**Data Structure Library**

1. Trie

//Implementation of Trie Data structure

#define MAX\_TRIE 28 // the type of the trie how much it can take

#define BASE 48 //which will value will be subtracted to make a proper array index

struct trieInfo{

int prefix; // how many prefix

int word; //how many word

struct trieInfo \*next[MAX\_TRIE+5]; // next trieInfo location

};

struct trieInfo\* createTrieNode() // creating the trie nodes space allocation

{

struct trieInfo \*temp = (struct trieInfo\*)malloc(sizeof(struct trieInfo));

temp->prefix=0;

temp->word=0;

for(int i=0;i<MAX\_TRIE;i++)

{

temp->next[i] = NULL;

}

return temp;

}

void addWord(struct trieInfo \*trieNode,string s, int idx) // adding the words

{

if(idx>=s.length()) // when no more call can be done with the next character

{

trieNode->prefix = trieNode->prefix+1;

trieNode->word = trieNode->word + 1;

return;

}

else

{

int firstCharacter = s[idx] - BASE;

trieNode->prefix = trieNode->prefix+1;

if(trieNode->next[firstCharacter] == NULL)

{

struct trieInfo \*temp = createTrieNode();

trieNode->next[firstCharacter] = temp;

}

addWord(trieNode->next[firstCharacter],s,idx+1);

}

return;

}

int countPrefix(struct trieInfo \*trieNode,string s, int idx) // counting the prefix number

{

if(idx>=s.length())

{

return trieNode->prefix;

}

else

{

int firstCharacter = s[idx] - BASE;

if(trieNode->next[firstCharacter] == NULL)

{

return 0;

}

else

{

return countPrefix(trieNode->next[firstCharacter],s,idx+1);

}

}

}

int countWord(struct trieInfo \*trieNode,string s, int idx)

{

if(idx>=s.length())

{

return trieNode->word;

}

else

{

int firstCharacter = s[idx] - BASE;

if(trieNode->next[firstCharacter] == NULL)

{

return 0;

}

else

{

return countWord(trieNode->next[firstCharacter],s,idx+1);

}

}

}

void deleteTrieNode(struct trieInfo \*trieNode)

{

for(int i=0;i<MAX\_TRIE;i++)

{

if(trieNode->next[i] != NULL)

{

deleteTrieNode(trieNode->next[i]);

}

}

free(trieNode);

}

**//Binary Index Tree**

typedef long long int ll;

ll bitTreeOne[MAX+5];

ll answer[MAX+5];

ll arr[MAX+5];

struct info{

int x,y,value;

}queries[MAX+5];

// the query part

//simply searching the previous range from where value should be taken

ll bitTreeQuery(int idx, ll bitTree[])

{

ll sum = 0;

while(idx>0)

{

sum += bitTree[idx];

idx = idx - (idx & (-idx)); // the previous sub index

}

return sum;

}

//the update part

//simply searching the next index value should be added

void bitTreeUpdate(int idx,ll value,int n,ll bitTree[])

{

while(idx<=n)

{

bitTree[idx] += value;

idx = idx + (idx & (-idx)); // the next sub array index

}

}

// the initialization part

void bitTreeInitialization(ll bitTree[], int n)

{

memset(bitTree,0,sizeof(bitTree));

}

//Heavy Light Decomposition

vector<int>graph[MAX\_HLD+5];

//Rizvee's Heavy Light Decomposition

int cost[MAX\_HLD+5]; //saves the cost of each node

int visitHLD[MAX\_HLD+5]; //visit array in HLD

int subTreeSize[MAX\_HLD+5]; //sub tree size in HLD per node

int chainNo; //how many chains are present in the graph

int chainHead[MAX\_HLD+5]; //says the head of each chain number

int chainPos[MAX\_HLD+5]; //says the nodes position in the chain

int chainInd[MAX\_HLD+5]; //each node belongs to which chain

int chainSize[MAX\_HLD+5]; //size of each chain

/\*segment tree part things\*/

int posInBase[MAX\_HLD+5]; //the position of node in the baseArray on which segment tree operation will be performed

int baseArray[MAX\_HLD+5]; //the value will be saved

int tree[6\*MAX\_HLD+5]; //the structure to be saved

int ptr; // the pointer in base array

/\*LCA part things \*/

int sparseTable[MAX\_HLD+5][20];

int parent[MAX\_HLD+5]; //parent of each node

int level[MAX\_HLD+5]; //level of each node

int maxPower; //max jth node

//the main array is the baseArray on which we will build our segment tree and

//save it in the tree[] array

void segmentTreeMake(int idx,int st,int en)

{

if(st == en)

{

tree[idx] = baseArray[st];

return;

}

int mid = (st+en)/2;

int left = 2 \* idx; //left sub part

int right = 2 \* idx + 1; //right sub part

segmentTreeMake(left,st,mid);

segmentTreeMake(right,mid+1,en);

tree[idx] = tree[left]+tree[right];

return;

}

//update the segment tree

void segmentTreeUpdate(int idx,int st, int en, int x,int value)

{

//cout<<"update e " << st <<" " << en << endl;

if(x<st||x>en) return;

if(st == en && st == x)

{

//cout<<"dhuke " << st << " " << en << " " << tree[st] << endl;

tree[idx] = value;

return;

}

int mid = (st+en)/2;

int left = 2 \* idx; //left sub part

int right = 2 \* idx + 1; //right sub part

segmentTreeUpdate(left,st,mid,x,value);

segmentTreeUpdate(right,mid+1,en,x,value);

tree[idx] = tree[left]+tree[right];

return;

}

//query the segment tree

int segmentTreeQuery(int idx, int st,int en,int x, int y)

{

//cout<<"st = " <<st <<" "<<en <<endl;

if(y<st || x>en) return 0;

if(x<=st && y>=en)

{

return tree[idx];

}

int mid = (st+en)/2;

int left = 2 \* idx; //left sub part

int right = 2 \* idx + 1; //right sub part

int sum1 = segmentTreeQuery(left,st,mid,x,y);

int sum2 = segmentTreeQuery(right,mid+1,en,x,y);

return sum1+sum2;

}

//the special LCA part

void LCA(int n) //the total number of nodes is n

{

maxPower = -1;

//2^0 th parent

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

{

sparseTable[i][0] = parent[i];

maxPower = 0;

}

//other parents

for(int j=0;(1<<j)<=n;j++)

{

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

{

if((j-1)<=17 && sparseTable[i][j-1] != -1)

{

sparseTable[i][j] = sparseTable[sparseTable[i][j-1]][j-1]; // 2^i = 2 \* 2 ^(i-1)

maxPower = max(maxPower,j); //taking the max jth power highest 2^j th node remains there

}

}

}

return;

}

//this function determines the LCA among two nodes

int lcaQuery(int u, int v)

{

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

{

swap(u,v); // u is in the lowset level and v is in the upper level

}

for(int i=maxPower;i>=0;i--)

{

if((level[u] - (1<<i))>=level[v])

{

u = sparseTable[u][i]; // k = 2^p + 2 ^ l + .... so now (k-2^p) remains

}

}

//change for this Aladdin 1348 problem

if(u == v) // they are in save level and same node

{

return u; //it's because only for this specifc problem

}

//both are in the same level

for(int i=maxPower;i>=0;i--)

{

if(sparseTable[u][i] != -1 && sparseTable[u][i] != sparseTable[v][i])

{

// not in valid and not same

u = sparseTable[u][i];

v = sparseTable[v][i];

}

//valid update will stop if they are just from one node distance from LCA

}

return parent[u]; //the node which is same for two

}

/\*The HLD part\*/

/\*setting up the HLD variables\*/

void setupHLD()

{

memset(visitHLD,0,sizeof(visitHLD)); //neither of the node is visited

for(int i=0;i<MAX\_HLD;i++)

{

graph[i].clear(); //clearing the graph

}

memset(subTreeSize,0,sizeof(subTreeSize)); //clearing all the sub tree size

chainNo = 0; //clearing the chain number

ptr = 0; //the pointer in base array

for(int i=0;i<MAX\_HLD;i++)

{

chainHead[i] = -1; //clearing the chain heads

chainPos[i] = -1; //clearing the positions in each chain

chainInd[i] = -1; //clearing the chain index in which chain it belongs

chainSize[i] = -1; //clearing the chain size

posInBase[i] = -1; //null position now for each node

baseArray[i] = -1; //no cost assigned now

}

//for the LCA part

//clear the sparse table information

memset(sparseTable,-1,sizeof(sparseTable));

memset(parent,-1,sizeof(parent)); //clearing the parent information

memset(level,-1,sizeof(level));

//for the segment tree calculation part

memset(tree,0,sizeof(tree));

}

/\*Function for calculating the sub tree size of each node \*/

void dfsHLD(int node)

{

visitHLD[node] = 1;

for(int i = 0; i<graph[node].size();i++)

{

int v = graph[node][i];

if(visitHLD[v] == 0)

{

level[v] = level[node]+1; //increase the level of this node one more than parent node

parent[v] = node; /\*setting the parent\*/

dfsHLD(v); /\*the node whcich I can visit \*/

subTreeSize[node] += subTreeSize[v]; /\*adding their sub tree size \*/

}

}

subTreeSize[node] += 1; /\*adding it's own node\*/

return;

