DU_Epinephrine
-Maxon5

Adaptive Simpson:

```
/*
    For finding the length of an arc in a range
    L = integrate(ds) from start to end of range
    where ds = sqrt(1+(d/dy(x))^2)dy
*/
const double SIMPSON_TERMINAL_EPS = 1e-12;
/// Function whose integration is to be calculated
double F(double x);
double simpson(double minx, double maxx)
{
    return (maxx - minx) / 6 * (F(minx) + 4 * F((minx + maxx) / 2.) +
F(maxx));
}
double adaptive_simpson(double minx, double maxx, double c, double EPS)
{
//    if(maxx - minx < SIMPSON_TERMINAL_EPS) return 0;

    double midx = (minx + maxx) / 2;
    double a = simpson(minx, midx);
    double b = simpson(midx, maxx);

    if(fabs(a + b - c) < 15 * EPS) return a + b + (a + b - c) / 15.0;

    return adaptive_simpson(minx, midx, a, EPS / 2.) +
adaptive_simpson(midx, maxx, b, EPS / 2.);
}
double adaptive_simpson(double minx, double maxx, double EPS)
{
    return adaptive_simpson(minx, maxx, simpson(minx, maxx), EPS);
}
```

Aho Corasick:

```
/// Call initRank() before doing anything
/// O(nlogn) for inserting n characters as pattern dynamically in the
```

```
dynamic aho corasick
/// Look at init() inside ahoCorasick if you need

/// NEVER FORGET TO CALL BUILD()

/// After build() is done, trie[i].edge[j] will contain an original edge if
the character j is
/// reachable directly from i.
/// Otherwise, trie[i].edge[j] will contain an edge where you will go after
traversing through failurelinks
/// till you reach a position where a node has an edge with character j or
the node is root.
/// So trie will not remain a tree, it will become a DAG except for the
root. There can be cycles containing the root.
/// Change inside build to keep it as a tree.
#define MAX     ?
#define ALPHABET_SIZE ?
const int LOG = log2(MAX)+5;
int Rank[300];
void initRank()
{
    for(char c = 'a'; c<='z'; c++)
        Rank[c] = c-'a';
}

struct ahoCorasick{
    struct node{
        int edge[ALPHABET_SIZE];
        int failLink;
        int counter; /// change this variable to whatever you need to do
    }dummy;

    bool alive;
    vector<node>trie;
    vector<string>dictionary;
    void Clear()
    {
        trie.clear();
        dictionary.clear();
        alive = false;
    }
```

```
ahoCorasick()
{
    alive = false;
}

void init()
{
    alive = true;
    dummy.failLink = 0;
    dummy.counter = 0;  /// set the dummy's value appropriately
    mem(dummy.edge, -1);
    trie.push_back(dummy);
}
void add(string &s)
{
    dictionary.pb(s);
    int len = s.length();
    int cur = 0;
    for(int i = 0; i<len; i++)
    {
        int nw = Rank[s[i]];
        if(trie[cur].edge[nw] == -1)
        {
            trie.pb(dummy);
            trie[cur].edge[nw] = trie.size()-1;
        }
        cur = trie[cur].edge[nw];
    }
    trie[cur].counter++;  /// change this line as you need
}

inline void build()
{
    queue<int>Q;
    trie[0].failLink = 0;
    for(int i = 0; i<ALPHABET_SIZE; i++)
    {
        if(trie[0].edge[i] == -1)
            trie[0].edge[i] = 0;
        else
        {
            /// Remove the following line to keep aho as tree
```

```
                trie[trie[0].edge[i]].failLink = 0;
                Q.push(trie[0].edge[i]);
            }
        }
        while(!Q.empty())
        {
            int cur = Q.front();
            Q.pop();
            for(int i = 0; i<ALPHABET_SIZE; i++)
            {
                if(trie[cur].edge[i] != -1)
                {
                    int v = trie[cur].edge[i];
                    trie[v].failLink = trie[trie[cur].failLink].edge[i];
                    trie[v].counter += trie[trie[v].failLink].counter; ///
change this line as you need
                    Q.push(v);
                }
                else
                {
                    /// Remove the following line to keep aho as tree
                    trie[cur].edge[i] = trie[trie[cur].failLink].edge[i];
                }
            }
        }
    }

    /// The following traverse function checks if a text has at least one
pattern as a substring
    /// change it as you need
    int traverse(string &text)
    {
        int len = text.length();
        int cur = 0;
        for(int i = 0; i<len; i++)
        {
            int nw = Rank[text[i]];
            cur = trie[cur].edge[nw];
            if(trie[cur].counter)
                return true;
        }
        return false;
    }

    int query(string &text)
    {
        return traverse(text);
        /// add whatever you need to do after traversing in the query
    }
};

struct dynamicAho{
    ahoCorasick ar[LOG];
    void add(string &pattern)
    {
        int myIdx = -1;
        for(int i = 0; i<LOG; i++)
        {
            if(ar[i].dictionary.size() == 0)
            {
                myIdx = i;
                break;
            }
        }
//        assert(myIdx != -1);
        ar[myIdx].init();
        ar[myIdx].add(pattern);
        for(int j = 0; j<myIdx; j++)
        {
            for(auto s: ar[j].dictionary)
                ar[myIdx].add(s);
            ar[j].Clear();
        }
        ar[myIdx].build();
    }

    int query(string &s)
    {
        for(int i=  0; i<LOG; i++)
        {
            if(ar[i].alive && ar[i].query(s))
                return true;
        }
        return false;
```

```
     }
 }AC;

 int main()
 {
     initRank();
 }
```

