**Template [New]:**

/\*\*

\* @author : Maruf Tuhin

\* @College : CUET CSE 11

\* @Topcoder : the\_redback

\* @CodeForces : the\_redback

\* @UVA : the\_redback

\* @link : http://www.fb.com/maruf.2hin

\*/

#include <bits/stdc++.h>

using namespace std;

typedef long long ll;

typedef unsigned long long llu;

#define ft first

#define sd second

#define mp make\_pair

#define pb(x) push\_back(x)

#define all(x) x.begin(),x.end()

#define allr(x) x.rbegin(),x.rend()

#define mem(a,b) memset(a,b,sizeof(a))

#define repv(i,a) for(i=0;i<(ll)a.size();i++)

#define revv(i,a) for(i=(ll)a.size()-1;i>=0;i--)

#define rep(i,a,b) for(i=a;i<=b;i++)

#define rev(i,a,b) for(i=a;i>=b;i--)

#define sf(a) scanf("%lld",&a)

#define sf2(a,b) scanf("%lld %lld",&a,&b)

#define sf3(a,b,c) scanf("%lld %lld %lld",&a,&b,&c)

#define inf 1e9

#define eps 1e-9

#define mod 1000000007

#define NN 100010

#ifdef redback

#define bug printf("line=%d\n",\_\_LINE\_\_);

#define debug(args...) {cout<<":: "; dbg,args; cerr<<endl;}

struct debugger{template<typename T>debugger& operator ,(const T& v){cerr<<v<<" ";return \*this;}}dbg;

#else

#define bug

#define debug(args...)

#endif //debugging macros

int main()

{

//ios\_base::sync\_with\_stdio(0); cin.tie(0);

#ifdef redback

freopen("C:\\Users\\Maruf\\Desktop\\in.txt","r",stdin);

#endif

ll t=1,tc;

//sf(tc);

ll l,m,n;

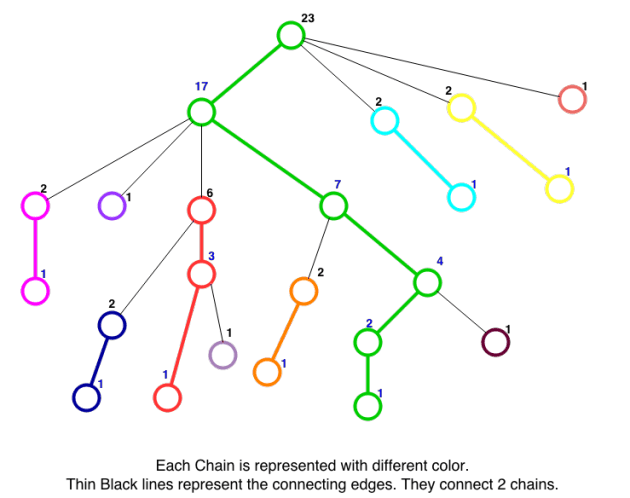
while(~sf(n)) {

ll i,j,k;

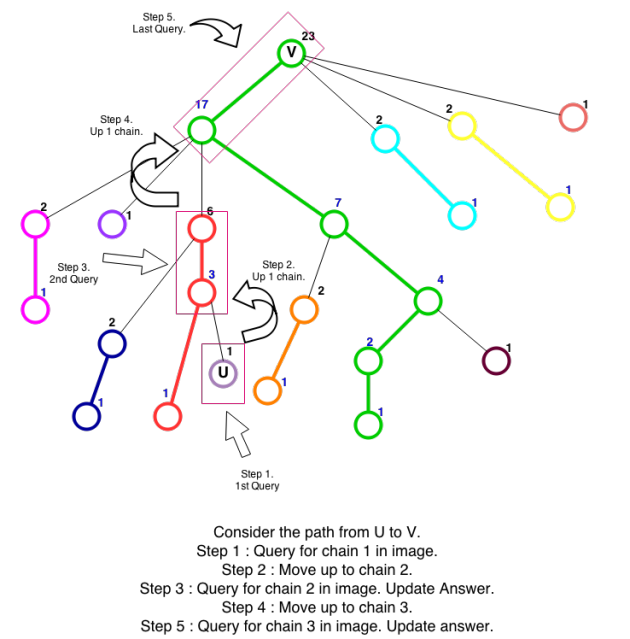
}

return 0;

}

**HLD [Initialization]:**

**HLD [Query]:**



**HLD- with comments [Loj-1348]:**

/\*\*

Problem Description:

you are given a tree (a connected graph with no cycles) with n nodes,

nodes represent places, edges represent roads. In each node,

initially there are an arbitrary number of genies.

But the numbers of genies change in time.

So, you are given a tree, the number of genies in each node and several queries of two types. They are:

1) 0 i j, it means that you have to find the total number of genies in the nodes that occur in path from node i to j (0 <= i, j < n).

2) 1 i v, it means that number of genies in node i is changed to v (0 <= i < n, 0 <= v <= 1000).

\*/

#include <bits/stdc++.h>

using namespace std;

typedef long long ll;

#define NN 50010

#define read(a) scanf("%lld",&a)

#define root 0

#define LN 16

vector <ll> adj[NN];

ll baseArray[NN], ptr, value[NN];

ll chainNo, chainInd[NN], chainHead[NN], posInBase[NN];

ll depth[NN], par[NN][LN], subsize[NN];

ll seg[NN\*4];

/\*

\* make\_tree:

\* Used to construct the segment tree. It uses the baseArray for construction

\*/

void make\_tree(ll node, ll low, ll high)

{

if(low == high)

{

seg[node] = baseArray[low];

return;

}

ll left = node<<1;

ll right = left | 1;

ll mid = (low + high)>>1;

make\_tree(left, low, mid);

make\_tree(right, mid+1, high);

seg[node]=seg[left]+seg[right];

return;

}

/\*

\* update\_tree:

\* Point update. Update a single element of the segment tree.

