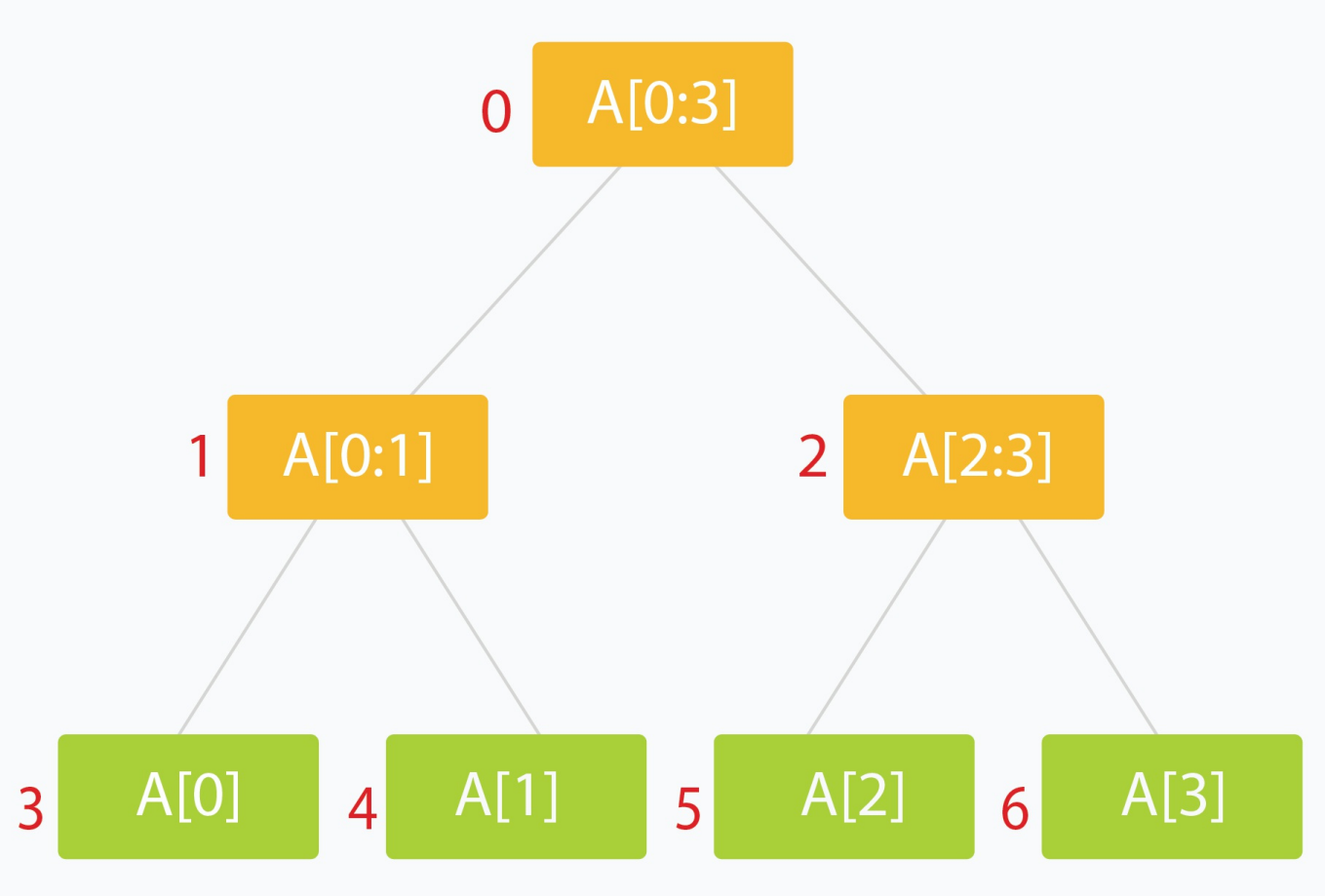
Segment Tree

What is a Segment Tree?

A Segment Tree is a full binary tree where each node represents an interval.

Generally a node would store one or more properties of an interval which can be queried later.



Segment Tree of an Array consisting of four elements

Why do we need Segment Tree?

Many problem require that we give results based on query over a range or segment of available data. This can be a tedious and slow process, especially if the number of queries is large.
A segment tree let’s us process such queries efficiently in logarithmic order of time.

How do we make a Segment Tree?

Let the data be in array arr[]

1. The **root** of our tree will represent the entire interval of data we are interested in, i.e, **arr[0…n-1]**.
2. Each **leaf** of the tree will represent a range consisting of just a single element. Thus the leaves represent arr[0], arr[1], …,arr[n-1].
3. The internal nodes will represent the merged result of their child nodes.
4. Each of the children nodes could represent approximately half of the range represented by their parent.