}

//the main function for performing the HLD

void HLD(int node) //the node for which we will perform our HLD

{

if(chainHead[chainNo] == -1)

{

//this chain has no starting

chainHead[chainNo] = node;

}

chainInd[node] = chainNo; //this node belongs to this chainNo

chainSize[chainNo]++; //increamenting this chain so increasing chainsize of this chain

chainPos[node] = chainSize[chainNo];

ptr++; //increase the pointer in the baseArray

posInBase[node] = ptr; // the index in posInBase array of this node

baseArray[ptr] = cost[node]; //the base array in this position saves the cost of this node

//now want to find the special child which has at least half of the subTree nodes

int idx = -1,highSize = -1;

//there should be so that same node is not visited twice

for(int i=0;i<graph[node].size();i++)

{

int v = subTreeSize[graph[node][i]];

if(v<subTreeSize[node]) //definitely not parent node

{

if(v>highSize) //temporarirly this has max sub tree candidate for special child

{

highSize = v; //save size

idx = i; //save index

}

}

}

if(idx>=0) //this node is not leaf node

{

// so increase the chain

HLD(graph[node][idx]);

}

//after completion of special chain made with special nodes

for(int i = 0; i<graph[node].size() ;i++)

{

if(i != idx) // not the special child

{

int v = subTreeSize[graph[node][i]];

if(v<=highSize) // definitely not it's parent

{

//new chain begins with other normal nodes

chainNo++;

HLD(graph[node][i]);

}

}

}

}

//the part by part calculation

int queryUp(int u,int v,int n)

{

int uchain; //the chain where is u

int vchain=chainInd[v]; //the chain where v belongs

int answer = 0;

while(true)

{

uchain = chainInd[u];

//cout<<"u " << u << " v " << v << " chains " << uchain << " " << vchain << endl;

if(uchain == vchain) //the breaking condition

{

int get = segmentTreeQuery(1,1,n,posInBase[v],posInBase[u]);

answer += get;

//cout<<"get " << get<<endl;

break;

}

int get = segmentTreeQuery(1,1,n,posInBase[chainHead[uchain]],posInBase[u]);

//cout<<"get " << get<<endl;

answer += get;

u = chainHead[uchain];

u = parent[u];

}

return answer;

}

/\*the Query part of HLD\*/

int query(int u,int v,int n) //calculation between u and v

{

int lca = lcaQuery(u,v); //taking the lca

int answer = 0;

if(lca == u)

{

//cout<<"dh " << endl;

answer = queryUp(v,u,n);

}

else if(lca == v)

{

//cout<<"dh " << endl;

answer = queryUp(u,v,n);

}

else if( u == 1)

{

answer = queryUp(v,u,n);

}

else if(v == 1)

{

answer = queryUp(u,v,n);

}

else

{

answer = queryUp(u,lca,n)+queryUp(v,lca,n)-queryUp(lca,lca,n); //over calculation

}

return answer;

}

void queryUpdate(int u, int value,int n)

{

int idx = posInBase[u];

//cout<<" idx " << u <<" " <<value << endl;

segmentTreeUpdate(1,1,n,idx,value);

return;

}

//Square Root Decomposition Mo’s Algorithm

//square root decomposition MO's Algorithm

int counter[MAX\_MO+5]; //counter will be kept of the values by this array

/\*My square root decomposition is totally zero based\*/

//structure for saving the queries

struct squareRoot

{

int l,r;//ranges of query

int n; //total value of the array on which we will perform our square root

int idx; //real order according to input

}query[MAX\_MO+5];

//implementing MO's Algorihtm

bool squareRootQuerySort(squareRoot a,squareRoot b)

{

int blockA = a.l/sqrt(a.n) ; //block number

int blockB = b.l/sqrt(b.n) ;//block number of b

if(blockA < blockB) return true; // we will query from the left blocks first, like first one then two ....

//per set of query Q \* sqrt(n)

//total M \* sqrt(n)

if(blockA == blockB) //now if starting is in the same block

{

//so choose the blocks where r is in ascending order

//because for each set of block we just want to get o(N) at most one time

//tot complexity here highest sqrt(N) blocks \* O(N)

blockA = a.r/sqrt(a.n);

blockB = b.r/sqrt(b.n);

if(blockA<blockB)return true;

else return false;

}

else return false;

}

//now the calculation part

ll answer;

int currentL;

int currentR;

void remove(int index)

{

ll v = counter[arr[index]];

answer = answer - (v \* v) \* arr[index];

counter[arr[index]]--;

v = counter[arr[index]];

answer = answer + (v \* v) \* arr[index];

}

void add(int index)

{

ll v = counter[arr[index]];

answer = answer - (v \* v) \* arr[index];

counter[arr[index]]++;

v = counter[arr[index]];

answer = answer + (v \* v) \* arr[index];

}

void calculation(int l,int r)

{

while(currentL<l)

{

// so I am having too much value from the left side

//need to decrease

//cout<<" answer before " << answer << endl;

remove(currentL);

currentL++;

}

while(currentL>l)

{

currentL--;

add(currentL);

}

while(currentR>r)

{

remove(currentR);

currentR--;

}

while(currentR<r)

{

currentR++;

add(currentR);

}

return;

}

//Rectangle Union + Orthogonal Range search

//Unordered Map

#include <bits/stdc++.h>

using namespace std;

struct HASH{

size\_t operator()(const pair<int,int>&x)const{

return hash<long long>()(((long long)x.first)^(((long long)x.second)<<32));

}

};

unordered\_map<pair<int,int>,int,HASH>mp;

unordered\_map<int,int>UM;

int main(void) {

UM.reserve(1024) // power of 2 to save value in list inside works for storing with hash value

UM.max\_load\_factor(0.25);

}

//Segment Tree

/Spoj GSS3

#include <bits/stdc++.h>

using namespace std;

#define MAX 50005

int arr[MAX+2];

struct segmentTreeInfo{

int bestPrefix;

int bestSuffix;

int totalSum;

int bestSum;

};

struct segmentTreeInfo segmentTree[4\*MAX+1];

void buildSegmentTree(int node, int st, int en) {

if(st == en) {

segmentTree[node].bestPrefix=arr[st];

segmentTree[node].bestSuffix=arr[st];

segmentTree[node].totalSum=arr[st];

segmentTree[node].bestSum=arr[st];

return;

}

int left=2\*node;

int right=2\*node+1;

int mid=(st+en)/2;

buildSegmentTree(left,st,mid);

buildSegmentTree(right,mid+1,en);

segmentTree[node].bestPrefix=max(segmentTree[left].bestPrefix,segmentTree[left].totalSum+segmentTree[right].bestPrefix);

segmentTree[node].bestSuffix=max(segmentTree[right].bestSuffix,segmentTree[right].totalSum+segmentTree[left].bestSuffix);

segmentTree[node].totalSum=segmentTree[left].totalSum+segmentTree[right].totalSum;

segmentTree[node].bestSum=max(max(segmentTree[left].bestSum,segmentTree[right].bestSum),segmentTree[left].bestSuffix+segmentTree[right].bestPrefix);

return;

}

void updateSegmentTree(int node, int st, int en, int x, int value){

if(st>x||en<x) return;

if(st==en && st==x) {

arr[st]=value;

segmentTree[node].bestPrefix=arr[st];

segmentTree[node].bestSuffix=arr[st];

segmentTree[node].totalSum=arr[st];

segmentTree[node].bestSum=arr[st];

return;

}

int left=2\*node;

int right=2\*node+1;

int mid=(st+en)/2;

updateSegmentTree(left,st,mid,x,value);

updateSegmentTree(right,mid+1,en,x,value);

segmentTree[node].bestPrefix=max(segmentTree[left].bestPrefix,segmentTree[left].totalSum+segmentTree[right].bestPrefix);

segmentTree[node].bestSuffix=max(segmentTree[right].bestSuffix,segmentTree[right].totalSum+segmentTree[left].bestSuffix);

segmentTree[node].totalSum=segmentTree[left].totalSum+segmentTree[right].totalSum;

segmentTree[node].bestSum=max(max(segmentTree[left].bestSum,segmentTree[right].bestSum),segmentTree[left].bestSuffix+segmentTree[right].bestPrefix);

return;

}

segmentTreeInfo segmentTreeQuery(int node, int st, int en, int x, int y){

if(x>en||y<st) {

segmentTreeInfo ret;

ret.bestPrefix= -10000000;

ret.bestSuffix= -10000000;

ret.totalSum=0;

ret.bestSum = -100000000;

return ret;

}

if(st>=x && en <= y) {

return segmentTree[node];

}

int left=2\*node;

int right=2\*node+1;

int mid=(st+en)/2;

segmentTreeInfo ret1=segmentTreeQuery(left, st,mid, x, y);

segmentTreeInfo ret2=segmentTreeQuery(right,mid+1, en, x,y);

segmentTreeInfo ret;

ret.bestPrefix=max(ret1.bestPrefix,ret1.totalSum+ret2.bestPrefix);

ret.bestSuffix=max(ret2.bestSuffix,ret2.totalSum+ret1.bestSuffix);

ret.totalSum=ret1.totalSum+ret2.totalSum;

ret.bestSum=max(max(ret1.bestSum,ret2.bestSum),ret1.bestSuffix+ret2.bestPrefix);

return ret;