\*/

void update\_tree(ll node, ll low, ll high, ll ind, ll val)

{

if(low == ind && low == high)

{

seg[node] = val;

return;

}

ll left = node<<1;

ll right = left | 1;

ll mid = (low + high)>>1;

if(ind<=mid)

update\_tree(left, low, mid, ind, val);

else

update\_tree(right, mid + 1, high, ind, val);

seg[node]=seg[left]+seg[right];

return ;

}

/\*

\* query\_tree:

\* Given S and E, it will return the maximum value in the range [S,E)

\*/

ll query\_tree(ll node, ll low, ll high, ll rlow, ll rhigh)

{

if(low>= rlow && high <= rhigh)

{

return seg[node];

}

ll left = node<<1;

ll right = left | 1;

ll mid = (low + high)>>1;

if(rhigh<=mid)

return query\_tree(left, low, mid, rlow, rhigh);

else if(rlow>mid)

return query\_tree(right, mid + 1, high, rlow, rhigh);

else

{

ll L = query\_tree(left, low, mid, rlow, mid);

ll R = query\_tree(right, mid + 1, high, mid + 1, rhigh);

return L+R;

}

}

/\*

\* query\_up:

\* It takes two nodes u and v, condition is that v is an ancestor of u

\* We query the chain in which u is present till chain head, then move to next chain up

\* We do that way till u and v are in the same chain, we query for that part of chain and break

\*/

ll query\_up(ll u, ll v)

{

ll uchain, vchain = chainInd[v], ans = 0;

// uchain and vchain are chain numbers of u and v

while(1)

{

uchain = chainInd[u];

if(uchain == vchain)

{

// Both u and v are in the same chain, so we need to query from u to v, update answer and break.

// We break because we came from u up till v, we are done

//if(u==v) break;

ans+=query\_tree(1, 1, ptr-1, posInBase[v], posInBase[u]);

// Above is call to segment tree query function

break;

}

ans+=query\_tree(1, 1, ptr-1, posInBase[chainHead[uchain]], posInBase[u]);

// Above is call to segment tree query function. We do from chainHead of u till u. That is the whole chain from

// start till head. We then update the answer

u = chainHead[uchain]; // move u to u's chainHead

u = par[u][0]; //Then move to its parent, that means we changed chains

}

return ans;

}

/\*

\* LCA:

\* Takes two nodes u, v and returns Lowest Common Ancestor of u, v

\*/

ll LCA(ll u, ll v)

{

if(depth[u] < depth[v])

swap(u,v);

ll diff = depth[u] - depth[v];

for(ll i=0; i<LN; i++)

if( (diff>>i)&1 )

u = par[u][i];

if(u == v)

return u;

for(ll i=LN-1; i>=0; i--)

if(par[u][i] != par[v][i])

{

u = par[u][i];

v = par[v][i];

}

return par[u][0];

}

ll query(ll u, ll v)

{

/\*

\* We have a query from u to v, we break it into two queries, u to LCA(u,v) and LCA(u,v) to v

\*/

ll lca = LCA(u, v);

ll ans = query\_up(u, lca); // One part of path

ll ans2 = query\_up(v, lca); // another part of path

return ans+ans2-query\_up(lca,lca); // take the maximum of both paths

}

/\*

\* change:

\* We just need to find its position in segment tree and update it

\*/

void change(ll u, ll val)

{

//ll u = otherEnd[i];

update\_tree(1, 1, ptr-1, posInBase[u], val);

}

/\*

\* Actual HL-Decomposition part

\* Initially all entries of chainHead[] are set to -1.

\* So when ever a new chain is started, chain head is correctly assigned.

\* As we add a new node to chain, we will note its position in the baseArray.

\* In the first for loop we find the child node which has maximum sub-tree size.

\* The following if condition is failed for leaf nodes.

\* When the if condition passes, we expand the chain to special child.

\* In the second for loop we recursively call the function on all normal nodes.

\* chainNo++ ensures that we are creating a new chain for each normal child.

\*/

void HLD(ll curNode, ll prev)

{

if(chainHead[chainNo] == -1)

{

chainHead[chainNo] = curNode; // Assign chain head

}

chainInd[curNode] = chainNo;

posInBase[curNode] = ptr; // Position of this node in baseArray which we will use in Segtree

baseArray[ptr++] = value[curNode];

ll sc = -1, ncost;

// Loop to find special child

for(ll i=0; i<adj[curNode].size(); i++)

if(adj[curNode][i] != prev)

{

if(sc == -1 || subsize[sc] < subsize[adj[curNode][i]])

{

sc = adj[curNode][i];

}

}

if(sc != -1)

{

// Expand the chain

HLD(sc, curNode);

}

for(ll i=0; i<adj[curNode].size(); i++)

if(adj[curNode][i] != prev)

{

if(sc != adj[curNode][i])

{

// New chains at each normal node

chainNo++;

HLD(adj[curNode][i], curNode);

}

}

}

/\*

\* dfs used to set parent of a node, depth of a node, subtree size of a node

\*/

void dfs(ll cur, ll prev, ll \_depth=0)

{

par[cur][0] = prev;

depth[cur] = \_depth;

subsize[cur] = 1;

for(ll i=0; i<adj[cur].size(); i++)

if(adj[cur][i] != prev)

{

dfs(adj[cur][i], cur, \_depth+1);

subsize[cur] += subsize[adj[cur][i]];

