all elements of lazy[] as 0. A value 0 in lazy[i] indicates that there are no pending updates on node i in segment tree. A non-zero value of lazy[i] means that this amount needs to be added to node i in segment tree before making any query to the node.

**Now, the important point is it can make range updates faster.**

// Program to show segment tree to demonstrate lazy

// propagation

**#include <stdio.h>**

**#include <math.h>**

**#define MAX 1000**

**// Ideally, we should not use global variables and large**

**// constant-sized arrays, we have done it here for simplicity.**

**int tree[MAX] = {0}; // To store segment tree**

**int lazy[MAX] = {0}; // To store pending updates**

**/\* si -> index of current node in segment tree**

**ss and se -> Starting and ending indexes of elements for**

**which current nodes stores sum.**

**us and ue -> starting and ending indexes of update query**

**diff -> which we need to add in the range us to ue \*/**

**void updateRangeUtil(int si, int ss, int se, int us,**

**int ue, int diff)**

**{**

**// If lazy value is non-zero for current node of segment**

**// tree, then there are some pending updates. So we need**

**// to make sure that the pending updates are done before**

**// making new updates. Because this value may be used by**

**// parent after recursive calls (See last line of this**

**// function)**

**if (lazy[si] != 0)**

**{**

**// Make pending updates using value stored in lazy**

**// nodes**

**tree[si] += (se-ss+1)\*lazy[si];**

**// checking if it is not leaf node because if**

**// it is leaf node then we cannot go further**

**if (ss != se)**

**{**

**// We can postpone updating children we don't**

**// need their new values now.**

**// Since we are not yet updating children of si,**

**// we need to set lazy flags for the children**

**lazy[si\*2 + 1] += lazy[si];**

**lazy[si\*2 + 2] += lazy[si];**

**}**

**// Set the lazy value for current node as 0 as it**

**// has been updated**

**lazy[si] = 0;**

**}**

**// out of range**

**if (ss>se || ss>ue || se<us)**

**return ;**

**// Current segment is fully in range**

**if (ss>=us && se<=ue)**

**{**

**// Add the difference to current node**

**tree[si] += (se-ss+1)\*diff;**

**// same logic for checking leaf node or not**

**if (ss != se)**

**{**

**// This is where we store values in lazy nodes,**

**// rather than updating the segment tree itelf**

**// Since we don't need these updated values now**

**// we postpone updates by storing values in lazy[]**

**lazy[si\*2 + 1] += diff;**

**lazy[si\*2 + 2] += diff;**

**}**

**return;**

**}**

**// If not completely in rang, but overlaps, recur for**

**// children,**

**int mid = (ss+se)/2;**

**updateRangeUtil(si\*2+1, ss, mid, us, ue, diff);**

**updateRangeUtil(si\*2+2, mid+1, se, us, ue, diff);**

**// And use the result of children calls to update this**

**// node**

**tree[si] = tree[si\*2+1] + tree[si\*2+2];**

**}**

**// Function to update a range of values in segment**

**// tree**

**/\* us and eu -> starting and ending indexes of update query**

**ue -> ending index of update query**

**diff -> which we need to add in the range us to ue \*/**

**void updateRange(int n, int us, int ue, int diff)**

**{**

**updateRangeUtil(0, 0, n-1, us, ue, diff);**

**}**

**/\* A recursive function to get the sum of values in given**

**range of the array. The following are parameters for**

**this function.**

**si --> Index of current node in the segment tree.