}

int main() {

int n;

scanf("%d",&n);

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

scanf("%d",&arr[i]);

}

buildSegmentTree(1,1,n);

int q;

scanf("%d",&q);

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

int ch;

scanf("%d",&ch);

if(ch == 0) {

int x,y;

scanf("%d %d",&x,&y);

updateSegmentTree(1,1,n,x,y);

}

else {

int x,y;

scanf("%d %d",&x,&y);

segmentTreeInfo ans = segmentTreeQuery(1,1,n,x,y);

int answer= max(max(ans.totalSum,ans.bestPrefix),ans.bestSuffix);

answer=max(answer,ans.bestSum);

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

}

}

return 0;

}

//Segment Tree , Binary search, Unordered Map

#include <bits/stdc++.h>

using namespace std;

#define MAX 200005

struct info{

string op;

int value;

}arr[MAX+1];

struct segmentTreeInfo{

int sum;

int present;

};

int given[MAX+5];

unordered\_map<int,int>M;

unordered\_map<int,int>Mnew;

int CCount[MAX+1];

bool cmpx(int a, int b) {

if(a<b) return true;

else return false;

}

segmentTreeInfo segmentTree[3\*MAX+5];

void updateSegmentTree(int node, int st, int en, int x, int value) {

if(st>x||en<x) return;

if(st==en && x == st) {

segmentTree[node].present += value;

segmentTree[node].sum += value;

return;

}

int left=2\*node;

int right=2\*node+1;

int mid=(st+en)/2;

updateSegmentTree(left,st,mid,x,value);

updateSegmentTree(right,mid+1,en,x,value);

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

return;

}

int queryOneCount(int node, int st, int en, int x, int y) {

if(x>y) return 0;

if(st>y||en<x) return 0;

//cout<<"st " << st <<" "<< en << endl;

if(st>=x&&en<=y) return segmentTree[node].sum;

int left=2\*node;

int right=2\*node+1;

int mid=(st+en)/2;

int ret1=queryOneCount(left,st,mid,x,y);

int ret2=queryOneCount(right,mid+1,en,x,y);

return ret1+ret2;

}

int binarySearch(int low,int high, int k){

int mid;

bool ok=false;

int save=high+1;

int cnt=high;

//cout<<"k th " << endl;

while(low<=high){

mid=(low+high)/2;

int value=queryOneCount(1,1,cnt,1,mid);

//cout<<mid<<" value " << value << endl;

//getchar();

if(value==k) {

ok=true;

high=mid-1;

save=min(save,mid);

}

else if(value>k){

ok=true;

high=mid-1;

}

else if(value<k){

low=mid+1;

}

}

if(ok) {

return save;

}

else return -1;

}

int main(void){

M.reserve(1024);

M.max\_load\_factor(0.24);

Mnew.reserve(1024);

Mnew.max\_load\_factor(0.24);

int q;

scanf("%d",&q);

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

char op[3];

int number;

scanf("%s %d",op,&number);

arr[i].op=op;

arr[i].value=number;

given[i]=number;

}

sort(given+1,given+1+q,cmpx);

int cnt=0;

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

if(M[given[i]]==0) {

cnt++;

M[given[i]]=cnt;

//cout<<given[i]<<endl;

CCount[cnt]=given[i];

}

}

/\*for(int i=1;i<=cnt;i++) {

cout<<CCount[i]<<endl;

}\*/

memset(segmentTree,0,sizeof(segmentTree));

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

if(arr[i].op[0]=='I') {

int value=arr[i].value;

Mnew[value]++;

if(Mnew[value]>1) {

Mnew[value]=1;

continue;

}

value=M[value];

updateSegmentTree(1,1,cnt,value,1);

/\*int s;

cin>>s;

for(int j=1;j<=s;j++) {

int a,b;

cin>>a>>b;

cout<<"out " << queryOneCount(1,1,cnt,a,b)<<" " << j << endl;

}

cout<<"done"<<endl;\*/

}

else if(arr[i].op[0] == 'D') {

int value=arr[i].value;

Mnew[value]--;

if(Mnew[value]<0){

Mnew[value]=0;

continue;

}

value=M[value];

updateSegmentTree(1,1,cnt,value,-1);

}

else if(arr[i].op[0] == 'K') {

int answer= binarySearch(1,cnt,arr[i].value);

if(answer == -1) printf("invalid\n");

else printf("%d\n",CCount[answer]);

}

else if(arr[i].op[0] == 'C') {

int value=arr[i].value;

value=M[value];

int answer=queryOneCount(1,1,cnt,1,value-1);

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

}

}

return 0;

}

//Rectangle Union

include<bits/stdc++.h>

**using** **namespace** std;

#define MAX 30010

typedef long long int ll;

*//Rectangle union*

struct point

**{**

int x,ymin,ymax;

int type;

}Point[5\*MAX+5]; *// to keep information about the points*

**vector<int>tempY; *// just to keep the input y co ordinate information***

int finalY[5\*MAX+5]; *//the final Y co ordinates*

map<int,int>M; *// the changed Y information*

struct segmentTreeInfo

**{**

ll sum; *//segment sum*

ll partSum; *//partial sum*

int midProp;

int propagation;

**}tree[600000+5];**

*//sort normal on the basis of y*

bool cmpY(int a, int b)

**{**

if(a<b) return **true**;

else return **false**;

}

*//sort on the basis of x*

**bool cmpX(point a, point b)**

{

if(a.x<b.x) return **true**;

if(a.x == b.x)

{

**if(a.ymin<b.ymin)return true;**

else if(a.ymin == b.ymin)

{

if(a.ymax<b.ymax)return **true**;

else if(a.ymax == b.ymax)

**{**

if(a.type<b.type) return **true**;

else return **false**;

}

else return **false**;

**}**

else return **false**;

}

return **false**;

}

void updateSegmentTree(int idx, int st, int en,int x, int y, int prop)

{

*//cout<<"st = " << st <<" "<<en <<" "<<x <<" " << y << endl;*

if(y<st||x>en){

**int v = tree[idx].propagation;**

if(v>0)

{

if(st == en)

{

**return;**

}

else

{

tree[idx].sum = finalY[en]-finalY[st];

**}**

}

return;

}

if(st>=x && en <= y)

**{**

tree[idx].propagation += prop;

if((tree[idx].propagation)>0)

{

if(st == en)

**{**

tree[idx].sum = 0;

return;

}

else

**{**

tree[idx].sum = finalY[en]-finalY[st];

}

}

else

**{**

return ;

}

return;

}

**int left = 2 \* idx;**

int right = 2 \* idx + 1;

int mid = (st+en)/2;

updateSegmentTree(left,st,mid,x,y,prop);

updateSegmentTree(right,mid+1,en,x,y,prop);

***//cout<<"joining node " << idx <<" " << tree[left].propagation <<" "<< tree[right].propagation << endl;***

tree[idx].partSum = 0;

if(tree[left].propagation >0)

{

tree[idx].partSum = tree[left].sum;

**}**

else tree[idx].partSum = tree[left].partSum;

if(tree[right].propagation >0)

{

tree[idx].partSum += tree[right].sum;

**}**

else tree[idx].partSum += tree[right].partSum;

if((mid+1)<=y && mid >=x)

{

*//cout<<"mid here " << mid <<" "<< mid+1<< endl;*

**tree[idx].midProp += prop;**

if(tree[idx].midProp>0)tree[idx].partSum += finalY[mid+1]-finalY[mid];

}

else if(tree[idx].midProp>0)

{

**tree[idx].partSum += finalY[mid+1]-finalY[mid];**

}

return;

}

ll rectangleUnion(int n) *// n \* 2 is the total number of points*

{

sort(tempY.begin(),tempY.end(),cmpY); *// normal Y sorting*

*//now leave the redundant y's just take the main y's and map them*

**int yCounter = 0;**

for(int i = 0; i<tempY.size();i++)

{

if(M[tempY[i]] == 0)

{

**yCounter++;**

M[tempY[i]] = yCounter;

finalY[yCounter] = tempY[i];

}

}

***//now sort on the basis of x***

sort(Point+1,Point+1+n,cmpX); *//total point is n*

*/\*cout<<"one the basis of x sorting " << endl;*

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

*{*

***printf("%d %d %d\n",Point[i].x,Point[i].ymin,Point[i].ymax);***

*}*

*cout<<"the y's " << endl;*

*for(int i = 1; i<= yCounter; i++)*

*{*

***printf("%d ",finalY[i]);***

*}*

*cout<<endl;\*/*

memset(tree,0,sizeof(tree)); *//segment tree clear*

*//now start line sweeping from the left side*

**updateSegmentTree(1,1,yCounter,M[Point[1].ymin],M[Point[1].ymax],1);**

*/\*cout<<"printing tree " << endl;*

*for(int j = 1; j<= 10; j++)*

*{*

*cout<<tree[j].sum <<" "<< tree[j].propagation<<" "<<tree[j].partSum << endl;*

***}***

*\* \*/*

ll answer = 0;