}

}

int main()

{

ll tc,t=1;

scanf("%lld ", &tc);

while(tc--)

{

ptr = 1;

ll n;

scanf("%lld", &n);

// Cleaning step, new test case

for(ll i=0; i<=n; i++)

{

adj[i].clear();

chainHead[i] = -1;

for(ll j=0; j<LN; j++) par[i][j] = -1;

}

for(ll i=0; i<n; i++)

{

read(value[i]);

}

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

{

ll u, v, c;

scanf("%lld %lld", &u, &v);

adj[u].push\_back(v);

adj[v].push\_back(u);

}

chainNo = 0;

dfs(root, -1); // We set up subsize, depth and parent for each node

HLD(root, -1); // We decomposed the tree and created baseArray

make\_tree(1, 1, ptr-1); // We use baseArray and construct the needed segment tree

// Below Dynamic programming code is for LCA.

for(ll lev = 1; lev <= LN-1; lev++)

{

for(ll i = 0; i < n; i++)

{

if(par[i][lev - 1] != -1)

par[i][lev] = par[par[i][lev - 1]][lev - 1];

}

}

ll q;

scanf("%lld",&q);

printf("Case %lld:\n",t++);

while(q--)

{

ll tp;

scanf("%lld", &tp);

ll a, b;

scanf("%lld %lld", &a, &b);

if(tp==0)

{

ll ans=query(a, b);

printf("%lld\n",ans);

}

else

{

change(a, b);

}

}

}

return 0;

}

/\*

Sample Input

1

4

10 20 30 40

0 1

1 2

1 3

3

0 2 3

1 1 100

0 2 3

Output for Sample Input

Case 1:

90

170

\*/

**HLD- without comments [Loj-1348]:**

#include <bits/stdc++.h>

using namespace std;

typedef long long ll;

#define NN 50010

#define read(a) scanf("%lld",&a)

#define root 0

#define LN 16

vector <ll> adj[NN];

ll baseArray[NN], ptr, value[NN];

ll chainNo, chainInd[NN], chainHead[NN], posInBase[NN];

ll depth[NN], par[NN][LN], subsize[NN];

ll seg[NN\*4];

void make\_tree(ll node, ll low, ll high)

{

if(low == high)

{

seg[node] = baseArray[low];

return;

}

ll left = node<<1;

ll right = left | 1;

ll mid = (low + high)>>1;

make\_tree(left, low, mid);

make\_tree(right, mid+1, high);

seg[node]=seg[left]+seg[right];

return;

}

void update\_tree(ll node, ll low, ll high, ll ind, ll val)

{

if(low == ind && low == high)

{

seg[node] = val;

return;

}

ll left = node<<1;

ll right = left | 1;

ll mid = (low + high)>>1;

if(ind<=mid)

update\_tree(left, low, mid, ind, val);

else

update\_tree(right, mid + 1, high, ind, val);

seg[node]=seg[left]+seg[right];

return ;

}

ll query\_tree(ll node, ll low, ll high, ll rlow, ll rhigh){

if(low>= rlow && high <= rhigh)

{

return seg[node];

}

ll left = node<<1;

ll right = left | 1;

ll mid = (low + high)>>1;

if(rhigh<=mid)

return query\_tree(left, low, mid, rlow, rhigh);

else if(rlow>mid)

return query\_tree(right, mid + 1, high, rlow, rhigh);

else

{

ll L = query\_tree(left, low, mid, rlow, mid);

ll R = query\_tree(right, mid + 1, high, mid + 1, rhigh);

return L+R;

}

}

ll query\_up(ll u, ll v) //v is an ancestor of u

{

ll uchain, vchain = chainInd[v], ans = 0;

// uchain and vchain are chain numbers of u and v

while(1)

{

uchain = chainInd[u];

if(uchain == vchain)

{

ans+=query\_tree(1, 1, ptr-1, posInBase[v], posInBase[u]);

break;

}

ans+=query\_tree(1, 1, ptr-1, posInBase[chainHead[uchain]], posInBase[u]);

u = chainHead[uchain]; // move u to u's chainHead

u = par[u][0]; //Then move to its parent, that means we changed chains

}

return ans;

}

ll LCA(ll u, ll v)

{

if(depth[u] < depth[v])

swap(u,v);

ll diff = depth[u] - depth[v];

for(ll i=0; i<LN; i++)

if( (diff>>i)&1 )

u = par[u][i];

if(u == v)

return u;

for(ll i=LN-1; i>=0; i--)

if(par[u][i] != par[v][i])

{

u = par[u][i];

v = par[v][i];

}

return par[u][0];

}

ll query(ll u, ll v){

ll lca = LCA(u, v);

ll ans = query\_up(u, lca); // One part of path

ll ans2 = query\_up(v, lca); // another part of path

return ans+ans2-query\_up(lca,lca); // take the maximum of both paths

}

void change(ll u, ll val){

update\_tree(1, 1, ptr-1, posInBase[u], val);

}

void HLD(ll curNode, ll prev)

{

if(chainHead[chainNo] == -1)

{

chainHead[chainNo] = curNode; // Assign chain head

}

chainInd[curNode] = chainNo;

posInBase[curNode] = ptr;

baseArray[ptr++] = value[curNode];

ll sc = -1, ncost;

// Loop to find special child

for(ll i=0; i<adj[curNode].size(); i++)

if(adj[curNode][i] != prev)

{

if(sc == -1 || subsize[sc] < subsize[adj[curNode][i]])

{

sc = adj[curNode][i];

}

}

if(sc != -1)

{

HLD(sc, curNode); // Expand the chain

}

for(ll i=0; i<adj[curNode].size(); i++)

if(adj[curNode][i] != prev)