**

**Initially 0 is passed as root is always at'**

**index 0**

**ss & se --> Starting and ending indexes of the**

**segment represented by current node,**

**i.e., tree[si]**

**qs & qe --> Starting and ending indexes of query**

**range \*/**

**int getSumUtil(int ss, int se, int qs, int qe, int si)**

**{**

**// If lazy flag is set for current node of segment tree,**

**// then there are some pending updates. So we need to**

**// make sure that the pending updates are done before**

**// processing the sub sum query**

**if (lazy[si] != 0)**

**{**

**// Make pending updates to this node. Note that this**

**// node represents sum of elements in arr[ss..se] and**

**// all these elements must be increased by lazy[si]**

**tree[si] += (se-ss+1)\*lazy[si];**

**// checking if it is not leaf node because if**

**// it is leaf node then we cannot go further**

**if (ss != se)**

**{**

**// Since we are not yet updating children os si,**

**// we need to set lazy values for the children**

**lazy[si\*2+1] += lazy[si];**

**lazy[si\*2+2] += lazy[si];**

**}**

**// unset the lazy value for current node as it has**

**// been updated**

**lazy[si] = 0;**

**}**

**// Out of range**

**if (ss>se || ss>qe || se<qs)**

**return 0;**

**// At this point we are sure that pending lazy updates**

**// are done for current node. So we can return value**

**// (same as it was for query in our previous post)**

**// If this segment lies in range**

**if (ss>=qs && se<=qe)**

**return tree[si];**

**// If a part of this segment overlaps with the given**

**// range**

**int mid = (ss + se)/2;**

**return getSumUtil(ss, mid, qs, qe, 2\*si+1) +**

**getSumUtil(mid+1, se, qs, qe, 2\*si+2);**

**}**

**// Return sum of elements in range from index qs (quey**

**// start) to qe (query end). It mainly uses getSumUtil()**

**int getSum(int n, int qs, int qe)**

**{**

**// Check for erroneous input values**

**if (qs < 0 || qe > n-1 || qs > qe)**

**{**

**printf("Invalid Input");**

**return -1;**

**}**

**return getSumUtil(0, n-1, qs, qe, 0);**

**}**

**// A recursive function that constructs Segment Tree for**

**// array[ss..se]. si is index of current node in segment**

**// tree st.**

**void constructSTUtil(int arr[], int ss, int se, int si)**

**{**

**// out of range as ss can never be greater than se**

**if (ss > se)**

**return ;**

**// If there is one element in array, store it in**

**// current node of segment tree and return**

**if (ss == se)**

**{**

**tree[si] = arr[ss];**

**return;**

**}**

**// If there are more than one elements, then recur**

**// for left and right subtrees and store the sum**

**// of values in this node**

**int mid = (ss + se)/2;**

**constructSTUtil(arr, ss, mid, si\*2+1);**

**constructSTUtil(arr, mid+1, se, si\*2+2);**

**tree[si] = tree[si\*2 + 1] + tree[si\*2 + 2];**

**}**

**/\* Function to construct segment tree from given array.**

**This function allocates memory for segment tree and**

**calls constructSTUtil() to fill the allocated memory \*/**

**void constructST(int arr[], int n)**

**{**

**// Fill the allocated memory st**

**constructSTUtil(arr, 0, n-1, 0);**

**}**

**// Driver program to test above functions**

**int main()**

**{**

**int arr[] = {1, 3, 5, 7, 9, 11};**

**int n = sizeof(arr)/sizeof(arr[0]);**

**// Build segment tree from given array**

**constructST(arr, n);**

**// Print sum of values in array from index 1 to 3**

**printf("Sum of values in given range = %d\n",**

**getSum(n, 1, 3));**

**// Add 10 to all nodes at indexes from 1 to 5.**

**updateRange(n, 1, 5, 10);**

**// Find sum after the value is updated**

**printf("Updated sum of values in given range = %d\n",**

**getSum( n, 1, 3));**

**return 0;**

**}**

Now, if you are planning to dynamically allocate memory for segment tree and construct tree, the **constructST** and **constructSTUtil** function will remain exactly same as the normal segment tree.