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

{

***//cout<<"input now " << Point[i].x <<" " << Point[i].ymin<<" "<<Point[i].ymax <<" "<< Point[i].type<< endl;***

ll dx = Point[i].x - Point[i-1].x;

ll dy;

if(tree[1].propagation == 1) dy = tree[1].sum;

else dy = tree[1].partSum;

**answer = answer + dx \* dy;**

*//cout<<"dx = " << dx <<" "<<dy << endl;*

if(Point[i].type == 0) *// opening*

{

updateSegmentTree(1,1 , yCounter , M[Point[i].ymin] , M[Point[i].ymax],1);

***/\*cout<<"printing tree " << endl;***

*for(int j = 1; j<= 10; j++)*

*{*

*cout<<tree[j].sum <<" "<< tree[j].propagation<<" "<<tree[j].partSum << endl;*

*}\*/*

**}**

else *//closing*

{

updateSegmentTree(1,1 , yCounter , M[Point[i].ymin] , M[Point[i].ymax],-1);

*/\*cout<<"printing tree " << endl;*

***for(int j = 1; j<= 10; j++)***

*{*

*cout<<tree[j].sum <<" "<< tree[j].propagation<<" "<<tree[j].partSum << endl;*

*}\*/*

}

**}**

return answer;

}

**int main(void)**

{

*//freopen("in.txt","r",stdin);*

*//freopen("out.txt","w",stdout);*

int T,t;

**scanf("%d",&T);**

for(t = 1; t <= T; t++)

{

int n;

scanf("%d",&n);

**tempY.clear();**

M.clear();

int xCounter = 0;

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

{

**int a,b,c,d;**

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

xCounter++;

Point[xCounter].type = 0; *// 0 means starting bahu*

Point[xCounter].x = a;

**Point[xCounter].ymin = b;**

Point[xCounter].ymax = d;

xCounter++;

Point[xCounter].type = 1; *//1 means closing bahu*

**Point[xCounter].x = c;**

Point[xCounter].ymin = b;

Point[xCounter].ymax = d;

*//keeping into normal Y array*

**tempY.push\_back(b);**

tempY.push\_back(d);

}

ll answer = rectangleUnion(xCounter);

printf("Case %d: %lld**\n**",t,answer);

**assert(answer>=0);**

}

return 0;

}

//LCA

#include<bits/stdc++.h>

1. **using** **namespace** std;
2. #define MX 100010
4. **typedef long long int ll;**
5. vector<long long int >graph[MX+2];
6. long long int cost[MX+2];
7. long long int T[MX+2];
8. long long int color[MX+2];
9. **long long int P[MX+2][30];**
10. long long int level[MX+1];
11. long long int maxLevelNode;
13. void dfs(long long int node)
14. **{**
15. color[node]=1;
16. for(long long int i=0; i<graph[node].size() ; i++)
17. {
18. if(color[graph[node][i]]==0)
19. **{**
20. T[graph[node][i]]= node;
21. level[graph[node][i]]=level[node]+1;
22. maxLevelNode=max(maxLevelNode,level[graph[node][i]]);
23. dfs(graph[node][i]);
24. **}**
25. }
27. }

30. void lcaPreprocess(long long int n)
31. {
32. *//here I am starting with zeroth node*
33. for(long long int i=0; i<n; i++)
34. **{**
35. P[i][0]=T[i];
36. }

39. **for(long long int j=1; (1<<j)< n; j++)**
40. {
41. for(long long int i=0; i<n; i++)
42. {
43. if(P[i][j-1] != -1)
44. **{**
45. P[i][j]= P[P[i][j-1]][j-1];
46. }
47. }
48. }
50. }
52. long long int lca(long long int node, long long int value)
53. {
55. long long int log;
56. for(log=0; (1<<log)<=level[node]; log++);
57. *//log--;*
58. long long int save=node;

61. for(long long int i=log; i>=0; i--)
62. {
63. long long int v=P[node][i];
64. **if(v==-1) continue;**
65. *//cout<<"v " << v << " " << i << " " << cost[v] << " " << value<< endl;*
66. if(cost[v] >= value)
67. {
68. node=v;
69. ***//cout<<"ase"<<endl;***
70. save=min(save,v);
71. }
72. }
73. return save;
74. **}**
76. int main()
77. {
78. long long int Test,t;
79. ***//freopen("input.txt","r",stdin);***
80. *//freopen("output.txt","w",stdout);*
81. scanf("%lld",&Test);
82. for(t=1; t<=Test; t++)
83. {
84. **long long int n,k;**
85. scanf("%lld %lld",&n,&k);
87. for(long long int i=0; i<MX; i++)
88. {
89. **graph[i].clear();**
90. cost[i]=0;
91. T[i]= -1;
92. level[i]=0;
93. }
95. cost[0]=1;
97. for(long long int i=1; i<=n-1; i++)
98. {
99. **long long int a;**
100. long long int b;
101. scanf("%lld %lld",&a,&b);
102. graph[a].push\_back(i);
103. cost[i]=b;
104. **}**
106. memset(P,-1,sizeof(P));
107. memset(color,0,sizeof(color));
108. maxLevelNode=-1;
109. **for(long long int i=0; i<n; i++)**
110. {
111. if(color[i]==0)
112. {
113. dfs(i);
114. **}**
115. }
116. */\*for(long long int i=0;i<n;i++){*
117. *prlong long int f("parent of %d = %d\n",i,T[i]);*
118. *}\*/*
119. **lcaPreprocess(n);**
120. printf("Case %lld:**\n**",t);
121. for(long long int i=1; i<=k; i++)
122. {
123. long long int a;
124. **long long int b;**
125. scanf("%lld %lld",&a,&b);
126. long long int v=lca(a,b);
127. printf("%lld**\n**",v);
128. }
130. }
131. return 0;
132. }

//Sqrt Root Decomposition

1. #include<bits/stdc++.h>
2. **using** **namespace** std;
3. #define MX 100010
4. #define MX\_ONE 50050
6. int arr[MX+5];
8. *//using Sqrt root decomposition with MO's algorithm*
10. **struct info**
11. {
12. int l,r,id; *//low and high range*
13. } Q[MX\_ONE+5];
15. **int offlineResult[MX+5];**
16. int frequency[MX+5];
18. int n; *//number of elements*
19. int answer; *// the main result*
20. **int sizeGuni[MX+5];**

23. *//sorting on the basis of low and high range*
24. bool MOAlgorithm(const info &a, const info &b)
25. **{**
26. int v=a.l/sqrt(n);
27. int u=b.l/sqrt(n);
28. if(v<u) return **true**;
29. if(v==u)
30. **{**
31. if(a.r<b.r) return **true**;
32. else return **false**;
33. }
34. return **false**;
35. **}**
37. *//addition*
39. void add(int idx)
40. **{**
41. frequency[arr[idx]]++;
42. int v= frequency[arr[idx]];
43. sizeGuni[v]++;
44. if(answer<frequency[arr[idx]]) {
45. **answer=frequency[arr[idx]];**
46. }
47. }
49. *//remove*
50. **void \_remove(int idx)**
51. {
52. int v=frequency[arr[idx]];
53. frequency[arr[idx]]--;
54. sizeGuni[v]--;
55. **if(sizeGuni[v] ==0 && answer == v) {**
56. answer--;
57. }
59. }
61. void sqrtRootDecomposition(int query)
62. {
63. *// taking the queries*
64. for(int i=1; i<=query; i++)
65. **{**
66. scanf("%d %d",&Q[i].l,&Q[i].r);
67. Q[i].id=i;
68. }
69. *//sorting the queries with MO's algorithm offline query*
70. **sort(Q+1,Q+1+query,MOAlgorithm);**
71. *//making the frequency zero*
72. memset(frequency,0,sizeof(frequency));
73. memset(sizeGuni,0,sizeof(sizeGuni));
74. int currentL=0,currentR=0; *//left and right pointer*
75. **answer=0; *// main result***
76. for(int i=1; i<=query; i++)
77. {
78. int L=Q[i].l;
79. int R=Q[i].r;
80. **if(currentL<L)**
81. {
82. while(currentL<L)
83. {
84. \_remove(currentL);
85. **currentL++;**
86. }
88. }
89. if(currentR>R)
90. **{**
92. while(currentR>R)
93. {
94. \_remove(currentR);
95. **currentR--;**
96. }
97. }
99. if(currentL>L)
100. **{**
101. while(currentL>L)
102. {
103. currentL--;
104. add(currentL);
105. **}**
106. }
107. if(currentR<R)
108. {
109. while(currentR<R)
110. **{**
111. currentR++;
112. add(currentR);
114. }
115. **}**
116. offlineResult[Q[i].id]=answer;
118. }
119. return;
120. **}**
122. int main()
123. {
124. int T,t;
125. **scanf("%d",&T);**
126. for(t=1; t<=T; t++)
127. {
128. int query,c;
129. scanf("%d %d %d",&n,&c,&query);
130. **for(int i=1; i<=n; i++)**
131. {
132. scanf("%d",&arr[i]);
133. }
134. sqrtRootDecomposition(query);
135. **printf("Case %d:\n",t);**
136. for(int i=1; i<=query; i++)
137. {
138. printf("%d**\n**",offlineResult[i]);
139. }
140. **}**
141. return 0;
142. }

Persistence Segment Tree (Got WA on Codechef Sorting problem)