Segment Tree generally contain three types of methods: **BUILD**, **QUERY**, **UPDATE**

Let’s start with an example:

Problem : Range Minimum Query

You are given a list of N numbers and Q queries. There are two types of query:

1. **0 l r** - find the minimum element in range [l,r].
2. **1 i a** - update the element at ith position of array to val.

The problem states that we have to find minimum element in a given range [i,j], or update the element of the given array.

Basic Approach

```
For any range [i,j], we can answer the minimum element in
 O(n). But as
we'll be having Q queries then we have to find minimum of
 a range,
Q times.
So the overall complexity will be O(QxN).
```

So, to solve range based problems and to bring the complexity to logarithmic time we use **Segment Tree**.
We'll use tree[] to store the nodes of our Segment Tree(initialized to all zeros).

- The root of the tree is at index 1, i.e, tree[1] is the root of our tree.
- The children of tree[i] are stored at tree[i * 2] and tree[i * 2 +

1].

```
#include <iostream>
using namespace std;
int tree[400005];   //will store our segment tree structu
re
int arr[100005];    //input array
```

BUILD Function

Here, Build Function takes three parameter, ***node***, ***a***, ***b***.

```
void build_tree(int node, int a, int b) {   //BUILD funct
ion

    // node represents the current node number
    // a, b represents the current node range

    // for leaf, node range will be single element
    // so 'a' will be equal to 'b' for leaf node
    if (a == b) {   //checking if node is leaf
        //for single element, minimum will be the element
 itself
        tree[node] = arr[a];    //storing the answer in o
ur tree structure
        return;
    }
```

```
    int mid = (a + b) >> 1;     //middle element of our r
ange
    build_tree(node * 2, a, mid);   //recursively call fo
r left half
    build_tree(node * 2 + 1, mid + 1, b);   //call for ri
ght half

    //left child and right child are build
    // now build the parent node using its child nodes

    tree[node] = __min(tree[node*2], tree[node * 2 + 1]);
}
```

QUERY Function

```
int query_tree(int node, int a, int b, int i, int j) {

    // i, j represents the range to be queried
    if (a > b || a > j || b < i) return INT_MAX;    //out
 of range
    if (a >= i && b <= j) { //current segment is totally
within range[i,j]
        return tree[node];
    }
    int mid = (a + b) >> 1;
    //Query left child
    int q1 = query_tree(node * 2, a, mid, i, j);
    //Query right child
```

```
    int q2 = query_tree(node * 2 + 1, mid + 1, b, i, j);
    return __min(q1, q2); // return final result
}
```

UPDATE Function

```
// updates the ith element to val
void update_tree(int node, int a, int b, int i, int val)
{

    if (a > b || a > i || b < i) return;    //out of rang
e
    if (a == b) {   //leaf node
        tree[node] = val;
        return;
    }
    int mid = (a + b) >> 1;
    update_tree(node * 2, a, mid, i, val);  // updating l
eft child
    update_tree(node * 2 + 1, mid + 1, b, i, val);// upda
ting right child

    // updating node with min value
    tree[node] = __min(tree[node * 2], tree[node * 2 + 1]
);
}
```

Complexity Analysis

BUILD

We visit each leaf of the Segment Tree. That makes **n** leaves. Also there will be **n-1** internal nodes. So we process about **2 * n** nodes. This makes the Build process run in **O(n)** linear complexity.

UPDATE

The update process discards half of the range for every level of recursion to reach the appropriate leaf in the tree. This is similar to binary search and takes **logarithmic time**. After the leaf is updated, its direct ancestors at each level of the tree are updated. This takes time linear to height of the tree. So the complexity will be **O(lg(n))**.

QUERY

The query process traverses depth-first through the tree looking for node(s) that match exactly with the queried range. At best, we query for the entire range and get our result from the root of the segment tree itself. At worst, we query for a interval/range of size 1 (which corresponds to a single element), and we end up traversing through the height of the tree. This takes time linear to height of the tree. So the complexity will be **O(lg(n))**.

Problem : Range Sum

We have an array arr{0,…….n,– 1} and we have to perform two types of queries:

1. Find sum of elements from index l to r where **0 <= l <= r <= n**
2. Increase the value of all elements from l to r by a given number.

As we can build the parent node using its two child nodes. So, we'll use the Segment Tree to answer the sum of elements from **l** to **r**.