Articulation Point:

```
/// Articulation Point
/// n is the number of nodes
/// AP will contain the articulation points
/// 1 based indexing

vector<int> E[MAX+10];
bool vis[MAX+10], pushed[MAX+10];
int disc[MAX+10], low[MAX+10], parent[MAX+10], n;
vector <int> AP;

///ARTICULATION POINT
bool is_AP(int u, int v, int children)
{
    if(parent[u] == -1)
    {
        /// u is root and has two children who have no connection except the
root. so it's an articulation point.
        if(children > 1 )
            return true;
        return false;
    }
    else
    {
        /// There is no backedge towards u's ancestor from any node of the
subtree from u going through v and u is not a root.
        /// so u must be an articulation point
        if(low[v] >= disc[u] )
            return true;
        return false;
    }
}

void dfs(int node)
{
    vis[node]=1;
    if(parent[node] == -1)
        disc[node] = low[node] = 1;
    else
        disc[node] = low[node] = disc[parent[node]]+1;

    int children = 0, v, u = node;
    for(int i = 0; i<E[u].size();i++)
    {
        v = E[u][i];
        if( !vis[v] )
        {
            children++;
            parent[v]=u;
            dfs(v);
            low[u]=min(low[u],low[v]);
            if(is_AP(u, v, children) && pushed[u] == 0)
            {
                pushed[u] = 1;
                AP.pb(u);
            }
        }
        else if(v!=parent[u])
            low[u]=min(low[u],disc[v]);
    }
    return;
}

void find_articulation_point()
{
    mem(vis, 0);
    mem(pushed, 0);
    mem(parent, -1);
    AP.clear();
    for(int i = 1; i<=n; i++)
    {
        if(!vis[i])
```

```
            dfs(i);
    }
}
```

Biconnected Component:

```
/// Decomposes the graph into  Biconnected Components and also finds the
articulation points
/// n is the number of nodes
/// BCC[i] will contain the nodes of i'th BCC
/// AP will contain the articulation points
/// 1 based indexing

vector<int> E[MAX+10], stck, BCC[MAX+10], AP;
int n, m, low[MAX+10], disc[MAX+10], BCC_cnt, parent[MAX+10];
bool vis[MAX+10], pushed[MAX+10];

bool is_AP(int u, int v, int children)
{
    if(parent[u] == -1)
    {
        /// u is root and has two children who have no connection except the
root. so it's an articulation point.
        if(children > 1)
            return true;
        return false;
    }
    else
    {
        /// There is no backedge towards u's ancestor from any node of the
subtree from u going through v and u is not a root.
        /// so u must be an articulation point
        if(low[v] >= disc[u])
            return true;
        return false;
    }
}

void dfs(int node)
{
    vis[node] = 1;
    stck.push_back(node);
    if(parent[node] == -1)
        disc[node] = low[node] = 1;
    else
        disc[node] = low[node] = disc[parent[node]]+1;

    int children = 0, v, u = node;
    for(int i = 0; i<E[u].size();i++)
    {
        v = E[u][i];
        if( !vis[v] )
        {
            children++;
            parent[v]=u;
            dfs(v);
            low[u]=min(low[u],low[v]);
            if(is_AP(u, v, children))
            {
                if(!pushed[u])
                    AP.pb(u);
                pushed[u] = 1;
                BCC_cnt++;
                BCC[BCC_cnt].push_back(u);
                while(true)
                {
                    int nw = stck.back();
                    BCC[BCC_cnt].pb(nw);
                    stck.pop_back();
                    if(nw == v)
                        break;
                }
            }
        }
        else if( v !=parent[u])
            low[u] = min(low[u],disc[v]);
    }
    return;
}


void find_BCC()
{
    for(int i = 0; i<=MAX; i++)
        BCC[i].clear();
    mem(parent, -1);
    BCC_cnt = 0;
    mem(vis, 0);
    AP.clear();
    mem(pushed, 0);
    for(int i = 1; i<=n; i++)
    {
        if(!vis[i])
        {
            dfs(i);
            if(!stck.empty())
                BCC_cnt++;
            while(!stck.empty())
            {
                BCC[BCC_cnt].pb(stck.back());
                stck.pop_back();
            }
        }

    }
}
```