#include <bits/stdc++.h>

using namespace std;

#define MAX 5000105

int arr[MAX+1];

int pos[MAX+1]; //position is used to keep the real position which should be

struct persistent\_segment\_tree{

int sum;

persistent\_segment\_tree \*left,\*right;

persistent\_segment\_tree(){}

persistent\_segment\_tree(int s,persistent\_segment\_tree \*l,persistent\_segment\_tree \*r){

sum=s;

left=l;

right=r;

}

};

persistent\_segment\_tree \*nodes[MAX+1];

void build(persistent\_segment\_tree \*current, int st, int en){

if(st==en){

current->sum=0;

return;

}

int mid=(st+en)/2;

current->left=new persistent\_segment\_tree(0,NULL,NULL);

current->right=new persistent\_segment\_tree(0,NULL,NULL);

build(current->left,st,mid);

build(current->right,mid+1,en);

current->sum=current->left->sum+current->right->sum;

return;

}

/\*This function is to update 1 in the postion of i like if 1 is in 5th position then there will be 1 \*/

void update(persistent\_segment\_tree \*current, persistent\_segment\_tree \*notun,int st,int en,int position){

if(st==en && position==st){

notun->sum=1;

return;

}

int mid=(st+en)/2;

if(position<=mid){

notun->right=current->right;

notun->left=new persistent\_segment\_tree(0,NULL,NULL);

update(current->left,notun->left,st,mid,position);

}

else{

notun->left=current->left;

notun->right=new persistent\_segment\_tree(0,NULL,NULL);

update(current->right,notun->right,mid+1,en,position);

}

notun->sum=notun->left->sum+notun->right->sum;

return;

}

//function to find position number value

int query(persistent\_segment\_tree \*b, persistent\_segment\_tree \*a, int st, int en,int position){

if(st==en) {

return st;

}

int mid=(st+en)/2;

int ret=0;

if((a->left->sum - b->left->sum) >= position){

ret=query(b->left,a->left,st,mid,position);

}

else{

ret=query(b->right,a->right,mid+1,en,position-(a->left->sum - b->left->sum));

}

return ret;

}

struct Queries{

int l,h;

};

long long int solve(int low,int high){

queue<Queries>Q;

Queries temp;

temp.l=low;

temp.h=high;

Q.push(temp);

int n=high;

int answer=0;

while(Q.empty() != true){

temp=Q.front();

Q.pop();

low=temp.l;

high=temp.h;

if((high-low+1)>=2){

//cout<<(high-low+2)/2 << endl;

int position = query(nodes[low-1],nodes[high],1,n,(high-low+2)/2);

int value=arr[position];

temp.l=low;

temp.h=value-1;

Q.push(temp);

temp.l=value+1;

temp.h=high;

Q.push(temp);

answer += (high-low+1);

//cout<<"value " << value << " "<<position << endl;

}

}

return answer;

}

int main(int argc, char \*\*argv) {

int n;

scanf("%d",&n);

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

scanf("%d",&arr[i]);

pos[arr[i]]=i; //saving the position of where i is like 1 is where 2 is where 3 is where etc

}

nodes[0]=new persistent\_segment\_tree(0,NULL,NULL); //the basic segment tree

build(nodes[0],1,n); //basic segment tree been built

//make n persistent segment trees

//each segment tree sets bit in certain position

//like for nodes[i] segment tree has i bits one and each corresponds to a certain position

//nodes[i] segment tree contains 1 for 1 upto i values in their positions

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

nodes[i] = new persistent\_segment\_tree(0,NULL,NULL);

update(nodes[i-1],nodes[i],1,n,pos[i]);

}

long long int answer=solve(1,n);

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

return 0;

}

Writing compare function

<http://codeforces.com/blog/entry/9604>

//compare function for map

struct tic\_tac\_toe{

int state[4][4];

int id;

int f\_n,g\_n;

};

struct compare{

bool operator()(const tic\_tac\_toe &a, const tic\_tac\_toe &b) {

if(a.state[1][1]<b.state[1][1]) return true;

else if(a.state[1][1]>b.state[1][1]) return false;

else {

if(a.state[1][2]<b.state[1][2]) return true;

else if(a.state[1][2] > b.state[1][2]) return false;

else{

if(a.state[1][3]<b.state[1][3]) return true;

else if(a.state[1][3] > b.state[1][3]) return false;

else{

if(a.state[2][1]<b.state[2][1]) return true;

else if(a.state[2][1]>b.state[2][1]) return false;

else{

if(a.state[2][2]<b.state[2][2]) return true;

else if(a.state[2][2]>b.state[2][2]) return false;

else{

if(a.state[2][3]<b.state[2][3]) return true;

else if(a.state[2][3]>b.state[2][3]) return false;

else{

if(a.state[3][1]<b.state[3][1]) return true;

else if(a.state[3][1]>b.state[3][1]) return false;

else{

if(a.state[3][2]<b.state[3][2]) return true;

else if(a.state[3][2]>b.state[3][2]) return true;

else{

if(a.state[3][3]<b.state[3][3]) return true;

else if(a.state[3][3]>b.state[3][3]) return true;

else {

return false;

}

}

}

}

}

}

}

}

}

}

};

map<tic\_tac\_toe,int,compare>M; // declaration

// Compare Function for priority queue

struct compare\_two{

bool operator ()(const tic\_tac\_toe &a,const tic\_tac\_toe &b) {

if((a.f\_n+a.g\_n)<(b.f\_n+b.g\_n)) return true;

else return false;

}

};

priority\_queue<tic\_tac\_toe,vector<tic\_tac\_toe>,compare\_two>PQ;

// Persistent Segment tree (Tested at Spoj Mkthnum)

#include <bits/stdc++.h>

using namespace std;

#define MAX 100005

int arr[MAX+1];

int temp\_arr[MAX+1];

int pos[MAX+1];

map<int,int>M,M2;

struct segment\_tree\_info\_1{

int low,high;

};

segment\_tree\_info\_1 tree1[4\*MAX+1];

void build\_segment\_tree(int node,int st,int en){

if(st==en){

tree1[node].high=M[arr[st]];

tree1[node].low=M[arr[st]];

return;

}

int mid = (st+en)/2;

int left=2\*mid;

int right=2\*mid+1;

build\_segment\_tree(left,st,mid);

build\_segment\_tree(right,mid+1,en);

tree1[node].high = max(tree1[left].high,tree1[right].high);

tree1[node].low = min(tree1[left].low,tree1[right].low);

return;

}

segment\_tree\_info\_1 query(int node,int st,int en,int x,int y){

if(st>y || en<x) {

segment\_tree\_info\_1 temp;

temp.low = 10000000;

temp.high = 0;

return temp;

}

if(st>=x && en <= y) {

return tree1[node];

}

int mid = (st+en)/2;

int left=2\*mid;

int right=2\*mid+1;

segment\_tree\_info\_1 ret1,ret2,ret;

ret1 = query(left,st,mid,x,y);

ret2 = query(right,mid+1,en,x,y);

ret.high = max(ret1.high,ret2.high);

ret.low = min(ret1.low,ret2.low);

return ret;

}

struct persistent\_segment\_tree{

int sum;

persistent\_segment\_tree \*left;

persistent\_segment\_tree \*right;

persistent\_segment\_tree(){}

persistent\_segment\_tree(int s, persistent\_segment\_tree \*l, persistent\_segment\_tree \*r){

sum=s;

left=l;

right=r;

}

};

persistent\_segment\_tree \*nodes[MAX+5];

void build\_persistent\_segment\_tree(persistent\_segment\_tree \*current,int st,int en){

if(st==en){

current->sum=0;

return;

}

int mid = (st+en)/2;

current->left = new persistent\_segment\_tree(0,NULL,NULL);

current->right = new persistent\_segment\_tree(0,NULL,NULL);

build\_persistent\_segment\_tree(current->left,st,mid);

build\_persistent\_segment\_tree(current->right,mid+1,en);

current->sum = current->left->sum + current->right->sum;

return;

}

void update\_persistent\_segment\_tree(persistent\_segment\_tree \*current,persistent\_segment\_tree \*notun,int st,int en,int pos){

if(st==en){

notun->sum = 1;

return;

}

int mid = (st+en)/2;

if(pos<=mid){

notun->right= current->right;

notun->left = new persistent\_segment\_tree(0,NULL,NULL);

update\_persistent\_segment\_tree(current->left,notun->left,st,mid,pos);

}

else{

notun->left= current->left;

notun->right = new persistent\_segment\_tree(0,NULL,NULL);

update\_persistent\_segment\_tree(current->right,notun->right,mid+1,en,pos);

}

notun->sum = notun->left->sum + notun->right->sum;

return;

}

int query\_persistent\_segment\_tree(persistent\_segment\_tree \*previous,persistent\_segment\_tree \*present,int st,int en, int num){

if(st==en) return st;

int mid=(st+en)/2;

int answer;

if((present->left->sum - previous->left->sum)>=num){

answer = query\_persistent\_segment\_tree(previous->left,present->left,st,mid,num);

}

else{

answer = query\_persistent\_segment\_tree(previous->right,present->right,mid+1,en,num-(present->left->sum - previous->left->sum));

}

return answer;

}

void mkthnum(){

int n,m;

scanf("%d %d",&n,&m);