{

if(sc != adj[curNode][i])

{

// New chains at each normal node

chainNo++;

HLD(adj[curNode][i], curNode);

}

}

}

void dfs(ll cur, ll prev, ll \_depth=0)

{

par[cur][0] = prev;

depth[cur] = \_depth;

subsize[cur] = 1;

for(ll i=0; i<adj[cur].size(); i++)

if(adj[cur][i] != prev)

{

dfs(adj[cur][i], cur, \_depth+1);

subsize[cur] += subsize[adj[cur][i]];

}

}

int main() {

ll tc,t=1;

scanf("%lld ", &tc);

while(tc--) {

ptr = 1;

ll n;

scanf("%lld", &n);

for(ll i=0; i<=n; i++) {

adj[i].clear();

chainHead[i] = -1;

for(ll j=0; j<LN; j++) par[i][j] = -1;

}

for(ll i=0; i<n; i++) {

read(value[i]);

}

for(ll i=1; i<n; i++) {

ll u, v, c;

scanf("%lld %lld", &u, &v);

adj[u].push\_back(v);

adj[v].push\_back(u);

}

chainNo = 0;

dfs(root, -1); // We set up subsize, depth and parent for each node

HLD(root, -1); // We decomposed the tree and created baseArray

make\_tree(1, 1, ptr-1); //We use baseArray and construct the needed segment tree

// Below Dynamic programming code is for LCA.

for(ll lev = 1; lev <= LN-1; lev++) {

for(ll i = 0; i < n; i++)

{

if(par[i][lev - 1] != -1)

par[i][lev] = par[par[i][lev - 1]][lev - 1];

}

}

ll q;

scanf("%lld",&q);

printf("Case %lld:\n",t++);

while(q--) {

ll tp;

scanf("%lld", &tp);

ll a, b;

scanf("%lld %lld", &a, &b);

if(tp==0)

{

ll ans=query(a, b);

printf("%lld\n",ans);

}

else

{

change(a, b);

}

}

}

return 0;

}

**HLD [Anudeep] Spoj-QTREE:**

/\*

You are given a tree (an acyclic undirected connected graph) with N nodes,

and edges numbered 1, 2, 3...N-1.

We will ask you to perfrom some instructions of the following form:

1.CHANGE i ti : change the cost of the i-th edge to ti

2.QUERY a b : ask for the maximum edge cost on the path from node a to node b

\*/

#include <cstdio>

#include <vector>

using namespace std;

#define root 0

#define N 10100

#define LN 14

vector <int> adj[N], costs[N], indexx[N];

int baseArray[N], ptr;

int chainNo, chainInd[N], chainHead[N], posInBase[N];

int depth[N], pa[LN][N], otherEnd[N], subsize[N];

int st[N\*6], qt[N\*6];

void make\_tree(int cur, int s, int e) {

if(s == e-1) {

st[cur] = baseArray[s];

return;

}

int c1 = (cur<<1), c2 = c1 | 1, m = (s+e)>>1;

make\_tree(c1, s, m);

make\_tree(c2, m, e);

st[cur] = st[c1] > st[c2] ? st[c1] : st[c2];

}

void update\_tree(int cur, int s, int e, int x, int val) {

if(s > x || e <= x) return;

if(s == x && s == e-1) {

st[cur] = val;

return;

}

int c1 = (cur<<1), c2 = c1 | 1, m = (s+e)>>1;

update\_tree(c1, s, m, x, val);

update\_tree(c2, m, e, x, val);

st[cur] = st[c1] > st[c2] ? st[c1] : st[c2];

}

void query\_tree(int cur, int s, int e, int S, int E) {

if(s >= E || e <= S) {

qt[cur] = -1;

return;

}

if(s >= S && e <= E) {

qt[cur] = st[cur];

return;

}

int c1 = (cur<<1), c2 = c1 | 1, m = (s+e)>>1;

query\_tree(c1, s, m, S, E);

query\_tree(c2, m, e, S, E);

qt[cur] = qt[c1] > qt[c2] ? qt[c1] : qt[c2];

}

int query\_up(int u, int v) {

if(u == v) return 0; // Trivial

int uchain, vchain = chainInd[v], ans = -1;

// uchain and vchain are chain numbers of u and v

while(1) {

uchain = chainInd[u];

if(uchain == vchain) {

// Both u and v are in the same chain, so we need to query from u to v, update answer and break.

// We break because we came from u up till v, we are done

if(u==v) break;

query\_tree(1, 0, ptr, posInBase[v]+1, posInBase[u]+1);

// Above is call to segment tree query function

if(qt[1] > ans) ans = qt[1]; // Update answer

break;

}

query\_tree(1, 0, ptr, posInBase[chainHead[uchain]], posInBase[u]+1);

// Above is call to segment tree query function. We do from chainHead of u till u. That is the whole chain from

// start till head. We then update the answer

if(qt[1] > ans) ans = qt[1];

u = chainHead[uchain]; // move u to u's chainHead

u = pa[0][u]; //Then move to its parent, that means we changed chains

}

return ans;

}

int LCA(int u, int v) {

if(depth[u] < depth[v]) swap(u,v);

int diff = depth[u] - depth[v];

for(int i=0; i<LN; i++) if( (diff>>i)&1 ) u = pa[i][u];

if(u == v) return u;

for(int i=LN-1; i>=0; i--) if(pa[i][u] != pa[i][v]) {

u = pa[i][u];

v = pa[i][v];

}

return pa[0][u];