Bellman Ford:

```
/*
    Detects in O(VE) if there is a negative cycle reachable from given
source.
    Run bellman taking every node as source to find if there exists any
negative cycle in the graph in O(V^2*E).
    Also finds shortest distance to each node from the given source.
*/
#define MAX         ?
const LL INF = 1e80;

struct edgeData{
    int u, v;
    LL cost;
};
vector<edgeData>edge;
LL dist[MAX+10];

/// cycleNodes will contain at least one node from a negative cycle if there
```

```
is any after bellman finishes.

bool bellmanFord(int n, int source, vector<edgeData>&edge,
vector<int>&cycleNodes)
{
    for(int i = 1; i<=n; i++)
        dist[i] = INF;
    dist[source] = 0;
    bool ret = false;
    for(int i = 1; i<=n; i++)
    {
        for(auto e: edge)
        {
            int u = e.u, v = e.v;
            LL cost = e.cost;
            if(dist[v] > dist[u] + e.cost + eps)
            {
                if(i == n)
                {
                    ret = true;
                    cycleNodes.push_back(v);
                }
                dist[v] = dist[u] + e.cost;

            }
        }
    }
    return ret;
}
```

Big Integer:

```
struct Bigint {
    // representations and structures
    string a; // to store the digits
    int sign; // sign = -1 for negative numbers, sign = 1 otherwise

    // constructors
    Bigint() {} // default constructor
    Bigint( string b ) { (*this) = b; } // constructor for string

    // some helpful methods
```

```
int size() { // returns number of digits
    return a.size();
}
Bigint inverseSign() { // changes the sign
    sign *= -1;
    return (*this);
}
Bigint normalize( int newSign ) { // removes leading 0, fixes sign
    for( int i = a.size() - 1; i > 0 && a[i] == '0'; i-- )
        a.erase(a.begin() + i);
    sign = ( a.size() == 1 && a[0] == '0' ) ? 1 : newSign;
    return (*this);
}

// assignment operator
void operator = ( string b ) { // assigns a string to Bigint
    a = b[0] == '-' ? b.substr(1) : b;
    reverse( a.begin(), a.end() );
    this->normalize( b[0] == '-' ? -1 : 1 );
}

// conditional operators
bool operator < ( const Bigint &b ) const { // less than operator
    if( sign != b.sign ) return sign < b.sign;
    if( a.size() != b.a.size() )
        return sign == 1 ? a.size() < b.a.size() : a.size() > b.a.size();
    for( int i = a.size() - 1; i >= 0; i-- ) if( a[i] != b.a[i] )
        return sign == 1 ? a[i] < b.a[i] : a[i] > b.a[i];
    return false;
}
bool operator == ( const Bigint &b ) const { // operator for equality
    return a == b.a && sign == b.sign;
}



// mathematical operators
Bigint operator + ( Bigint b ) { // addition operator overloading
    if( sign != b.sign ) return (*this) - b.inverseSign();
    Bigint c;
    for(int i = 0, carry = 0; i<a.size() || i<b.size() || carry; i++ ) {
        carry+=(i<a.size() ? a[i]-48 : 0)+(i<b.a.size() ? b.a[i]-48 : 0);
        c.a += (carry % 10 + 48);
```

```
            carry /= 10;
        }
        return c.normalize(sign);
    }
    Bigint operator - ( Bigint b ) { // subtraction operator overloading
        if( sign != b.sign ) return (*this) + b.inverseSign();
        int s = sign; sign = b.sign = 1;
        if( (*this) < b ) return ((b - (*this)).inverseSign()).normalize(-s);
        Bigint c;
        for( int i = 0, borrow = 0; i < a.size(); i++ ) {
            borrow = a[i] - borrow - (i < b.size() ? b.a[i] : 48);
            c.a += borrow >= 0 ? borrow + 48 : borrow + 58;
            borrow = borrow >= 0 ? 0 : 1;
        }
        return c.normalize(s);
    }
    Bigint operator * ( Bigint b ) { // multiplication operator overloading
        Bigint c("0");
        for( int i = 0, k = a[i] - 48; i < a.size(); i++, k = a[i] - 48 ) {
            while(k--) c = c + b; // ith digit is k, so, we add k times
            b.a.insert(b.a.begin(), '0'); // multiplied by 10
        }
        return c.normalize(sign * b.sign);
    }
    Bigint operator / ( Bigint b ) { // division operator overloading
        if( b.size() == 1 && b.a[0] == '0' ) b.a[0] /= ( b.a[0] - 48 );
        Bigint c("0"), d;
        for( int j = 0; j < a.size(); j++ ) d.a += "0";
        int dSign = sign * b.sign; b.sign = 1;
        for( int i = a.size() - 1; i >= 0; i-- ) {
            c.a.insert( c.a.begin(), '0');
            c = c + a.substr( i, 1 );
            while( !( c < b ) ) c = c - b, d.a[i]++;
        }
        return d.normalize(dSign);
    }
    Bigint operator % ( Bigint b ) { // modulo operator overloading
        if( b.size() == 1 && b.a[0] == '0' ) b.a[0] /= ( b.a[0] - 48 );
        Bigint c("0");
        b.sign = 1;
        for( int i = a.size() - 1; i >= 0; i-- ) {
            c.a.insert( c.a.begin(), '0');
            c = c + a.substr( i, 1 );
            while( !( c < b ) ) c = c - b;
        }
        return c.normalize(sign);
    }

    // output method
    void print() {
        if( sign == -1 ) putchar('-');
        for( int i = a.size() - 1; i >= 0; i-- ) putchar(a[i]);
    }
};
```