**void updateRangeUtil(int si, int ss, int se, int us,**

**int ue, int diff)**

**{**

**// If lazy value is non-zero for current node of segment**

**// tree, then there are some pending updates. So we need**

**// to make sure that the pending updates are done before**

**// making new updates. Because this value may be used by**

**// parent after recursive calls (See last line of this**

**// function)**

**if (lazy[si] != 0)**

**{**

**// Make pending updates using value stored in lazy**

**// nodes**

**tree[si] += (se-ss+1)\*lazy[si];**

**// checking if it is not leaf node because if**

**// it is leaf node then we cannot go further**

**if (ss != se)**

**{**

**// We can postpone updating children we don't**

**// need their new values now.**

**// Since we are not yet updating children of si,**

**// we need to set lazy flags for the children**

**lazy[si\*2 + 1] += lazy[si];**

**lazy[si\*2 + 2] += lazy[si];**

**}**

**// Set the lazy value for current node as 0 as it**

**// has been updated**

**lazy[si] = 0;**

**}**

**// out of range**

**if (ss>se || ss>ue || se<us)**

**return ;**

**// Current segment is fully in range**

**if (ss>=us && se<=ue)**

**{**

**// Add the difference to current node**

**tree[si] += (se-ss+1)\*diff;**

**// same logic for checking leaf node or not**

**if (ss != se)**

**{**

**// This is where we store values in lazy nodes,**

**// rather than updating the segment tree itelf**

**// Since we don't need these updated values now**

**// we postpone updates by storing values in lazy[]**

**lazy[si\*2 + 1] += diff;**

**lazy[si\*2 + 2] += diff;**

**}**

**return;**

**}**

**// If not completely in rang, but overlaps, recur for**

**// children,**

**int mid = (ss+se)/2;**

**updateRangeUtil(si\*2+1, ss, mid, us, ue, diff);**

**updateRangeUtil(si\*2+2, mid+1, se, us, ue, diff);**

**// And use the result of children calls to update this**

**// node**

**tree[si] = tree[si\*2+1] + tree[si\*2+2];**

**}**

**void updateRange(int n, int us, int ue, int diff)**

**{**

**updateRangeUtil(0, 0, n-1, us, ue, diff);**

**}**

Now, you have to give the diff in updateRange. Now, suppose, from 2 to 5, every element is going to be increased by 10.

Now, update starting range is us, update ending range is ue. Now, in updateRangeUtil we will send 0 as segment tree starting index. N-1 as segment tree ending index and si as current index.  
Now, check it:  
  
 **if (lazy[si] != 0)**

**{**

**// Make pending updates using value stored in lazy**

**// nodes**

**tree[si] += (se-ss+1)\*lazy[si];**

**// checking if it is not leaf node because if**

**// it is leaf node then we cannot go further**

**if (ss != se)**

**{**

**// We can postpone updating children we don't**

**// need their new values now.**

**// Since we are not yet updating children of si,**

**// we need to set lazy flags for the children**

**lazy[si\*2 + 1] += lazy[si];**

**lazy[si\*2 + 2] += lazy[si];**

**}**

**// Set the lazy value for current node as 0 as it**

**// has been updated**

**lazy[si] = 0;**

**}**

If any lazy task is bound to si then we will increment st[si] or tree[si] by (se-ss+1)\*lazy[si] as si in segment tree represents the actual index range of [ss,se] in array, and, if lazy[si] is not 0, that means there’s an update task pending for the index range [ss,se]. Now, we do the update immediately in si. And, we make that particular update pending for childen of si (as si is a segment tree index)

Now, after that, the normal calculation will take place:

**if (ss>se || ss>ue || se<us)**

**return ;**

**// Current segment is fully in range**

**if (ss>=us && se<=ue)**

**{**

**// Add the difference to current node**

**tree[si] += (se-ss+1)\*diff;**

**// same logic for checking leaf node or not**

**if (ss != se)**

**{**

**// This is where we store values in lazy nodes,**

**// rather than updating the segment tree itelf**

**// Since we don't need these updated values now**

**// we postpone updates by storing values in lazy[]**

**lazy[si\*2 + 1] += diff;**

**lazy[si\*2 + 2] += diff;**

**}**

**return;**

**}**

**// If not completely in rang, but overlaps, recur for**

**// children,**

**int mid = (ss+se)/2;**

**updateRangeUtil(si\*2+1, ss, mid, us, ue, diff);**

**updateRangeUtil(si\*2+2, mid+1, se, us, ue, diff);**

**// And use the result of children calls to update this**

**// node**

**tree[si] = tree[si\*2+1] + tree[si\*2+2];**

Now, we update the current segment tree index by (se-ss+1)\*diff. And, make the same update pending for children of si. (I.e. updated by diff. Now, after recursively calling for is children, finally we do again,

**tree[si] = tree[si\*2+1] + tree[si\*2+2];**

**Persistent Segment Tree:**

Segment Tree is itself a great data structure that comes into play in many cases. In this post we will introduce the concept of Persistency in this data structure. Persistency, simply means to retain the changes. But obviously, retaining the changes cause extra memory consumption and hence affect the Time Complexity.