CODE:

```
int tree[400005];   //will store our segment tree structu
re
int arr[100005];    //input array
void build_tree(int node, int a, int b) {   //BUILD funct
ion
    if (a == b) {   //checking if node is leaf
        //for single element, sum will be the element its
elf
        tree[node] = arr[a];    //storing the answer in o
ur tree structure
        return;
    }
    int mid = (a + b) >> 1;     //middle element of our r
ange
    build_tree(node * 2, a, mid);   //recursively call fo
r left half
    build_tree(node * 2 + 1, mid + 1, b);   //call for ri
ght half
```

```
    //left child and right child are build
    // now build the parent node using its child nodes
    tree[node] = tree[node * 2] + tree[node * 2 + 1];
}

int query_tree(int node, int a, int b, int i, int j) {
    // i, j represents the range to be queried
    if (a > b || a > j || b < i) return 0;  //out of rang
e
    if (a >= i && b <= j) { //current segment is totally
within range[i,j]
        return tree[node];
    }
    int mid = (a + b) >> 1;
    int q1 = query_tree(node * 2, a, mid, i, j); //Query
left child
    int q2 = query_tree(node * 2 + 1, mid + 1, b, i, j);
 //Query right child
    return (q1 + q2); // return final result
}

void update_tree(int node, int a, int b, int i,int j, int
 val) {
    if (a > b || a > j || b < i) return;    //out of rang
e
    if (a == b) {   //leaf node
        tree[node] += val;
        return;
```

```
    }
    int mid = (a + b) >> 1;
    update_tree(node * 2, a, mid, i, j, val);;// updating
 left child
    update_tree(node * 2 + 1, mid + 1, b, i, j, val);   /
/ updating right child
    tree[node] = tree[node * 2] + tree[node * 2 + 1];
}
```

The above approach will query the tree in **O(lg(n))** time. But the problem will arise in the **update** function. As updating a single element takes **O(lg(n))** time, now we are doing a range update. So, the overall complexity for update will be **O(nlg(n))**, which will get **TLE** as we have to answer for multiple queries.

Lazy Propagation

In the current version when we update a range, we branch its childs even if the segment is covered within range. In the lazy version we only mark its child that it needs to be updated and update it when needed.

In short, we try to postpone updating descendants of a node, until the descendants themselves need to be accessed.

We use another array **lazy[]** which is the same size as our segment tree array **tree[]** to represent a lazy node. **lazy[i]** holds the amount by which the node **tree[i]** needs to be incremented, when that node is finally accessed or queried. When **lazy[i]** is zero, it means that node **tree[i]** is not lazy and has no pending updates.

UPDATE function

```
void update_tree(int node, int a, int b, int i,int j, int
 val) {
   if (lazy[node] != 0) {  // lazy node
       tree[node] += (b - a + 1)*lazy[node]; //normalize
current node by removing laziness
       if (a != b) {       // update lazy[] for children
nodes
           lazy[node * 2] += lazy[node];
           lazy[node * 2 + 1] += lazy[node];
       }
       lazy[node] = 0; // remove laziness from current no
de
   }
   if (a > b || a > j || b < i) return;    //out of range
   if (a >= i && b <= j) { // segment is fully within upd
ate range
       tree[node] += (b - a + 1)*val;
       if (a != b) {       // update lazy[] for children
nodes
           lazy[node * 2] += lazy[node];
           lazy[node * 2 + 1] += lazy[node];
       }
       return;
   }
   int mid = (a + b) >> 1;
   update_tree(node * 2, a, mid, i, j, val);;// updating
```

```
left child
   update_tree(node * 2 + 1, mid + 1, b, i, j, val);   //
 updating right child
   tree[node] = tree[node * 2] + tree[node * 2 + 1];
}
```

QUERY function

```
int query_tree(int node, int a, int b, int i, int j) {
   // i, j represents the range to be queried
   if (a > b || a > j || b < i) return INT_MAX;    //out
of range
   if (lazy[node] != 0) {  // lazy node
       tree[node] += (b - a + 1)*lazy[node]; //normalize
current node by removing laziness
       if (a != b) {       // update lazy[] for children
nodes
           lazy[node * 2] += lazy[node];
           lazy[node * 2 + 1] += lazy[node];
       }
       lazy[node] = 0; // remove laziness from current no
de
   }
   if (a >= i && b <= j) { //current segment is totally w
ithin range[i,j]
       return tree[node];
   }
   int mid = (a + b) >> 1;
```

```
   int q1 = query_tree(node * 2, a, mid, i, j); //Query l
eft child
   int q2 = query_tree(node * 2 + 1, mid + 1, b, i, j);
//Query right child
   return (q1 + q2); // return final result
}
```

Using Lazy Propagation the range update will be done in **O(lg(n))** time complexity.

Note :

The following lines:

```
tree[node] += (b - a + 1)*lazy[node]; //normalize current
 node by removing     laziness
and
tree[node] += (b - a + 1)*val; // update segment
and
tree[node] = tree[node * 2] + tree[node * 2 + 1];  //merg
e updates
```

are specific to Range Sum Query problem. Different problems may have different updating and merging schemes.