}

void query(int u, int v) {

int lca = LCA(u, v);

int ans = query\_up(u, lca); // One part of path

int temp = query\_up(v, lca); // another part of path

if(temp > ans) ans = temp; // take the maximum of both paths

printf("%d\n", ans);

}

void change(int i, int val) {

int u = otherEnd[i];

update\_tree(1, 0, ptr, posInBase[u], val);

}

void HLD(int curNode, int cost, int prev) {

if(chainHead[chainNo] == -1) {

chainHead[chainNo] = curNode; // Assign chain head

}

chainInd[curNode] = chainNo;

posInBase[curNode] = ptr; // Position of this node in baseArray which we will use in Segtree

baseArray[ptr++] = cost;

int sc = -1, ncost;

// Loop to find special child

for(int i=0; i<adj[curNode].size(); i++) if(adj[curNode][i] != prev) {

if(sc == -1 || subsize[sc] < subsize[adj[curNode][i]]) {

sc = adj[curNode][i];

ncost = costs[curNode][i];

}

}

if(sc != -1) {

// Expand the chain

HLD(sc, ncost, curNode);

}

for(int i=0; i<adj[curNode].size(); i++) if(adj[curNode][i] != prev) {

if(sc != adj[curNode][i]) {

// New chains at each normal node

chainNo++;

HLD(adj[curNode][i], costs[curNode][i], curNode);

}

}

}

void dfs(int cur, int prev, int \_depth=0) {

pa[0][cur] = prev;

depth[cur] = \_depth;

subsize[cur] = 1;

for(int i=0; i<adj[cur].size(); i++)

if(adj[cur][i] != prev) {

otherEnd[indexx[cur][i]] = adj[cur][i];

dfs(adj[cur][i], cur, \_depth+1);

subsize[cur] += subsize[adj[cur][i]];

}

}

int main() {

int t;

scanf("%d ", &t);

while(t--) {

ptr = 0;

int n;

scanf("%d", &n);

// Cleaning step, new test case

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

adj[i].clear();

costs[i].clear();

indexx[i].clear();

chainHead[i] = -1;

for(int j=0; j<LN; j++) pa[j][i] = -1;

}

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

int u, v, c;

scanf("%d %d %d", &u, &v, &c);

u--; v--;

adj[u].push\_back(v);

costs[u].push\_back(c);

indexx[u].push\_back(i-1);

adj[v].push\_back(u);

costs[v].push\_back(c);

indexx[v].push\_back(i-1);

}

chainNo = 0;

dfs(root, -1); // We set up subsize, depth and parent for each node

HLD(root, -1, -1); // We decomposed the tree and created baseArray

make\_tree(1, 0, ptr); // We use baseArray and construct the needed segment tree

// Below Dynamic programming code is for LCA.

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

for(int j=0; j<n; j++)

if(pa[i-1][j] != -1)

pa[i][j] = pa[i-1][pa[i-1][j]];

while(1) {

char s[100];

scanf("%s", s);

if(s[0]=='D') {

break;

}

int a, b;

scanf("%d %d", &a, &b);

if(s[0]=='Q') {

query(a-1, b-1);

} else {

change(a-1, b);

}

}

}

}

int query\_up(int u, int v)

{

if(u == v) return 0; // Trivial

int uchain, vchain = chainInd[v], ans = -1;

// uchain and vchain are chain numbers of u and v

while(1)

{

uchain = chainInd[u];

if(uchain == vchain)

{

// Both u and v are in the same chain, so we need to query from u to v,

// update answer and break.

// We break because we came from u up till v, we are done

if(u==v)

break;

query\_tree(1, 0, ptr, posInBase[v]+1, posInBase[u]+1);

// Above is call to segment tree query function

if(qt[1] > ans)

ans = qt[1]; // Update answer

break;

}

query\_tree(1, 0, ptr, posInBase[chainHead[uchain]], posInBase[u]+1);

// Above is call to segment tree query function. We do from chainHead of u

// till u. That is the whole chain from

// start till head. We then update the answer

if(qt[1] > ans)

ans = qt[1];

u = chainHead[uchain]; // move u to u's chainHead

u = pa[0][u]; //Then move to its parent, that means we changed chains

}

return ans;

}

int LCA(int u, int v)

{

if(depth[u] < depth[v]) swap(u,v);

int diff = depth[u] - depth[v];

for(int i=0; i<LN; i++) if( (diff>>i)&1 ) u = pa[i][u];

if(u == v) return u;

for(int i=LN-1; i>=0; i--)

if(pa[i][u] != pa[i][v])

{

u = pa[i][u];

v = pa[i][v];

}

return pa[0][u];

}

void query(int u, int v)

{

int lca = LCA(u, v);

int ans = query\_up(u, lca); // One part of path

int temp = query\_up(v, lca); // another part of path

if(temp > ans) ans = temp; // take the maximum of both paths

printf("%d\n", ans);

}

void change(int i, int val)

{

int u = otherEnd[i];

update\_tree(1, 0, ptr, posInBase[u], val);

}

void HLD(int curNode, int cost, int prev) {

if(chainHead[chainNo] == -1) {

chainHead[chainNo] = curNode; // Assign chain head

}

chainInd[curNode] = chainNo;

posInBase[curNode] = ptr; // Position of this node in baseArray which we will use in Segtree

baseArray[ptr++] = cost;

int sc = -1, ncost;

// Loop to find special child

for(int i=0; i<adj[curNode].size(); i++) if(adj[curNode][i] != prev) {

if(sc == -1 || subsize[sc] < subsize[adj[curNode][i]]) {

sc = adj[curNode][i];

ncost = costs[curNode][i];

}

}

if(sc != -1) {

// Expand the chain

HLD(sc, ncost, curNode);