Let’s think in terms of versions i.e. for each change in our segment tree we create a new version of it.We will consider our initial version to be Version-0. Now, as we do any update in the segment tree we will create a new version for it and in similar fashion track the record for all versions.

**But creating the whole tree for every version will take O(n log n) extra space and O(n log n) time. So, this idea runs out of time and memory for large number of versions.**

That is the downside of persistent segment tree.

**// C++ program to implement persistent segment**

**// tree.**

**#include<bits/stdc++.h>**

**using namespace std;**

**#define MAXN 100**

**/\* data type for individual**

**\* node in the segment tree \*/**

**struct node**

**{**

**// stores sum of the elements in node**

**int val;**

**// pointer to left and right children**

**node\* left, \*right;**

**// required constructors........**

**node() {}**

**node(node\* l, node\* r, int v)**

**{**

**left = l;**

**right = r;**

**val = v;**

**}**

**};**

**// input array**

**int arr[MAXN];**

**// root pointers for all versions**

**node\* version[MAXN];**

**// Constructs Version-0**

**// Time Complexity : O(nlogn)**

**void build(node\* n,int low,int high)**

**{**

**if (low==high)**

**{**

**n->val = arr[low];**

**return;**

**}**

**int mid = (low+high) / 2;**

**n->left = new node(NULL, NULL, 0);**

**n->right = new node(NULL, NULL, 0);**

**build(n->left, low, mid);**

**build(n->right, mid+1, high);**

**n->val = n->left->val + n->right->val;**

**}**

**/\*\***

**\* Upgrades to new Version**

**\* @param prev : points to node of previous version**

**\* @param cur : points to node of current version**

**\* Time Complexity : O(logn)**

**\* Space Complexity : O(logn) \*/**

**void upgrade(node\* prev, node\* cur, int low, int high,**

**int idx, int value)**

**{**

**if (idx > high or idx < low or low > high)**

**return;**

**if (low == high)**

**{**

**// modification in new version**

**cur->val = value;**

**return;**

**}**

**int mid = (low+high) / 2;**

**if (idx <= mid)**

**{**

**// link to right child of previous version**

**cur->right = prev->right;**

**// create new node in current version**

**cur->left = new node(NULL, NULL, 0);**

**upgrade(prev->left,cur->left, low, mid, idx, value);**

**}**

**else**

**{**

**// link to left child of previous version**

**cur->left = prev->left;**

**// create new node for current version**

**cur->right = new node(NULL, NULL, 0);**

**upgrade(prev->right, cur->right, mid+1, high, idx, value);**

**}**

**// calculating data for current version**

**// by combining previous version and current**

**// modification**

**cur->val = cur->left->val + cur->right->val;**

**}**

**int query(node\* n, int low, int high, int l, int r)**

**{**

**if (l > high or r < low or low > high)**

**return 0;**

**if (l <= low and high <= r)**

**return n->val;**

**int mid = (low+high) / 2;**

**int p1 = query(n->left,low,mid,l,r);**

**int p2 = query(n->right,mid+1,high,l,r);**

**return p1+p2;**

**}**

**int main(int argc, char const \*argv[])**

**{**

**int A[] = {1,2,3,4,5};**

**int n = sizeof(A)/sizeof(int);**

**for (int i=0; i<n; i++)**

**arr[i] = A[i];**

**// creating Version-0**

**node\* root = new node(NULL, NULL, 0);**

**build(root, 0, n-1);**

**// storing root node for version-0**

**version[0] = root;**

**// upgrading to version-1**

**version[1] = new node(NULL, NULL, 0);**

**upgrade(version[0], version[1], 0, n-1, 4, 1);**

**// upgrading to version-2**

**version[2] = new node(NULL, NULL, 0);**

**upgrade(version[1],version[2], 0, n-1, 2, 10);**

**cout << "In version 1 , query(0,4) : ";**

**cout << query(version[1], 0, n-1, 0, 4) << endl;**

**cout << "In version 2 , query(3,4) : ";**

**cout << query(version[2], 0, n-1, 3, 4) << endl;**

**cout << "In version 0 , query(0,3) : ";**

**cout << query(version[0], 0, n-1, 0, 3) << endl;**

**return 0;**

**}**

I remember it.