Problem : GSS1 - Can you answer

these queries I

You are given a sequence A[1], A[2], ..., A[N] . (|A[i]| ≤ 15007 , 1 ≤ N ≤ 50000). A query is defined as follows:

Query(x,y) = Max { a[i]+a[i+1]+...+a[j] ; x ≤ i ≤ j ≤ y }.

Given M queries, your program must output the results of these queries.

INPUT

```
3
-1 -2 -3
1
1 2
```

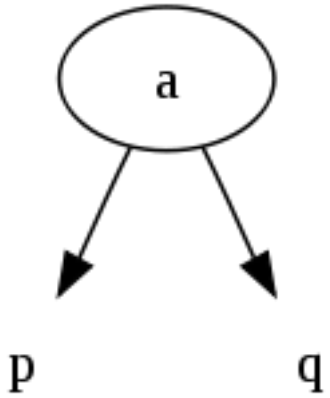
OUTPUT

```
2
```

http://www.spoj.com/problems/GSS1/

Now we need to use a Segment Tree. But what data to store in each node, such that it is easy to compute the data associated with a given node if we already know the data associated with its two child nodes.

We need to find a maximum sum subarray in a given range.



Say we have **a** as a parent node and **p** and **q** as its child nodes.

Now we need to build data for **a** from **p** and **q** such that node **a** can give the maximum sum subinterval for its range when queried.

So for this what do we need??

Maximum sum subarray in **a** can be equal to either:

1. Maximum sum subarray in **p**
2. Maximum sum subarray in **q**
3. Elements including from both **p** and **q**

So for each node we need to store:

1. Maximum Prefix Sum
2. Maximum Suffix Sum
3. Total Sum
4. Best Sum (Maximum Sum Subarray for **a**)

Maximum Prefix Sum can be calculated for a node as:

a.prefix = max(p.prefix,p.total + q.prefix)

Maximum Suffix Sum can be calculated for a node as:

a.suffix = max(p.suffix,q.total + p.suffix)

Total Sum:

a.total = p.total + q.total

Best Sum:

a.best = max(p.suffix + q.prefix , max(p.best,q.best))

CODE :

```
long long int arr[50005];   //input array
struct Tree { long long int prefix, suffix, total, best;
};
Tree tree[200005];

void build_tree(long long int node, long long int a, long
 long int b) {
    if (a == b) {   //leaf node
        tree[node].prefix = arr[a]; // prefix sum
        tree[node].suffix = arr[a]; // suffix sum
        tree[node].total = arr[a];  // total sum
        tree[node].best = arr[a];   // best sum
        return;
    }
    int mid = (a + b) >> 1;
    build_tree(node * 2, a, mid);
    build_tree(node * 2 + 1, mid + 1, b);

    tree[node].prefix = max(tree[node * 2].prefix, tree[n
ode*2].total + tree[node * 2 + 1].prefix); //prefix sum
    tree[node].suffix = max(tree[node * 2 + 1].suffix, tr
ee[node * 2].suffix + tree[node * 2 + 1].total);   //suff
```

```
ix sum
    tree[node].total = tree[node * 2].total + tree[node *
 2 + 1].total; //total sum
    tree[node].best = max(tree[node * 2].suffix + tree[no
de * 2 + 1].prefix, max(tree[node * 2].best, tree[node *
2 + 1].best));    //best sum
}

Tree query_tree(long long int node, long long int a, long
 long int b, long long int i, long long int j) {
    Tree t;
    if (a > b || a > j || b < i) {
        t.prefix = t.suffix = t.best = INT_MIN ;
        t.total = 0;
        return t;
    }
    if (a >= i && b <= j) {
        return tree[node];
    }
    long long int mid = (a + b) >> 1;
    Tree q1 = query_tree(node * 2, a, mid, i, j);
    Tree q2 = query_tree(node * 2 + 1, mid + 1, b, i, j);

    t.prefix = max(q1.prefix, q1.total + q2.prefix);
    t.suffix = max(q2.suffix, q1.suffix + q2.total);
    t.total = q1.total + q2.total;
    t.best = max(q1.suffix + q2.prefix, max(q1.best, q2.b
est));
```

```
    return t;
}
```

Similar Problem: http://www.spoj.com/problems/GSS3/

The problem also includes an update operation, rest of the methods are same.

Problem : QSET (Queries on the String)

You have a string of N decimal digits, denoted by numbers A1,A2, …, AN.

Now you are given M queries, each of whom is of following two types.

1. 1 X Y: Replace AX by Y.
2. 2 C D: Print the number of sub-strings divisible by 3 in range [C,D]

Formally, you have to print the number of pairs (i,j) such that the string Ai,Ai+1…Aj,

(C ≤ i ≤ j ≤ D), when considered as a decimal number, is divisible by 3.