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

scanf("%d",&arr[i]);

}

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

temp\_arr[i]=arr[i];

}

sort(temp\_arr+1,temp\_arr+1+n);

int counter=0;

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

counter++;

M[temp\_arr[i]]=counter;

M2[counter]=temp\_arr[i];

}

nodes[0] = new persistent\_segment\_tree(0,NULL,NULL);

build\_persistent\_segment\_tree(nodes[0],1,n);

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

nodes[i]=new persistent\_segment\_tree(0,NULL,NULL);

update\_persistent\_segment\_tree(nodes[i-1],nodes[i],1,n,M[arr[i]]);

}

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

int a,b,c;

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

int v = query\_persistent\_segment\_tree(nodes[a-1],nodes[b],1,n,c);

printf("%d\n",M2[v]);

}

return;

}

int main(int argc,char \*\*argv){

//freopen("in.txt","r",stdin);

mkthnum();

return 0;

}

// Implicit Persistent Segment tree

#include <bits/stdc++.h>

using namespace std;

#define MAX 100002

#define MAX\_VALUE 1000000000

#define MAX\_STRING 100003

/\*\*\* For taking query\*\*/

struct Query{

string type;

string op;

int priority;

};

vector<Query>input;

/\*\* For assignment map \*\*/

map<string,int>M;

map<int,string>rev\_map;

int counter\_string;

struct persistent\_segment\_tree\_info{

int priority;

persistent\_segment\_tree\_info \*left,\*right;

persistent\_segment\_tree\_info(){}

persistent\_segment\_tree\_info(int v, persistent\_segment\_tree\_info \*l, persistent\_segment\_tree\_info \*r){

priority=v;

left=l;

right=r;

}

};

persistent\_segment\_tree\_info \*tree\_one[MAX+1];

void build\_persisent\_segment\_tree(persistent\_segment\_tree\_info \*node,int st,int en){

if(st==en){

node->priority=0;

return;

}

int mid=(st+en)/2;

node->left=new persistent\_segment\_tree\_info(0,NULL,NULL);

node->right=new persistent\_segment\_tree\_info(0,NULL,NULL);

build\_persisent\_segment\_tree(node->left,st,mid);

build\_persisent\_segment\_tree(node->right,mid+1,en);

node->priority=0;

}

int query\_persistent\_segment\_tree(persistent\_segment\_tree\_info \*node,int st,int en,int x){

if(st==en){

return node->priority;

}

int mid=(st+en)/2;

if(x<=mid){

return query\_persistent\_segment\_tree(node->left,st,mid,x);

}

else{

return query\_persistent\_segment\_tree(node->right,mid+1,en,x);

}

}

void update\_persistent\_segment\_tree(persistent\_segment\_tree\_info \*current, persistent\_segment\_tree\_info \*notun,int st,int en,int x, int value){

if(st==en){

notun->priority=value;

return;

}

int mid=(st+en)/2;

if(x<=mid){

notun->right = current->right;

notun->left=new persistent\_segment\_tree\_info(0,NULL,NULL);

update\_persistent\_segment\_tree(current->left,notun->left,st,mid,x,value);

}

else{

notun->left = current->left;

notun->right=new persistent\_segment\_tree\_info(0,NULL,NULL);

update\_persistent\_segment\_tree(current->right,notun->right,mid+1,en,x,value);

}

return;

}

/\*\* for counter of lower priorities \*\*/

struct implicit\_persistent\_segment\_tree\_info{

int sum;

implicit\_persistent\_segment\_tree\_info \*left,\*right;

implicit\_persistent\_segment\_tree\_info(){}

implicit\_persistent\_segment\_tree\_info(int v,implicit\_persistent\_segment\_tree\_info \*l, implicit\_persistent\_segment\_tree\_info \*r){

left=l;

right=r;

sum=v;

}

};

implicit\_persistent\_segment\_tree\_info \*tree[MAX+1];

void update\_implicit\_persistent\_segment\_tree(implicit\_persistent\_segment\_tree\_info \*current, implicit\_persistent\_segment\_tree\_info \*notun, int st, int en, int x,int value){

if(st==en){

notun->sum = current->sum;

notun->sum +=value;

return;

}

int mid=(st+en)/2;

if(x<=mid){

notun->right = current->right;

if(current->left==NULL){

current->left = new implicit\_persistent\_segment\_tree\_info(0,NULL,NULL);

}

notun->left = new implicit\_persistent\_segment\_tree\_info(0,NULL,NULL);

update\_implicit\_persistent\_segment\_tree(current->left,notun->left,st,mid,x,value);

}

else{

notun->left = current->left;

if(current->right==NULL){

current->right = new implicit\_persistent\_segment\_tree\_info(0,NULL,NULL);

}

notun->right = new implicit\_persistent\_segment\_tree\_info(0,NULL,NULL);

update\_implicit\_persistent\_segment\_tree(current->right,notun->right,mid+1,en,x,value);

}

int left\_sum=0,right\_sum=0;

if(notun->left==NULL) left\_sum = 0;

else left\_sum = notun->left->sum;

if(notun->right==NULL) right\_sum = 0;

else right\_sum = notun->right->sum;

notun->sum = left\_sum + right\_sum;

return;

}

int query\_implicit\_persistent\_segment\_tree(implicit\_persistent\_segment\_tree\_info \*node, int st, int en,int x,int y){

if(node == NULL) return 0;

if(st>y||en<x) return 0;

if(st>=x&&en<=y){

if(node == NULL) return 0;

else return node->sum;

}

int mid=(st+en)/2;

int ret1=0,ret2=0;

ret1=query\_implicit\_persistent\_segment\_tree(node->left,st,mid,x,y);

ret2=query\_implicit\_persistent\_segment\_tree(node->right,mid+1,en,x,y);

return ret1+ret2;

}

void string\_map(){

counter\_string = 0;

/\*for(int i=0;i<input.size();i++){

if(input[i].type=="set" || input[i].type=="query" || input[i].type=="remove"){

if(M[input[i].op] == 0){

counter\_string++;

M[input[i].op]=counter\_string;

//rev\_map[counter\_string]=input[i].op;

}

}

}\*/

return;

}

void soln(int i){

//for(int i=0;i<input.size();i++){

if(input[i].type=="set"){

int v;

if(M[input[i].op]==0){

counter\_string++;

M[input[i].op] = counter\_string;

//cout<<M[input[i].op]<<" and " << input[i].op<<endl;

v = 0;

}

else {

v = query\_persistent\_segment\_tree(tree\_one[i],1,MAX\_STRING,M[input[i].op]);

}

if(v == 0){

//cout<<"create hoi " <<input[i].priority << endl;

tree\_one[i+1] = new persistent\_segment\_tree\_info(0,NULL,NULL);

update\_persistent\_segment\_tree(tree\_one[i],tree\_one[i+1],1,MAX\_STRING,M[input[i].op],input[i].priority);

tree[i+1] = new implicit\_persistent\_segment\_tree\_info(0,NULL,NULL);

update\_implicit\_persistent\_segment\_tree(tree[i],tree[i+1],1,MAX\_VALUE,input[i].priority,1);

}

else{

//cout<<"age chilo " << M[input[i].op]<<" "<< v<< endl;

tree\_one[i+1] = new persistent\_segment\_tree\_info(0,NULL,NULL);

update\_persistent\_segment\_tree(tree\_one[i],tree\_one[i+1],1,MAX\_STRING,M[input[i].op],input[i].priority);

tree[i+1] = new implicit\_persistent\_segment\_tree\_info(0,NULL,NULL);

implicit\_persistent\_segment\_tree\_info \*notun = new implicit\_persistent\_segment\_tree\_info(0,NULL,NULL);

update\_implicit\_persistent\_segment\_tree(tree[i],notun,1,MAX\_VALUE,v,-1);

update\_implicit\_persistent\_segment\_tree(notun,tree[i+1],1,MAX\_VALUE,input[i].priority,1);

}

}

else if(input[i].type=="remove"){

int v;

if(M[input[i].op] == 0) {

counter\_string++;

M[input[i].op] = counter\_string;

v = 0;

}

else {

v = query\_persistent\_segment\_tree(tree\_one[i],1,MAX\_STRING,M[input[i].op]);

}

if(v == 0) {

tree[i+1]=tree[i];

tree\_one[i+1]=tree\_one[i];

}

else {

tree\_one[i+1] = new persistent\_segment\_tree\_info(0,NULL,NULL);

update\_persistent\_segment\_tree(tree\_one[i],tree\_one[i+1],1,MAX\_STRING,M[input[i].op],0);

tree[i+1] = new implicit\_persistent\_segment\_tree\_info(0,NULL,NULL);

update\_implicit\_persistent\_segment\_tree(tree[i],tree[i+1],1,MAX\_VALUE,v,-1);

}

}

else if(input[i].type=="query"){

tree[i+1]=tree[i];

tree\_one[i+1]=tree\_one[i];

int v;

if(M[input[i].op] == 0) {

counter\_string++;

M[input[i].op] = counter\_string;

v = -1;

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

fflush(stdout);

return;

}

else{

v = query\_persistent\_segment\_tree(tree\_one[i+1],1,MAX\_STRING,M[input[i].op]);

if(v==0){ //removed

v = -1;