**Efficiently Insert Delete Median Queries Set:**

This is very important problem.

Read the concepts from geeksforgeeks:

<https://www.geeksforgeeks.org/efficiently-design-insert-delete-median-queries-set/>

**Smallest Subarray With Given GCD:**

**// C++ Program to find GCD of a number in a given Range**

**// using segment Trees**

**#include <bits/stdc++.h>**

**using namespace std;**

**// To store segment tree**

**int \*st;**

**// Function to find gcd of 2 numbers.**

**int gcd(int a, int b)**

**{**

**if (a < b)**

**swap(a, b);**

**if (b==0)**

**return a;**

**return gcd(b, a%b);**

**}**

**/\* A recursive function to get gcd of given**

**range of array indexes. The following are parameters for**

**this function.**

**st --> Pointer to segment tree**

**si --> Index of current node in the segment tree. Initially**

**0 is passed as root is always at index 0**

**ss & se --> Starting and ending indexes of the segment**

**represented by current node, i.e., st[index]**

**qs & qe --> Starting and ending indexes of query range \*/**

**int findGcd(int ss, int se, int qs, int qe, int si)**

**{**

**if (ss>qe || se < qs)**

**return 0;**

**if (qs<=ss && qe>=se)**

**return st[si];**

**int mid = ss+(se-ss)/2;**

**return gcd(findGcd(ss, mid, qs, qe, si\*2+1),**

**findGcd(mid+1, se, qs, qe, si\*2+2));**

**}**

**//Finding The gcd of given Range**

**int findRangeGcd(int ss, int se, int arr[], int n)**

**{**

**if (ss<0 || se > n-1 || ss>se)**

**{**

**cout << "Invalid Arguments" << "\n";**

**return -1;**

**}**

**return findGcd(0, n-1, ss, se, 0);**

**}**

**// A recursive function that constructs Segment Tree for**

**// array[ss..se]. si is index of current node in segment**

**// tree st**

**int constructST(int arr[], int ss, int se, int si)**

**{**

**if (ss==se)**

**{**

**st[si] = arr[ss];**

**return st[si];**

**}**

**int mid = ss+(se-ss)/2;**

**st[si] = gcd(constructST(arr, ss, mid, si\*2+1),**

**constructST(arr, mid+1, se, si\*2+2));**

**return st[si];**

**}**

**/\* Function to construct segment tree from given array.**

**This function allocates memory for segment tree and**

**calls constructSTUtil() to fill the allocated memory \*/**

**int \*constructSegmentTree(int arr[], int n)**

**{**

**int height = (int)(ceil(log2(n)));**

**int size = 2\*(int)pow(2, height)-1;**

**st = new int[size];**

**constructST(arr, 0, n-1, 0);**

**return st;**

**}**

**// Returns size of smallest subarray of arr[0..n-1]**

**// with GCD equal to k.**

**int findSmallestSubarr(int arr[], int n, int k)**

**{**

**// To check if a multiple of k exists.**

**bool found = false;**

**// Find if k or its multiple is present**

**for (int i=0; i<n; i++)**

**{**

**// If k is present, then subarray size is 1.