INPUT

```
5 3
01245
2 1 3
1 4 5
2 3 5
```

OUTPUT

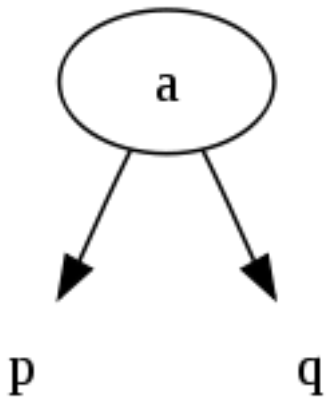
```
3
1
```

https://www.codechef.com/problems/QSET

A number is divisible by 3 if sum of its digits are divisible by 3.

Use segment trees. Store in node for each interval,

- the answer for that interval
- prefix[], where prefix[i] denotes number of prefixes of interval which modulo 3 give i.
- suffix[], where suffix[i] denotes number of suffixes of interval which modulo 3 give i.
- total sum of interval modulo 3.



Answer for interval denoted by **a** will be:

a.ans = p.ans + q.ans.

(No. of Suffixes in **p** which when taken with modulo 3 gives 1) + (No. of Prefixes in **q** which when taken with modulo 3 gives 2) --> also gives

the substring divisible by 3 in **a**.

Similarly, for all possible combinations we can calculate the **answer** for an interval.

```
a.ans = p.ans + q.ans;
    for (int i = 0; i < 3; ++i) {
        for (int j = 0; j < 3; ++j) {
            if ((i + j) % 3 == 0) {
                a.ans += p.suffix[i] * q.prefix[j];
            }
        }
    }
```

How to build prefix and suffix array for a node?

There are p.prefix[i] prefixes of **p** which when taken modulo with 3 gives i, and there are some prefixes which are made by combining whole of **p** and some prefixes of **q**.

So, total prefixes in **a** which when taken modulo with 3 gives i will be: (Same for suffix)

```
    for (int i = 0; i < 3; ++i) {
        a.prefix[i] = p.prefix[i] + q.prefix[(3 - q1.tota
l + i) % 3];
        a.suffix[i] = q.suffix[i] + p.suffix[(3 - q2.tota
l + i) % 3];
```

```
    }
```

CODE :

```
int arr[100005];    //input array
struct Tree {
    int ans;
    int prefix[3], suffix[3];
    int total;
};
Tree tree[400005];

void build_tree(int node, int a, int b) {

    if (a == b) {
        tree[node].prefix[arr[a] % 3] = 1;
        tree[node].suffix[arr[a] % 3] = 1;
        tree[node].total = arr[a] % 3;
        tree[node].ans = (arr[a] % 3 == 0 ? 1 : 0);
        return;
    }
    int mid = (a + b) >> 1;
    build_tree(node * 2, a, mid);
    build_tree(node * 2 + 1, mid + 1, b);
    for (int i = 0; i < 3; ++i) {
        tree[node].prefix[i] = tree[node * 2].prefix[i] +
 tree[node * 2 + 1].prefix[(3 - tree[node * 2].total + i)
 % 3];
```

```
        tree[node].suffix[i] = tree[node * 2 + 1].suffix[
i] + tree[node * 2].suffix[(3 - tree[node * 2 + 1].total
+ i) % 3];
    }
    tree[node].total = (tree[node * 2].total + tree[node
* 2 + 1].total) % 3;
    tree[node].ans = tree[node * 2].ans + tree[node * 2 +
 1].ans;
    for (int i = 0; i < 3; ++i) {
        for (int j = 0; j < 3; ++j) {
            if ((i + j) % 3 == 0) {
                tree[node].ans += tree[node * 2].suffix[i
] * tree[node * 2 + 1].prefix[j];
            }
        }
    }
}

Tree query_tree(int node,int a,int b,int i,int j) {
    Tree t;
    if (a > b || a > j || b < i) {
        t.ans = t.total = 0;
        for (int i = 0; i < 3; ++i) {
            t.suffix[i] = t.prefix[i] = 0;
        }
        return t;
    }
    if (a >= i && b <= j) {
```

```
        return tree[node];
    }
    int mid = (a + b) >> 1;
    Tree q1 = query_tree(node * 2, a, mid, i, j);
    Tree q2 = query_tree(node * 2 + 1, mid + 1, b, i, j);

    for (int i = 0; i < 3; ++i) {
        t.prefix[i] = q1.prefix[i] + q2.prefix[(3 - q1.to
tal + i) % 3];
        t.suffix[i] = q2.suffix[i] + q1.suffix[(3 - q2.to
tal + i) % 3];
    }
    t.total = (q1.total + q2.total) % 3;
    t.ans = q1.ans + q2.ans;
    for (int i = 0; i < 3; ++i) {
        for (int j = 0; j < 3; ++j) {
            if ((i + j) % 3 == 0) {
                t.ans += q1.suffix[i] * q2.prefix[j];
            }
        }
    }
    return t;
}
```