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

fflush(stdout);

return;

}

}

int res = query\_implicit\_persistent\_segment\_tree(tree[i+1],1,MAX\_VALUE,1,v-1);

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

fflush(stdout);

}

else if(input[i].type=="undo") {

tree[i+1]=tree[i-input[i].priority];

tree\_one[i+1] = tree\_one[i-input[i].priority];

}

//}

}

void jamie\_and\_to\_do(){

counter\_string = 0;

tree\_one[0] = new persistent\_segment\_tree\_info(0,NULL,NULL);

build\_persisent\_segment\_tree(tree\_one[0],1,MAX\_STRING);

tree[0] = new implicit\_persistent\_segment\_tree\_info(0,NULL,NULL);

int n;

scanf("%d",&n);

//cout<<n<<endl;

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

string a,b;

cin>>a;

Query temp;

temp.type=a;

if(a=="set"){

cin>>b;

temp.op=b;

//cout<<b<<endl;

int pr;

scanf("%d",&pr);

temp.priority=pr;

}

else if(a=="remove"){

cin>>b;

temp.op=b;

temp.priority=0;

}

else if(a=="query"){

cin>>b;

temp.op=b;

temp.priority=0;

}

else if(a=="undo"){

int pr;

scanf("%d",&pr);

temp.priority=pr;

}

input.push\_back(temp);

soln(i);

}

//string\_map();

//soln();

return;

}

int main(){

//freopen("in.txt","r",stdin);

jamie\_and\_to\_do();

return 0;

}

/**/Lowest Commmon Ancestor + Bridge Tree**

**Tested on Hacker rank sherlock and the queries**

#include <bits/stdc++.h>

using namespace std;

#define MAX 100005

//input

vector<int>graph[MAX+1];

//Formation of Bridge Tree

vector<int>corres[MAX+5]; //corresponding edges

vector<int>articulationBridge[MAX+5]; //says this edge is articulation bridge or not

//to find articulation bridge

int color[MAX+1];

int Time;

int disc[MAX+1];

int low[MAX+1];

int parent[MAX+1];

void find\_articulation\_bridge(int node){

color[node] = 1; //Grey

Time++;

disc[node] = Time;

low[node] = disc[node];

for(int i=0;i<graph[node].size();i++) {

int v = graph[node][i];

if(color[v] == 0) {

parent[v] = node;

find\_articulation\_bridge(v); //dfs call

low[node] = min(low[node],low[v]);

}

else if(color[v] != 0 && v != parent[node]) {

low[node] = min(low[node],disc[v]);

}

if(disc[node]<low[v]) {

//this means this child has no previous upper shorter linkage this edge can be considered as a bridge

//so this edge can be considered as an articulation bridge

articulationBridge[node][i] = 1;

int u = corres[node][i];

articulationBridge[v][u] = 1; //same edge from opposite node

}

}

color[node] = 2;

return;

}

void initialize\_articulation\_bridge(){

memset(color,0,sizeof(color));

Time = 0;

memset(disc,0,sizeof(disc));

memset(low,0,sizeof(low));

memset(parent,0,sizeof(parent));

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

corres[i].clear();

articulationBridge[i].clear();

}

}

//this function makes the bridge tree

int num\_compo; //represents nth bridge

vector<int>tree[MAX+5]; //the bridge nodes

vector<int>tree\_edge[MAX+5];//the tree edge of the graph

int visit[MAX+5];

//identifier

int node\_id[MAX+1]; //this says a node belongs to which tree node

void make\_bridge\_tree(int node){

int present\_num = num\_compo;

visit[node] = 1;

tree[num\_compo].push\_back(node);

queue<int>Q;

Q.push(node);

node\_id[node] = present\_num;

while(Q.empty() != true) {

int u = Q.front();

Q.pop();

for(int i=0;i<graph[u].size();i++) {

int v = graph[u][i];

if(visit[v] == 1) continue;

else {

if(articulationBridge[u][i] == 1) {

num\_compo++;

tree\_edge[present\_num].push\_back(num\_compo);

tree\_edge[num\_compo].push\_back(present\_num);

node\_id[v] = num\_compo;

make\_bridge\_tree(v);

}

else {

tree[present\_num].push\_back(v);

visit[v] = 1;

Q.push(v);

node\_id[v] = present\_num;

}

}

}

}

}

struct Lowest\_Common\_Ancestor{

int sparse\_table[MAX+1][18];

int visit[MAX+1];

int level[MAX+1];

int parent[MAX+1];

int max\_depth;

int save;//says max power of 2

void bfs(Lowest\_Common\_Ancestor &node,int start,vector<int>g[MAX+1]){

queue<int>Q;

Q.push(start);

node.visit[start] = 1;

node.level[start] = 0;

node.max\_depth = 0;

while(Q.empty() != true) {

int u = Q.front();

Q.pop();

for(int i=0;i<g[u].size();i++) {

int v = g[u][i];

if(node.visit[v] == 0) {

node.visit[v] = 1;

Q.push(v);

node.level[v] = node.level[u]+1;

node.max\_depth = max(node.max\_depth,node.level[v]);

node.parent[v] = u;

}

}

}

}

//void calculate LCA

void calculate\_sparse\_table(Lowest\_Common\_Ancestor &node,int tot\_node){

//0th parent

for(int i=1;i<=tot\_node;i++) {

node.sparse\_table[i][0] = node.parent[i];

}

//other parents 1<<2

for(int i=1;(1<<i) <= node.max\_depth; i++){

save=i;

for(int j=1;j<=tot\_node;j++){

if(node.sparse\_table[j][i-1] != -1) {

node.sparse\_table[j][i] = node.sparse\_table[node.sparse\_table[j][i-1]][i-1];

}

}

}

}

//void find lowest common ancestor

//will find LCA of p and q

int find\_lca(Lowest\_Common\_Ancestor &node, int p, int q){

//highest level will be in p

if(node.level[q]>node.level[p]) {

swap(p,q);

}

//to bring p,q in same level

for(int i=save;i>=0;i--){

if((node.level[p] - (1<<i)) >= (node.level[q])) {

p=sparse\_table[p][i];

}

}

**/**/now calculate LCA

for(int i=save;i>=0;i--){

//taking them up

if(sparse\_table[p][i] != -1 && sparse\_table[p][i] != sparse\_table[q][i]) {

p=sparse\_table[p][i];

q=sparse\_table[q][i];

}

}

if(p==q) return p;

return sparse\_table[p][0];

}

bool is\_ancestor(Lowest\_Common\_Ancestor &node, int u, int v){

//says if u is ancestor of v

if(node.level[u]<=node.level[v]) return true;

return false;

}

pair<int,int> find\_common\_path(Lowest\_Common\_Ancestor &node, int u, int a, int v, int b){

//cout<<"common path " << u<<" "<<a<<" "<<v<<" "<<b<<endl;

bool ok = node.is\_ancestor(node,v,a);

if(!ok) {

//if v is not ancestor of a no common path can't exist in(u->a) and (v->b)

pair<int,int>temp;

temp.first=0;

temp.second=0;

return temp;

}

int x = node.find\_lca(node,a,b);

//cout<<"x = " << x<<" a " << a<<" "<<b <<endl;

if(node.level[u]<node.level[v]) {

//common path must start from v then

ok = node.is\_ancestor(node,v,x);

if(ok) {

pair<int,int>temp;

temp.first=v;

temp.second=x;

return temp;

}

}

else {

ok = node.is\_ancestor(node,u,x);

if(ok) {

pair<int,int>temp;

temp.first=u;

temp.second=x;

return temp;

}

}

pair<int,int>temp;

temp.first=0;

temp.second=0;

return temp;

}

**/**/this function determines the distance between 2 nodes

int dist(Lowest\_Common\_Ancestor &node,int a,int b){

int lca = node.find\_lca(node,a,b);

int value = node.level[a]+node.level[b] - 2\*node.level[lca];

return value;

}

void init(Lowest\_Common\_Ancestor &node){

memset(node.visit,0,sizeof(node.visit));

memset(node.sparse\_table,-1,sizeof(node.sparse\_table));

memset(node.level,0,sizeof(node.level));

memset(node.parent,-1,sizeof(node.parent));

}

};

Lowest\_Common\_Ancestor obj;

void initialize\_bridge\_tree(){

num\_compo = 1;

memset(visit,0,sizeof(visit));

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

tree[i].clear();

tree\_edge[i].clear();

}

}

int main(){

initialize\_articulation\_bridge();

initialize\_bridge\_tree();

//freopen("in.txt","r",stdin);

int n,m,q;

scanf("%d %d %d",&n,&m,&q);

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

int a,b;

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

graph[a].push\_back(b);

graph[b].push\_back(a);

articulationBridge[a].push\_back(0);

articulationBridge[b].push\_back(0);

corres[a].push\_back(graph[b].size()-1);

corres[b].push\_back(graph[a].size()-1);

}

find\_articulation\_bridge(1);

//now making bridge tree

make\_bridge\_tree(1);

//LCA

obj.init(obj);

obj.bfs(obj,1,tree\_edge);

obj.calculate\_sparse\_table(obj,num\_compo);

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

int a,b,c,d;