**

**if (arr[i] == k)**

**return 1;**

**// Break the loop to indicate presence of a**

**// multiple of k.**

**if (arr[i] % k == 0)**

**found = true;**

**}**

**// If there was no multiple of k in arr[], then**

**// we can't get k as GCD.**

**if (found == false)**

**return -1;**

**// If there is a multiple of k in arr[], build**

**// segment tree from given array**

**constructSegmentTree(arr, n);**

**// Initialize result**

**int res = n+1;**

**// Now consider every element as starting point**

**// and search for ending point using Binary Search**

**for (int i=0; i<n-1; i++)**

**{**

**// a[i] cannot be a starting point, if it is**

**// not a multiple of k.**

**if (arr[i] % k != 0)**

**continue;**

**// Initialize indexes for binary search of closest**

**// ending point to i with GCD of subarray as k.**

**int low = i+1;**

**int high = n-1;**

**// Initialize closest ending point for i.**

**int closest = 0;**

**// Binary Search for closest ending point**

**// with GCD equal to k.**

**while (true)**

**{**

**// Find middle point and GCD of subarray**

**// arr[i..mid]**

**int mid = low + (high-low)/2;**

**int gcd = findRangeGcd(i, mid, arr, n);**

**// If GCD is more than k, look further**

**if (gcd > k)**

**low = mid;**

**// If GCD is k, store this point and look for**

**// a closer point**

**else if (gcd == k)**

**{**

**high = mid;**

**closest = mid;**

**break;**

**}**

**// If GCD is less than, look closer**

**else**

**high = mid;**

**// If termination condition reached, set**

**// closest**

**if (abs(high-low) <= 1)**

**{**

**if (findRangeGcd(i, low, arr, n) == k)**

**closest = low;**

**else if (findRangeGcd(i, high, arr, n)==k)**

**closest = high;**

**break;**

**}**

**}**

**if (closest != 0)**

**res = min(res, closest - i + 1);**

**}**

**// If res was not changed by loop, return -1,**

**// else return its value.**

**return (res == n+1) ? -1 : res;**

**}**

**// Driver program to test above functions**

**int main()**

**{**

**int n = 8;**

**int k = 3;**

**int arr[] = {6, 9, 7, 10, 12, 24, 36, 27};**

**cout << "Size of smallest sub-array with given"**

**<< " size is " << findSmallestSubarr(arr, n, k);**

**return 0;**

**}**

The important part is how do we find the range using a binary search based method.

**for(int i=0;i<n;i++)**

**{**

**If(arr[i]%k!=0)**

**{**

**Continue;**

**//because, no range can have gcd k with starting index as i**

}

Now, calculate mid=(i+1,n-1)/2;

Now, see if gcd(i,mid) is k if. It is more than k, then we will set low=mid as the range needs be expanded. Otherwise, shorten the range if gcd==k.