PROBLEM : JTREE (JosephLand)

Nick lives in a country named JosephLand. JosephLand consists of **N**

cities. **City 1** is the capital city. There are **N - 1** directed roads. It’s guaranteed that it is possible to reach capital city from any city, and in fact there is a unique path from any city to the capital city.
Besides, you can’t cross roads for free. To pass a road, you must have a ticket. There are total **M** tickets. You can not have more than one ticket at a time! Each ticket is represented by three integers:

v k w : you can buy a ticket with cost **w** in the city **v**. This ticket can be used at max **k** times. That means, after using this ticket for **k** roads ticket can’t be used!

By the way, you can tear your ticket any time and buy a new one. But you are not allowed to buy a ticket if you are still having a ticket with you!
Nick’s home is located in the capital city. He has **Q** friends, and he wants to invite all of them for dinner! So he is interested in knowing about how much each of his friends is going to spend in the journey! His friends are quite smart and always choose a route to capital city that minimizes his/her spending! Nick has to prepare dinner, so he doesn’t have time to figure out himself, Can you please help him?
Please note that **it’s guaranteed that, one can reach the capital from any city using the tickets!**

INPUT :

```
7 7
3 1
2 1
```

```
7 6
6 3
5 3
4 3
7 2 3
7 1 1
2 3 5
3 6 2
4 2 4
5 3 10
6 1 20
3
5
6
7
```

OUTPUT :

```
10
22
5
```

https://www.codechef.com/problems/JTREE

The problem in simple words is:
Given a tree with edges directed towards root and nodes having some ticket information which allows you to travel **k** units towards the root

with cost **w** . Answer **Q** queries i.e output the minimum cost for travelling from a node **x** to root.

Let’s try to solve problem for a node X at depth H from root 1. Think of this path as a 1D vector V where we have all the H-1 nodes between root and X. The nodes are stored in vector in increasing order of depth.
Let dp[x] be the minimum cost to travel from X to the root.
So, for a given node, iterate over all the tickets present at node X and DP state will be like this:

```
for(int i=0;i<total_tickets;i++)
{
     K=current_ticket_jump_info;
     W=weight_ticket_info;
     for(int j=V.size()-1;j>= max(0,V.size()-1-k);j++) //
loop 2
     {
        DP[X]=min( DP[X] , DP[V[j]] + W );
     }
}
```

We will use DFS to find vector V for every node X.
Now the code will be :

```
void dfs(int u,int p)
{
         V.push_back(u);
```

```
          for(int l=0 ; l< adj[u].size() ; l++)
          {
             if(adj[u][l]==p)continue;
             int x =adj[u][l];
             for(int i=0;i<total_tickets;i++) // loop 1
            {
                K=current_ticket_jump_info;
                W=weight_ticket_info;
                for(int j=V.size()-1;j>= max(0,V.size()-1-
k);j++) //loop 2
                {
                     DP[x]=min( DP[x] , DP[V[j]] + W );
                 }
              }
             dfs(adj[u][l],u);
          }
          V.pop_back();
}
```

The complexity will be **O(n + m * n)**.
But If we look at the inner loop of the code which goes up to K times and we find the minimum of DP , this can be done using any data structure that supports **RMQ**(Range Minimum QUery) + Point updates in O(logn) time . Which will make the total complexity to be **O(N + MlogN)**.
So we can build a segment tree that supports two operations.
First : find the minimum element in range L,R and

Second: update an element's value to VAL.

And we need to query for the minimum element between [H-K , H] so we will build the tree over height H .

Now just print DP[x] for every query.

CODE :

```
void dfs(long long int cur, long long int h) {
    for (int j = 0; j < n_info[cur].size(); ++j) {  //no. of tickets
        long long int k1 = n_info[cur][j].first;
        long long int w1 = n_info[cur][j].second;
        dp[cur] = min(dp[cur], query_tree(1, 0, N - 1, max(0LL, h - k1), h) + w1);    //find the min cost from cur to root
    }
    update_tree(1, 0, N - 1, h, dp[cur]);   //update the tree at posn h with its DP value
    for (int i = 0; i < g_tree[cur].size(); ++i) {
        long long int x = g_tree[cur][i];
        dfs(x, h + 1);
    }
    update_tree(1, 0, N - 1, h, INF_n); //update the tree at posn h with INF as we are done with its subtree
}

int main() {
    for (int i = 0; i < 400003; ++i) {
```

```
            tree[i] = INF_n;
            dp[i / 4] = INF_n;
        }
        cin >> N >> M;
        for (int i = 0; i < N - 1; ++i) {
            cin >> a >> b;
            g_tree[b].push_back(a);
        }
        for (int i = 0; i < M; ++i) {
            cin >> v >> k >> w;
            n_info[v].push_back(make_pair(k, w));
        }
        dp[1] = 0;
        update_tree(1, 0, N - 1, 0, dp[1]);
        dfs(1, 0);
        cin >> Q;
        while (Q--) {
            cin >> a;
            cout << dp[a] << "\n";
        }
        return 0;
}
```