Binarize Any Tree:

```
/// 1 based indexing
/// Original Tree in G
/// Binary version created in E
/// call goBinarize with the number of nodes in the original tree. It returns
number of nodes in the binary tree.
#define MAX         100000
vector<int> E[MAX+10], G[MAX+10];
int nxt;
void binarize(int node, int par)
{
    if( (G[node].size() <= 3 && par != -1) || (G[node].size() <= 2 && par ==
-1))
    {
        for(auto v: G[node])
        {
            if(v != par)
            {
                E[node].pb(v);
                E[v].pb(node);
                binarize(v, node);
            }
        }
    }
    else
    {
        int st;
        for(int i = 0; i<G[node].size(); i++)
        {
            int v = G[node][i];
            if(v != par)
            {
                st = i;
                E[node].pb(v);
                E[v].pb(node);
                break;
            }
        }
        int u = node, v = ++nxt;
        E[u].pb(v);
        E[v].pb(u);
        for(int i = st+1; i<G[node].size(); i++)
        {
            int v = G[node][i];
            if(v != par)
            {
                E[nxt].pb(v);
                E[v].pb(nxt);

                u = nxt+1;
                E[nxt].pb(u);
                E[u].pb(nxt);
                nxt++;
            }
        }
        for(auto v: G[node])
        {
            if(v != par)
                binarize(v, node);
        }
    }
}

int goBinarize(int nNode)
{
    nxt = nNode;
    binarize(1, -1);
    return nxt;
}
```

```
int main()
{
    G[1].pb(2);
    G[2].pb(1);
    G[1].pb(3);
    G[3].pb(1);
    G[1].pb(4);
    G[4].pb(1);
    int a = goBinarize(4);
    D(a);
    for(int u = 1; u<=a; u++)
    {
        for(auto v: E[u])
            DD(u, v);
    }
    return 0;
}

/*
    Binary tree Output:
    u = 1   v = 2
    u = 1   v = 5
    u = 2   v = 1
    u = 3   v = 5
    u = 4   v = 6
    u = 5   v = 1
    u = 5   v = 3
    u = 5   v = 6
    u = 6   v = 5
    u = 6   v = 4
    u = 6   v = 7
    u = 7   v = 6
*/
```