}

for(int i=0; i<adj[curNode].size(); i++) if(adj[curNode][i] != prev) {

if(sc != adj[curNode][i]) {

// New chains at each normal node

chainNo++;

HLD(adj[curNode][i], costs[curNode][i], curNode);

}

}

}

void dfs(int cur, int prev, int \_depth=0) {

pa[0][cur] = prev;

depth[cur] = \_depth;

subsize[cur] = 1;

for(int i=0; i<adj[cur].size(); i++)

if(adj[cur][i] != prev) {

otherEnd[indexx[cur][i]] = adj[cur][i];

dfs(adj[cur][i], cur, \_depth+1);

subsize[cur] += subsize[adj[cur][i]];

}

}

int main() {

int t;

scanf("%d ", &t);

while(t--) {

ptr = 0;

int n;

scanf("%d", &n);

// Cleaning step, new test case

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

adj[i].clear();

costs[i].clear();

indexx[i].clear();

chainHead[i] = -1;

for(int j=0; j<LN; j++) pa[j][i] = -1;

}

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

int u, v, c;

scanf("%d %d %d", &u, &v, &c);

u--; v--;

adj[u].push\_back(v);

costs[u].push\_back(c);

indexx[u].push\_back(i-1);

adj[v].push\_back(u);

costs[v].push\_back(c);

indexx[v].push\_back(i-1);

}

chainNo = 0;

dfs(root, -1); // We set up subsize, depth and parent for each node

HLD(root, -1, -1); // We decomposed the tree and created baseArray

make\_tree(1, 0, ptr); // We use baseArray and construct the needed segment tree

// Below Dynamic programming code is for LCA.

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

for(int j=0; j<n; j++)

if(pa[i-1][j] != -1)

pa[i][j] = pa[i-1][pa[i-1][j]];

while(1) {

char s[100];

scanf("%s", s);

if(s[0]=='D') {

break;

}

int a, b;

scanf("%d %d", &a, &b);

if(s[0]=='Q') {

query(a-1, b-1);

} else {

change(a-1, b);

}

}

}

}

void HLD(int curNode, int cost, int prev) {

if(chainHead[chainNo] == -1) {

chainHead[chainNo] = curNode; // Assign chain head

}

chainInd[curNode] = chainNo;

posInBase[curNode] = ptr; // Position of this node in baseArray which we will use

// in Segtree

baseArray[ptr++] = cost;

int sc = -1, ncost;

// Loop to find special child

for(int i=0; i<adj[curNode].size(); i++) if(adj[curNode][i] != prev) {

if(sc == -1 || subsize[sc] < subsize[adj[curNode][i]]) {

sc = adj[curNode][i];

ncost = costs[curNode][i];

}

}

if(sc != -1) {

HLD(sc, ncost, curNode); // Expand the chain

}

for(int i=0; i<adj[curNode].size(); i++) if(adj[curNode][i] != prev) {

if(sc != adj[curNode][i]) {

// New chains at each normal node

chainNo++;

HLD(adj[curNode][i], costs[curNode][i], curNode);

}

}

}

void dfs(int cur, int prev, int \_depth=0) {

pa[0][cur] = prev;

depth[cur] = \_depth;

subsize[cur] = 1;

for(int i=0; i<adj[cur].size(); i++)

if(adj[cur][i] != prev) {

otherEnd[indexx[cur][i]] = adj[cur][i];

dfs(adj[cur][i], cur, \_depth+1);

subsize[cur] += subsize[adj[cur][i]];

}

}

int main() {

int t;

scanf("%d ", &t);

while(t--) {

ptr = 0;

int n;

scanf("%d", &n);

// Cleaning step, new test case

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

adj[i].clear();

costs[i].clear();

indexx[i].clear();

chainHead[i] = -1;

for(int j=0; j<LN; j++) pa[j][i] = -1;

}

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

int u, v, c;

scanf("%d %d %d", &u, &v, &c);

u--; v--;

adj[u].push\_back(v);

costs[u].push\_back(c);

indexx[u].push\_back(i-1);

adj[v].push\_back(u);

costs[v].push\_back(c);

indexx[v].push\_back(i-1);

}

chainNo = 0;

dfs(root, -1); // We set up subsize, depth and parent for each node

HLD(root, -1, -1); // We decomposed the tree and created baseArray

make\_tree(1, 0, ptr); // We use baseArray and construct the needed segment tree

// Below Dynamic programming code is for LCA.

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

for(int j=0; j<n; j++)

if(pa[i-1][j] != -1)

pa[i][j] = pa[i-1][pa[i-1][j]];

while(1) {

char s[100];

scanf("%s", s);

if(s[0]=='D') {

break;

}

int a, b;

scanf("%d %d", &a, &b);

if(s[0]=='Q') {

query(a-1, b-1);

} else {

change(a-1, b);

}

}

}

}

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

int u, v, c;

scanf("%d %d %d", &u, &v, &c);

u--; v--;

adj[u].push\_back(v);

costs[u].push\_back(c);

indexx[u].push\_back(i-1);

adj[v].push\_back(u);

costs[v].push\_back(c);

indexx[v].push\_back(i-1);

}

chainNo = 0;

dfs(root, -1); // We set up subsize, depth and parent for each node

HLD(root, -1, -1); // We decomposed the tree and created baseArray

make\_tree(1, 0, ptr);//We use baseArray and construct the needed segment tree

// Below Dynamic programming code is for LCA.