PROBLEM : COUNZ (Counting Zeroes)

You are given an array of N integers(Indexed at 1)

For the given array you have to answer some queries given later.The

queries are of 2 types:

1. **TYPE 1** -> **1 L R** (where L and R are integers)
   For this query you have to calculate the product of elements of the array in the range L to R (both inclusive) and print the number of zeros
   at the end of the result.
2. **TYPE 2** -> **0 L R V** (where L,R,V are integers)
   For this query you have to set the value of all the elements in the array ranging from L to R (both inclusive) to V.

INPUT :

```
5
1 3 5 8 9
3
1 2 5
0 1 4 10
1 1 5
```

OUTPUT :

```
1
4
```

https://www.codechef.com/problems/COUNZ

We'll use Segment Tree to answer the queries.

To determine the number of zeroes at the end of a number, we should know number of 2’s and 5’s in its prime factorization.

Number Of Zeroes = min(No. of 2’s, No. of 5’s)

Number of Zeroes in product = min(sum of 2’s in elements in product, sum of 5’s in elements in product).

***Be careful of element having value 0 in product as number of zeroes will be 1.**

Since any array element can have value 0 at any point of time we store 3 values at a node in segment tree *sum of number of twos in prime factorization of all the elements in range, sum of number of twos in prime factorization of all the elements in range and number of elements in range with value 0.*

Now, using lazy propagation we can answer queries in **O(log2(Max A[i])+logN) time.**

Total Complexity : O(Nlog2(Max A[i])+Q(log2(Max A[i])+logN))

CODE :

```
void process_node(int node,int num) {
    tree[node][0] = tree[node][1] = tree[node][2] = 0;
    if (num == 0)tree[node][0]++;
    while (num % 2 == 0 && num != 0) {
        num /= 2;
        tree[node][1]++;    //no of 2 in PF of arr[a];
    }
    while (num % 5 == 0 && num != 0) {
        num /= 5;
```

```
        tree[node][2]++;    //no of 5 in PF of arr[a];
    }
    return;
}
void build_tree(int node, int a, int b) {
    if (a == b) {
        int num = arr[a];
        process_node(node, num);
        return;
    }
    int mid = (a + b) >> 1;
    build_tree(node * 2, a, mid);
    build_tree(node * 2 + 1, mid + 1, b);
    for (int i = 0; i < 3;++i)  //build the parent node
        tree[node][i] = tree[node * 2][i] + tree[node * 2
 + 1][i];
}

void update_tree(int node, int a, int b, int i, int j,int
 val) {
    if (lazy[node] != -1) { //if lazy
        process_node(node, lazy[node]);
        for (int i = 0; i < 3; ++i)
            tree[node][i] *= (b - a + 1);
        if (a != b) {   //not a leaf node
            lazy[node * 2] = lazy[node * 2 + 1] = lazy[no
de];   //mark lazy
        }
```

```
        lazy[node] = -1;    //unmark lazy
    }
    if (a > b || a > j || b < i)return;
    if (a >= i && b <= j) {
        process_node(node, val);
        for (int i = 0; i < 3;++i)
            tree[node][i] *= (b - a + 1);
        if (a != b) {
            lazy[node * 2] = lazy[node * 2 + 1] = val;
        }
        return;
    }
    int mid = (a + b) >> 1;
    update_tree(node * 2, a, mid, i, j, val);
    update_tree(node * 2 + 1, mid + 1, b, i, j, val);
    for (int i = 0; i < 3; ++i)
        tree[node][i] = tree[node * 2][i] + tree[node * 2
 + 1][i];
}

void query_tree(int node, int a, int b, int i, int j) {
    if (lazy[node] != -1) { //if lazy
        process_node(node, lazy[node]);
        for (int i = 0; i < 3; ++i)
            tree[node][i] *= (b - a + 1); //multiply valu
e by range times

        if (a != b) {   //if not leaf
```

```
            lazy[node * 2] = lazy[node * 2 + 1] = lazy[no
de];//mark as lazy
        }
        lazy[node] = -1;    //unmark lazy
    }
    if (a > b || a > j || b < i) {
        q_tree[node][0] = q_tree[node][1] = q_tree[node][
2] = 0;
        return;
    }
    if (a >= i && b <= j) {
        for (int i = 0; i < 3;++i)
            q_tree[node][i] = tree[node][i];
        return;
    }
    int mid = (a + b) >> 1;
    query_tree(node * 2, a, mid, i, j);
    query_tree(node * 2 + 1, mid + 1, b, i, j);
    for (int i = 0; i < 3; ++i)
        q_tree[node][i] = q_tree[node * 2][i] + q_tree[no
de * 2 + 1][i];
}
```

PROBLEM : DIVMAC (Dividing Machine)

Chef has created a special dividing machine that supports the below given operations on an array of positive integers.