BIT Point Update, Range Query:

```
#define MAX     100

// Point update_Range_query_BIT
// An array, suppose arr[MAX]
// 1 based indexing
// mxIdx is the highest index
LL BIT[MAX+10];
int mxIdx;

void init(int n)
{
    mxIdx = n;
    mem(BIT, 0);
}

void update(int idx, LL val) //single point update, arr[idx] += val
{
    while(idx <= mxIdx)
        BIT[idx] += val, idx += idx&-idx;
}

LL query(int idx) // single point query, cumulative sum from arr[1] to
arr[idx]
{
    LL ret = 0;
    while(idx)
        ret += BIT[idx], idx -= idx&-idx;
    return ret;
}
```

BIT Range Update Range Query:

```
// Range_update_Range_query_BIT
// An array, suppose arr[MAX]
// 1 based indexing
// mxIdx is the highest index
// If you want only single point update, use only BIT_ADD and the first two
functions
LL BIT_ADD[MAX+10];
LL BIT_SUB[MAX+10];
int mxIdx;

void init(int n)
{
    mxIdx = n;
    mem(BIT_ADD, 0);
    mem(BIT_SUB, 0);
}

void update(LL BIT[], int idx, LL val) //single point update, arr[idx] += val
{
    while(idx <= mxIdx)
        BIT[idx] += val, idx += idx&-idx;
}

LL query(LL BIT[], int idx) // single point query, cumulative sum from arr[1]
to arr[idx]
{
    LL ret = 0;
    while(idx)
        ret += BIT[idx], idx -= idx&-idx;
    return ret;
}


LL range_query(int L, int R) // cumulative sum from arr[L] to arr[R]
{
    LL ret = (R*query(BIT_ADD, R) - (L-1)*query(BIT_ADD, L-1)) -
(query(BIT_SUB, R) - query(BIT_SUB, L-1)) ;
    return ret;
}

void range_update(int L, int R, LL v) // For i = L to R, arr[i] += val
{
    update(BIT_ADD, L,v);
    update(BIT_ADD, R+1, -v);
    update(BIT_SUB, L, v*(L-1));
    update(BIT_SUB, R+1, -v*(R));
}
```

Block Cut Tree:

```
namespace BCT{
/*
    1 based indexing
    Basic idea:
```

```
    In the block cut tree(BCT), each BCC will be a node and each articulation
point will be a node.
    Each edge will be between an articulation point and a BCC whose part is
that articulation point.
    Remember, only articulation points can be part of more than one BCC.
Normal nodes will be part
    of only one BCC. So you need to do something with node u of NORMAL GRAPH,
you do that something
    with the node that represents BCC of u ( if u is not an articulation
point ) or the node that
    represents u ( if u is an articulation point ) in the BCT.
myBCC[u].back() will be the representative
    of u in both case. myBCC[] and tree[] are the things that you would need
to access for solving problems.
    isAP[u] will tell you if node u of the NEW TREE is an articulation point
or not.
    *** MAX must be twice as big as maximum number of nodes.
    *** _n is the number of nodes in initial graph. nNode will be the number
of nodes in the block cut tree.
*/
#define MAX     ?
vector<int> E[MAX+10], stck, BCC[MAX+10], AP, myBCC[MAX+10];
int n, m, low[MAX+10], disc[MAX+10], BCC_cnt, parent[MAX+10],
sizeOfBCC[MAX+10];
bool vis[MAX+10], pushed[MAX+10], isAP[MAX+10];
bool checkAP(int u, int v, int children)
{
    if(parent[u] == -1)
    {
        /// u is root and has two children who have no connection except the
root. so it's an articulation point.
        if(children > 1)
            return true;
        return false;
    }
    else
    {
        /// There is no backedge towards u's ancestor from any node of the
subtree from u going through v and u is not a root.
        /// so u must be an articulation point
        if(low[v] >= disc[u])
            return true;
        return false;
    }
}

void dfs(int node)
{
    vis[node] = 1;
    stck.push_back(node);
    if(parent[node] == -1)
        disc[node] = low[node] = 1;
    else
        disc[node] = low[node] = disc[parent[node]]+1;

    int children = 0, v, u = node;
    for(int i = 0; i<E[u].size();i++)
    {
        v = E[u][i];
        if( !vis[v] )
        {
            children++;
            parent[v]=u;
            dfs(v);
            low[u]=min(low[u],low[v]);
            if(checkAP(u, v, children))
            {
                if(!pushed[u])
                    AP.pb(u);
                pushed[u] = 1;
                BCC_cnt++;
                myBCC[u].push_back(BCC_cnt);
                BCC[BCC_cnt].push_back(u);
                while(true)
                {
                    int nw = stck.back();
                    myBCC[nw].pb(BCC_cnt);
                    BCC[BCC_cnt].pb(nw);
                    stck.pop_back();
                    if(nw == v)
                        break;
                }
            }
        }
        else if( v !=parent[u])
            low[u] = min(low[u],disc[v]);
    }
    return;
}


void find_BCC()
{
    for(int i = 0; i<=MAX; i++)
        BCC[i].clear(), myBCC[i].clear();
    mem(parent, -1);
    BCC_cnt = 0;
    mem(vis, 0);
    AP.clear();
    mem(pushed, 0);
    for(int i = 1; i<=n; i++)
    {
        if(!vis[i])
        {
            dfs(i);
            if(!stck.empty())
                BCC_cnt++;
            while(!stck.empty())
            {
                myBCC[stck.back()].pb(BCC_cnt);
                BCC[BCC_cnt].pb(stck.back());
                stck.pop_back();
            }
        }
    }
}

vector<int>tree[MAX+10];
void createBlockCutTree(int _n, int &nNode)
{
    mem(isAP, 0);
    for(int i = 1; i<=MAX; i++)
        tree[i].clear();
    n = _n;
    find_BCC();
    for(int i = 1; i<=BCC_cnt; i++)
        sizeOfBCC[i] = BCC[i].size();
    for(int i = 1; i<=n; i++)
    {
        if(myBCC[i].size() > 1)
        {
            BCC_cnt++;
            for(int j = 0; j<myBCC[i].size(); j++)
            {
                int u = myBCC[i][j];
                int v = BCC_cnt;
                tree[u].push_back(v);
                tree[v].push_back(u);

                sizeOfBCC[u]--;
            }
            myBCC[i].push_back(BCC_cnt);
            BCC[BCC_cnt].pb(i);
            isAP[BCC_cnt] = true;
            sizeOfBCC[BCC_cnt] = 1;
        }
    }
    nNode = BCC_cnt;
}
}
```