**Finding Largest Rectangular Area In The Histogram:**

**// A Divide and Conquer Program to find maximum rectangular area in a histogram**

**#include <math.h>**

**#include <limits.h>**

**#include <iostream>**

**using namespace std;**

**// A utility function to find minimum of three integers**

**int max(int x, int y, int z)**

**{ return max(max(x, y), z); }**

**// A utility function to get minimum of two numbers in hist[]**

**int minVal(int \*hist, int i, int j)**

**{**

**if (i == -1) return j;**

**if (j == -1) return i;**

**return (hist[i] < hist[j])? i : j;**

**}**

**// A utility function to get the middle index from corner indexes.**

**int getMid(int s, int e)**

**{ return s + (e -s)/2; }**

**/\* A recursive function to get the index of minimum value in a given range of**

**indexes. The following are parameters for this function.**

**hist --> Input array for which segment tree is built**

**st --> Pointer to segment tree**

**index --> Index of current node in the segment tree. Initially 0 is**

**passed as root is always at index 0**

**ss & se --> Starting and ending indexes of the segment represented by**

**current node, i.e., st[index]**

**qs & qe --> Starting and ending indexes of query range \*/**

**int RMQUtil(int \*hist, int \*st, int ss, int se, int qs, int qe, int index)**

**{**

**// If segment of this node is a part of given range, then return the**

**// min of the segment**

**if (qs <= ss && qe >= se)**

**return st[index];**

**// If segment of this node is outside the given range**

**if (se < qs || ss > qe)**

**return -1;**

**// If a part of this segment overlaps with the given range**

**int mid = getMid(ss, se);**

**return minVal(hist, RMQUtil(hist, st, ss, mid, qs, qe, 2\*index+1),**

**RMQUtil(hist, st, mid+1, se, qs, qe, 2\*index+2));**

**}**

**// Return index of minimum element in range from index qs (quey start) to**

**// qe (query end). It mainly uses RMQUtil()**

**int RMQ(int \*hist, int \*st, int n, int qs, int qe)**

**{**

**// Check for erroneous input values**

**if (qs < 0 || qe > n-1 || qs > qe)**

**{**

**cout << "Invalid Input";**

**return -1;**

**}**

**return RMQUtil(hist, st, 0, n-1, qs, qe, 0);**

**}**

**// A recursive function that constructs Segment Tree for hist[ss..se].**

**// si is index of current node in segment tree st**

**int constructSTUtil(int hist[], int ss, int se, int \*st, int si)**

**{**

**// If there is one element in array, store it in current node of**

**// segment tree and return**

**if (ss == se)**

**return (st[si] = ss);**

**// If there are more than one elements, then recur for left and**

**// right subtrees and store the minimum of two values in this node**

**int mid = getMid(ss, se);**

**st[si] = minVal(hist, constructSTUtil(hist, ss, mid, st, si\*2+1),**

**constructSTUtil(hist, mid+1, se, st, si\*2+2));**

**return st[si];**

**}**

**/\* Function to construct segment tree from given array. This function**

**allocates memory for segment tree and calls constructSTUtil() to**

**fill the allocated memory \*/**

**int \*constructST(int hist[], int n)**

**{**

**// Allocate memory for segment tree**

**int x = (int)(ceil(log2(n))); //Height of segment tree**

**int max\_size = 2\*(int)pow(2, x) - 1; //Maximum size of segment tree**

**int \*st = new int[max\_size];**

**// Fill the allocated memory st**

**constructSTUtil(hist, 0, n-1, st, 0);**

**// Return the constructed segment tree**

**return st;**

**}**

**// A recursive function to find the maximum rectangular area.**

**// It uses segment tree 'st' to find the minimum value in hist[l..r]**

**int getMaxAreaRec(int \*hist, int \*st, int n, int l, int r)**

**{**

**// Base cases**

**if (l > r) return INT\_MIN;**

**if (l == r) return hist[l];**

**// Find index of the minimum value in given range**

**// This takes O(Logn)time**

**int m = RMQ(hist, st, n, l, r);**

**//minimum element’s index is found**

**/\* Return maximum of following three possible cases**

**a) Maximum area in Left of min value (not including the min)**

**a) Maximum area in right of min value (not including the min)**

**c) Maximum area including min \*/**

**/\*Now, RMQ must return index of minimum element\*/**

**return max(getMaxAreaRec(hist, st, n, l, m-1),**

**getMaxAreaRec(hist, st, n, m+1, r),**

**(r-l+1)\*(hist[m]) );**

**}**

**// The main function to find max area**

**int getMaxArea(int hist[], int n)**

**{**

**// Build segment tree from given array. This takes**

**// O(n) time**

**int \*st = constructST(hist, n);**

**// Use recursive utility function to find the**

**// maximum area**

**return getMaxAreaRec(hist, st, n, 0, n-1);**

**}**

**// Driver program to test above functions**

**int main()**

**{**

**int hist[] = {6, 1, 5, 4, 5, 2, 6};**

**int n = sizeof(hist)/sizeof(hist[0]);**

**cout << "Maximum area is " << getMaxArea(hist, n);**

**return 0;**

**}**

Now, note that, in the segment tree, we are storing the index of the minimum.

**Find LCA in Binary Tree using RMQ:**

Range Minimum Query (RMQ) is used on arrays to find the position of an element with the minimum value between two specified indices. Different approaches for solving RMQ have been discussed here and here. In this article, Segment Tree based approach is discussed. With segment tree, preprocessing time is O(n) and time to for range minimum query is O(Logn). The extra space required is O(n) to store the segment tree.