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

int c1 = node\_id[a];

int c2 = node\_id[b];

if(c1 == c2) {

//so articulation bridge condition doesn't change

int c3 = node\_id[c];

int c4 = node\_id[d];

if(c3 == c4) {

//stays in same composed node

printf("0\n");

}

else {

int lca = obj.find\_lca(obj,c3,c4);

//sum of the nodes distance

int ans = obj.level[c3] + obj.level[c4] - 2 \* obj.level[lca];

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

}

}

else {

//articulation bridge condition change

int lca = obj.find\_lca(obj,c1,c2);

//cout<<"c1 = "<<c1<<" "<<c2<<" "<<lca<<endl;

int c3 = node\_id[c];

int c4 = node\_id[d];

int lca2 = obj.find\_lca(obj,c3,c4);

//cout<<"c3 = "<<c3<<" "<<c4<<" "<<lca2<<endl;

int res = obj.level[c3] + obj.level[c4] - 2 \* (obj.level[lca2]);

//common path

//(lca2->c3),(lca->c1);

pair<int,int>temp;

temp=obj.find\_common\_path(obj,lca2,c3,lca,c1);

if(temp.first != 0 && temp.second != 0) {

//cout<<"1 " <<temp.first<<" "<<temp.second<<endl;

int temp\_lca = obj.find\_lca(obj,temp.first,temp.second);

int value = obj.level[temp.first]+obj.level[temp.second] - 2\*obj.level[temp\_lca];

res=res-value;

}

//(lca2,c3) (lca,c2);

temp=obj.find\_common\_path(obj,lca2,c3,lca,c2);

if(temp.first != 0 && temp.second != 0) {

//cout<<"2 " <<temp.first<<" "<<temp.second<<endl;

int temp\_lca = obj.find\_lca(obj,temp.first,temp.second);

int value = obj.level[temp.first]+obj.level[temp.second] - 2\*obj.level[temp\_lca];

res=res-value;

}

//(lca2,c4) (lca,c1)

temp=obj.find\_common\_path(obj,lca2,c4,lca,c1);

if(temp.first != 0 && temp.second != 0) {

//cout<<"3 " <<temp.first<<" "<<temp.second<<endl;

int temp\_lca = obj.find\_lca(obj,temp.first,temp.second);

int value = obj.level[temp.first]+obj.level[temp.second] - 2\*obj.level[temp\_lca];

res=res-value;

}

//(lca2,c4) (lca,c2)

temp=obj.find\_common\_path(obj,lca2,c4,lca,c2);

if(temp.first != 0 && temp.second != 0) {

//cout<<"4 " <<temp.first<<" "<<temp.second<<endl;

int temp\_lca = obj.find\_lca(obj,temp.first,temp.second);

int value = obj.level[temp.first]+obj.level[temp.second] - 2\*obj.level[temp\_lca];

res=res-value;

}

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

}

}

}

//Centeroid Decomposition

//implementation of centeroid decomposition

struct Centeroid\_Decomposition{

set<int>centeroid\_tree[MAX+1]; //centeroid tree

int node; //total node number

int nn;

int subtree[MAX+1];

int parent[MAX+1];

//extra for problem

int rank[MAX+1];

//to find the subtree size

void dfs0(Centeroid\_Decomposition &obj,int u, int p){

obj.subtree[u]=1; //subtree rooted at u

obj.nn++; //total node number in this subtree

for(set<int>::iterator it=obj.centeroid\_tree[u].begin();it != obj.centeroid\_tree[u].end();it++){

int v = \*it;

if(v != p) { //v is not parent of u

obj.dfs0(obj,v,u);

obj.subtree[u] += obj.subtree[v];

}

}

return;

}

//to find the centeroid

int dfs1(Centeroid\_Decomposition &obj,int u,int p){

for(set<int>::iterator it=obj.centeroid\_tree[u].begin();it != obj.centeroid\_tree[u].end();it++){

int v = \*it;

if(v != p) { // v is not p or parent of u

if(obj.subtree[v] > obj.nn/2){ //this subtree rooted at v, contains more than half of total node of the main tree , so we need decomposition

return dfs1(obj,v,u);

}

}

}

return u;

}

//u where to start

//p parent of u/centeroid of u

void decompose(Centeroid\_Decomposition &obj,int u, int p,int level){

obj.nn=0;

dfs0(obj,u,p); //to find the total subtree size nn and to find each subtree size

int centeroid = obj.dfs1(obj,u,p); //getting centeroid

if(p == -1) p=centeroid; //for root case

obj.parent[centeroid]=p; //parent of centeroid is anoter centeroid

//debug: cout<<"centeroids "<<centeroid<< " level = "<<level<<endl;

obj.rank[centeroid]=level;

for(set<int>::iterator it=obj.centeroid\_tree[centeroid].begin();it != obj.centeroid\_tree[centeroid].end();it++){

int v = \*it;

obj.centeroid\_tree[v].erase(centeroid); //from centeroid connection changes, only connected to other centeroid

obj.decompose(obj,v,centeroid,level+1); //new decomposition starts

}

return;

}

void process\_start(Centeroid\_Decomposition &obj){

obj.decompose(obj,1,-1,0);

}

void init(Centeroid\_Decomposition &obj,int n,set<int>V[MAX+1]){

obj.node = n;

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

obj.centeroid\_tree[i].insert(V[i].begin(),V[i].end());

}

memset(obj.parent,-1,sizeof(obj.parent));

}

};

//Lowest Common Ancestors

struct Lowest\_Common\_Ancestor{

int sparse\_table[MAX+1][18];

int visit[MAX+1];

int level[MAX+1];

int parent[MAX+1];

int max\_depth;

int save;//says max power of 2

void bfs(Lowest\_Common\_Ancestor &node,int start,vector<int>g[MAX+1]){

queue<int>Q;

Q.push(start);

node.visit[start] = 1;

node.level[start] = 0;

node.max\_depth = 0;

while(Q.empty() != true) {

int u = Q.front();

Q.pop();

for(int i=0;i<g[u].size();i++) {

int v = g[u][i];

if(node.visit[v] == 0) {

node.visit[v] = 1;

Q.push(v);

node.level[v] = node.level[u]+1;

node.max\_depth = max(node.max\_depth,node.level[v]);

node.parent[v] = u;

}

}

}

}

//void calculate LCA

void calculate\_sparse\_table(Lowest\_Common\_Ancestor &node,int tot\_node){

//0th parent

for(int i=1;i<=tot\_node;i++) {

node.sparse\_table[i][0] = node.parent[i];

}

//other parents 1<<2

for(int i=1;(1<<i) <= node.max\_depth; i++){

save=i;

for(int j=1;j<=tot\_node;j++){

if(node.sparse\_table[j][i-1] != -1) {

node.sparse\_table[j][i] = node.sparse\_table[node.sparse\_table[j][i-1]][i-1];

}

}

}

}

//void find lowest common ancestor

//will find LCA of p and q

int find\_lca(Lowest\_Common\_Ancestor &node, int p, int q){

//highest level will be in p

if(node.level[q]>node.level[p]) {

swap(p,q);

}

//to bring p,q in same level

for(int i=save;i>=0;i--){

if((node.level[p] - (1<<i)) >= (node.level[q])) {

p=sparse\_table[p][i];

}

}

//now calculate LCA

for(int i=save;i>=0;i--){

//taking them up

if(sparse\_table[p][i] != -1 && sparse\_table[p][i] != sparse\_table[q][i]) {

p=sparse\_table[p][i];

q=sparse\_table[q][i];

}

}

if(p==q) return p;

return sparse\_table[p][0];

}

//this function determines the distance between 2 nodes

int dist(Lowest\_Common\_Ancestor &node,int a,int b){

int lca = node.find\_lca(node,a,b);

int value = node.level[a]+node.level[b] - 2\*node.level[lca];

return value;

}

void init(Lowest\_Common\_Ancestor &node){

memset(node.visit,0,sizeof(node.visit));

memset(node.sparse\_table,-1,sizeof(node.sparse\_table));

memset(node.level,0,sizeof(node.level));

memset(node.parent,-1,sizeof(node.parent));

}

};

// Disjoint set union(DSU)

#include <bits/stdc++.h>

using namespace std;

#define MAX 100005

map<string,int>M;

int parent[MAX+5];

int totalConn[MAX+5];

int findParent(int a){

if(parent[a] == a) return a;

parent[a]=findParent(parent[a]);

return parent[a];

}

int main(void){

//freopen("in.txt","r",stdin);

//freopen("out.txt","w",stdout);

int T,t;

scanf("%d",&T);

int n;

string a,b;

int cnt=0;

for(t=1;t<=T;t++){

scanf("%d",&n);

M.clear();

cnt=0;

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

parent[i]=i;

totalConn[i]=1;

}

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

cin >>a>>b;

if(M[a]==0){

cnt++;

M[a]=cnt;

}

if(M[b] == 0){

cnt++;

M[b]=cnt;

}

int p\_a=findParent(M[a]);

int p\_b=findParent(M[b]);

if(p\_a == p\_b) {

cout<<totalConn[p\_a]<<endl;

}

else{

parent[p\_a]=p\_b;

totalConn[p\_b] += totalConn[p\_a];

cout<<totalConn[p\_b]<<endl;

}

}

}

return 0;

}