There are two operations that Chef implemented on the machine.

Type 0 Operation

Update(L,R):

for i = L to R:

a[i] = a[i] / LeastPrimeDivisor(a[i])

Type 1 Operation

Get(L,R):

result = 1

for i = L to R:

result = max(result, LeastPrimeDivisor(a[i]))

return result;

The function **LeastPrimeDivisor(x)** finds the smallest prime divisor of a number. If the number does not have any prime divisors, then it returns **1**.

Chef has provided you an array of size **N**, on which you have to apply **M** operations using the special machine. Each operation will be one of the above given two types. Your task is to implement the special dividing machine operations designed by Chef. Chef finds this task quite easy using his machine, do you too?

INPUT :

```
2
6 7
2 5 8 10 3 44
1 2 6
```

```
0 2 3
1 2 6
0 4 6
1 1 6
0 1 6
1 4 6
2 2
1 3
0 2 2
1 1 2
```

OUTPUT :

```
5 3 5 11
1
```

The problem is :

Given an array of numbers A, support the following two queries:

1. for a given range [L, R], reduce each number in A[L…R] by its smallest prime factor
2. for a given range [L, R], find the number in A[L…R] with largest smallest prime factor.

When we get the range modification entry, we modify each element in the range one by one, and hence perform (R - L + 1) single element update queries. This approach would be to slow, as it makes the complexity of range update query to **O ((R - L + 1) lg N)**

Note that, if the value of an element A[x] is 1, then update query on A[x] does not have any effect on the segment tree. We call such queries as degenerate queries, and can discard them. Also note that a number M can be written as a product of at most **O (lg M)** prime numbers, Hence, we can perform at most **O (lg M)** non-degenerate modification queries on **A[x] = M** all the latter queries will be degenerate. This means that if all numbers in the array are smaller than **M**, then a most **O (N lg M)** non-degenerate single element update queries would be performed. Each single element update query takes **O (lg N)** time, and hence all update queries can be performed in **O (N lg M lg N)** time.

CODE :

```
void modified_sieve() {              //linear time sieve
    lp[1] = 1;
    for (int i = 2; i < maxn; ++i) {
        if (lp[i] == 0) {
            lp[i] = i;
            prime.push_back(i);
        }
        for (int j = 0; j < prime.size() && i*prime[j] <
maxn; ++j) {
            lp[i*prime[j]] = prime[j];
        }
    }
}
```

```
void build_tree(int node, int a, int b) {
    if (a == b) {
        tree[node] = lp[arr[a]];
        ones[node] = (arr[a] == 1 ? 1 : 0);
        return;
    }
    int mid = (a + b) >> 1;
    build_tree(node * 2, a, mid);
    build_tree(node * 2 + 1, mid + 1, b);
    ones[node] = ones[node * 2] + ones[node * 2 + 1];
    tree[node] = max(tree[node * 2], tree[node * 2 + 1]);
}

void update_tree(int node, int a, int b, int i, int j) {
    if (a > b || a > j || b < i)return;
    if (a >= i && b <= j) {
        if (ones[node] == (b - a + 1)) {
            return;
        }
        if (a == b) {   //leaf node
            arr[a] /= lp[arr[a]];
            tree[node] = lp[arr[a]];
            ones[node] = (arr[a] == 1 ? 1 : 0);
            return;
        }
    }
    int mid = (a + b) >> 1;
    update_tree(node * 2, a, mid, i, j);
```

```
    update_tree(node * 2 + 1, mid + 1, b, i, j);
    ones[node] = ones[node * 2] + ones[node * 2 + 1];
    tree[node] = max(tree[node * 2], tree[node * 2 + 1]);
}

int query_tree(int node, int a, int b, int i, int j) {
    if (a > b || a > j || b < i) {
        return INT_MIN;
    }
    if (a >= i && b <= j) {
        return tree[node];
    }
    int mid = (a + b) >> 1;
    int q1 = query_tree(node * 2, a, mid, i, j);
    int q2 = query_tree(node * 2 + 1, mid + 1, b, i, j);
    return q1 > q2 ? q1 : q2;
}
```

Created By *Shubham Rawat*