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

for(int j=0; j<n; j++)

if(pa[i-1][j] != -1)

pa[i][j] = pa[i-1][pa[i-1][j]];

while(1) {

char s[100];

scanf("%s", s);

if(s[0]=='D') {

break;

}

int a, b;

scanf("%d %d", &a, &b);

if(s[0]=='Q') {

query(a-1, b-1);

} else {

change(a-1, b);

}

}

}

}

Input:

1

3

1 2 1

2 3 2

QUERY 1 2

CHANGE 1 3

QUERY 1 2

DONE

Output:

1

3

**Segmented Sieve [Prime] :**

#define MAX 46656

#define LMT 216

#define LEN 4830

#define RNG 100032

unsigned base[MAX/64], segment[RNG/64], primes[LEN];

#define sq(x) ((x)\*(x))

#define mset(x,v) memset(x,v,sizeof(x))

#define chkC(x,n) (x[n>>6]&(1<<((n>>1)&31)))

#define setC(x,n) (x[n>>6]|=(1<<((n>>1)&31)))

/\* Generates all the necessary prime

numbers and marks them in base[]\*/

void sieve()

{

unsigned i, j, k;

for(i=3; i<LMT; i+=2)

if(!chkC(base, i))

for(j=i\*i,k=i<<1; j<MAX; j+=k)

setC(base, j);

for(i=3, j=0; i<MAX; i+=2)

if(!chkC(base, i))

primes[j++] = i;

}

/\* Returns the prime-count within range

[a,b] and marks them in segment[]

\*/

int segmented\_sieve(int a, int b)

{

unsigned i, j, k;

unsigned cnt=(a<=2 && 2<=b)? 1 : 0;

if(b<2) return 0;

if(a<3) a = 3;

if(a%2==0) a++;

mset(segment,0);

for(i=0; sq(primes[i])<=b; i++)

{

j = primes[i] \*

((a+primes[i]-1) / primes[i]);

if(j%2==0) j += primes[i];

for(k=primes[i]<<1; j<=b; j+=k)

if(j!=primes[i])

setC(segment, (j-a));

}

for(i=0; i<=b-a; i+=2)

if(!chkC(segment, i))

cnt++;

return cnt;

}

**Catalan Number :**

\*\* Formula:

catalan.png

Recursive:

#include<iostream>

using namespace std;

unsigned long catalan(unsigned int n)

{

if (n <= 1) return 1;

unsigned long int res = 0;

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

res += catalan(i)\*catalan(n-i-1);

return res;

}

// Driver program to test above function

int main()

{

for (int i=0; i<10; i++)

cout << catalan(i) << " ";

return 0;

}

DP:

#include<iostream>

using namespace std;

unsigned long catalanDP(unsigned int n)

{

unsigned long int catalan[n+1];

catalan[0] = catalan[1] = 1;

for (int i=2; i<=n; i++)

{

catalan[i] = 0;

for (int j=0; j<i; j++)

catalan[i] +=

catalan[j] \* catalan[i-j-1];

}

return catalan[n];

}

int main()

{

for (int i = 0; i < 10; i++)

cout << catalanDP(i) << " ";

return 0;

}

Output: 1 1 2 5 14 42 132 429 1430 4862

Complexity:O(n2)

**Segment Tree-Computing Fast Average [Loj-1183]:**

/\*

Problem Description:

Given an array of integers (0 indexed),

you have to perform two types of queries in the array.

1. 1 i j v - change the value of the elements from ith index to jth index to v.

2. 2 i j - find the average value of the integers from ith index to jth index.

You can assume that initially all the values in the array are 0.

\*/

#include <bits/stdc++.h>

using namespace std;

typedef long long ll;

#define read(a) scanf("%lld",&a)

struct data {

ll sum , xtra;

}tree[300010];

void init(ll node, ll low, ll high) {

if(low==high) {

tree[node].sum=0;

tree[node].xtra=-1;

return;

}

ll left = node\*2;

ll right = left + 1;

ll mid = (low + high)/2;

init(left, low, mid);

init(right, mid + 1, high);

tree[node].sum = tree[left].sum+tree[right].sum;

tree[node].xtra=-1;

return;

}

void update(ll node, ll low, ll high, ll rlow, ll rhigh, ll value) {

if(low>=rlow && high<=rhigh) {

tree[node].sum = (high-low+1)\*value;

tree[node].xtra = value;

return;

}

ll left = node\*2;

ll right = left+1;

ll mid = (low+high)/2;

if(tree[node].xtra!=-1) {

tree[left].xtra=tree[node].xtra;

tree[right].xtra=tree[node].xtra;

tree[left].sum=(mid-low+1)\*tree[left].xtra;

tree[right].sum=(high-mid)\*tree[right].xtra;

tree[node].xtra=-1;

}

if(rhigh <= mid) update(left, low, mid, rlow, rhigh, value);

else if(rlow > mid) update(right, mid+1, high, rlow, rhigh, value);

else {

update(left, low, mid, rlow, mid, value);

update(right, mid+1, high, mid+1, rhigh, value);

}

tree[node].sum = tree[left].sum+ tree[right].sum;

}

ll query(ll node, ll low, ll high, ll rlow, ll rhigh, ll carry)

{

if(carry!=-1)

{

return (rhigh-rlow+1)\*carry;

}

if(low>=rlow && high<=rhigh)

{

return tree[node].sum;

}

ll left = node\*2;

ll right = left + 1;

ll mid = (low + high)/2;

ll p1=0, p2=0;

if((high-low+1)\*tree[node].xtra == tree[node].sum )

carry=tree[node].xtra;

if(rhigh<=mid)

p1=query(left, low, mid, rlow, rhigh, carry);

else if(rlow>mid)

p2=query(right, mid+1, high, rlow, rhigh, carry);

else

{

p1=query(left, low, mid, rlow, mid, carry);

p2=query(right, mid+1, high, mid+1, rhigh, carry);