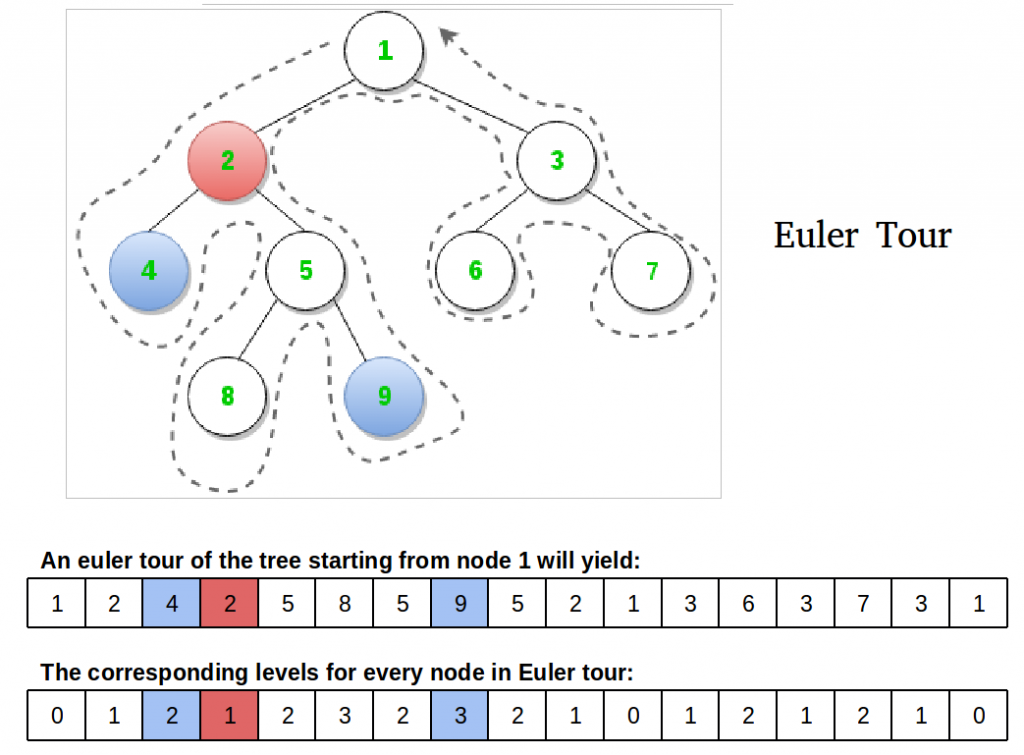
Reduction of LCA to RMQ:

The idea is to traverse the tree starting from root by an Euler tour (traversal without lifting pencil), which is a DFS-type traversal with preorder traversal characteristics.

Do a Euler tour on the tree, and fill the euler, level and first occurrence arrays.

Using the first occurrence array, get the indices corresponding to the two nodes which will be the corners of the range in the level array that is fed to the RMQ algorithm for the minimum value.

Once the algorithm return the index of the minimum level in the range, we use it to determine the LCA using Euler tour array.



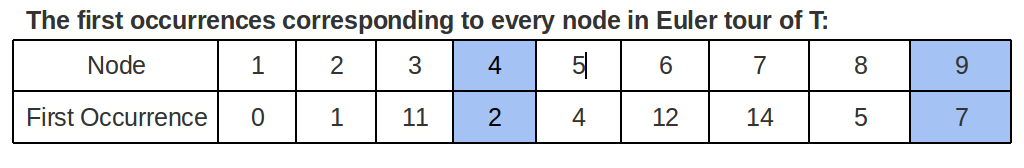
Now, this tree’s Euler tour is 1 2 4 2 5 8 5 9 5 2 1 3 6 3 7 3 1

Now, Level of every node is needed to be stored:

0 1 2 1 2 3 2 3 2 1 0 1 2 1 2 1 0

Now, level is very important. As, this will be actually stored in segment tree.

Now, first occurrence is important too. (in euler tour, the first occurrence of every element)



**Segment Tree Java Tips:**To calculate the size of segment tree, you need: log2 and ceil

public static double log( double a, double b )

{

return Math.log(a) / Math.log(b);

}

public static double log2( double a )

{

return logb(a,2);

}

Now, there is not direct function for log2.