}

return p1+p2;

}

main()

{

ll tc, t=1;

cin>>tc;

while(tc--)

{

ll n, q;

cin>>n>>q;

printf("Case %d:\n", t++);

init(1,1,n);

while(q--)

{

ll i, j, k, l;

cin>>i;

if(i==1)

{

cin>>j>>k>>l;

update(1, 1, n, j+1, k+1, l);

}

else if(i==2)

{

cin>>j>>k;

ll ans=query(1, 1, n, j+1, k+1, -1);

ll res=(k-j+1);

ll gcd=\_\_gcd(res,ans);

if(res/gcd>1)

printf("%lld/%lld\n", ans/gcd,res/gcd );

else

printf("%lld\n", ans/gcd);

}

}

}

return 0;

}

ll query(ll node, ll low, ll high, ll rlow, ll rhigh, ll carry)

{

if(carry!=-1) {

return (rhigh-rlow+1)\*carry;

}

if(low>=rlow && high<=rhigh) {

return tree[node].sum;

}

ll left = node\*2;

ll right = left + 1;

ll mid = (low + high)/2;

ll p1=0, p2=0;

if((high-low+1)\*tree[node].xtra == tree[node].sum )

carry=tree[node].xtra;

if(rhigh<=mid) p1=query(left, low, mid, rlow, rhigh, carry);

else if(rlow>mid) p2=query(right, mid+1, high, rlow, rhigh, carry);

else {

p1=query(left, low, mid, rlow, mid, carry);

p2=query(right, mid+1, high, mid+1, rhigh, carry);

}

return p1+p2;

}

main()

{

ll tc, t=1;

cin>>tc;

while(tc--)

{

ll n, q;

cin>>n>>q;

printf("Case %d:\n", t++);

init(1,1,n);

while(q--)

{

ll i, j, k, l;

cin>>i;

if(i==1)

{

cin>>j>>k>>l;

update(1, 1, n, j+1, k+1, l);

}

else if(i==2)

{

cin>>j>>k;

ll ans=query(1, 1, n, j+1, k+1, -1);

ll res=(k-j+1);

ll gcd=\_\_gcd(res,ans);

if(res/gcd>1)

printf("%lld/%lld\n", ans/gcd,res/gcd );

else

printf("%lld\n", ans/gcd);

}

}

}

return 0;

}

Sample Input:

1

10 6

1 0 6 6

2 0 1

1 1 1 2

2 0 5

1 0 3 7

2 0 1

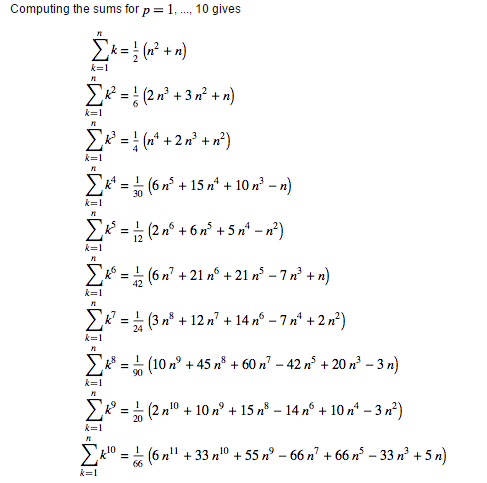
Output for Sample Input:

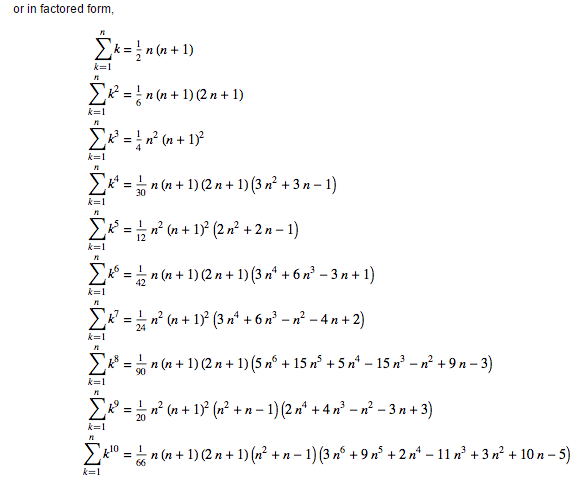
Case 1:

6

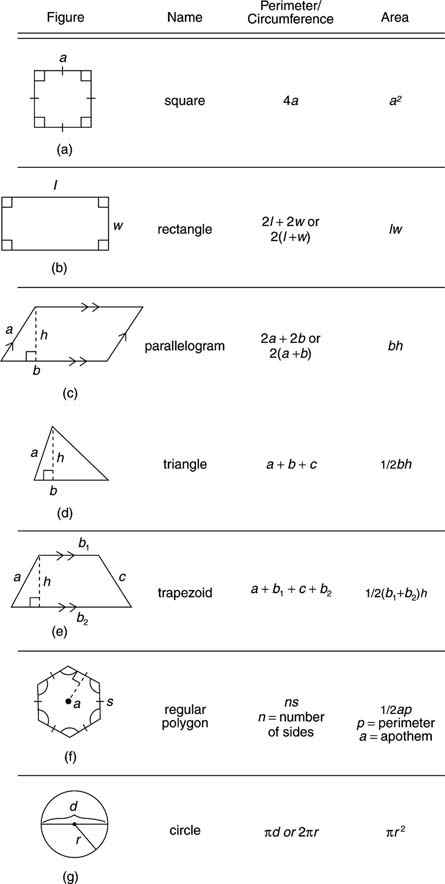
16/3

7

**Sum of Series:**

****

**Geometry Areas:**

****