BPM:

```
/// Kuhn's BPM
/// O(V*E)
/// n is the number of nodes on the left side
/// 1 based indexing
#define MAX ?
int Left[MAX+10], Right[MAX+10];
vector<int> E[MAX+10];
bool vis[MAX+10];

bool dfs(int node)
{
    if(vis[node])
        return false;
    vis[node] = 1;
    for(int i = 0; i<E[node].size(); i++)
    {
        int v = E[node][i];
        if(Right[v] == -1)
        {
            Left[node] = v;
            Right[v] = node;
            return true;
        }
    }
    for(int i = 0; i<E[node].size(); i++)
    {
        int v = E[node][i];
        if(dfs(Right[v]))
        {
            Left[node] = v;
            Right[v] = node;
            return true;
        }
    }
    return false;
}
int BPM(int n)
{
    mem(Left,-1);
    mem(Right,-1);
    bool done = false;
    while(!done)
    {
        done = true;
        mem(vis,0);
        for(int i = 1; i<=n; i++)
        {
            if(Left[i] == -1 && dfs(i))
                done = false;
        }
    }
    int ret = 0;
    for(int i = 1; i<=n; i++)
        ret += (Left[i] != -1);
    return ret;
}


int main()
{
    E[1].pb(2);
    E[1].pb(3);
    E[1].pb(1);

    E[2].pb(1);
    E[2].pb(2);

    E[3].pb(1);
    D(BPM(3));
}
```

Bridge:

```
/// Bridge
/// n is the number of nodes
/// bridge will contain the bridge edges
/// 1 based indexing

vector<int> E[MAX+10];
vector <pii> bridge;
bool vis[MAX+10];
int parent[MAX+10], disc[MAX+10], low[MAX+10], n;
void dfs(int node)
{
    vis[node] = 1;
    if(parent[node] == -1)
        disc[node] = low[node] = 1;
    else
        disc[node] = low[node] = disc[parent[node]]+1;
    int i, u, v;
    u = node;
    for(i = 0; i<E[u].size(); i++)
    {
        v = E[u][i];
        if(!vis[v])
        {
            parent[v] = u;
            dfs(v);
            low[u] = min( low[u], low[v] );
            if(low[v] > disc[u] )
                bridge.pb( {min(u,v), max(u,v)} );
        }
        else if(v!=parent[u])
            low[u] = min( low[u], disc[v] );
    }
}

void find_bridge()
{
    mem(parent, -1);
    mem(vis, 0);
    bridge.clear();
    for(int i = 1; i<=n; i++)
    {
        if(!vis[i])
            dfs(i);
    }
}
```

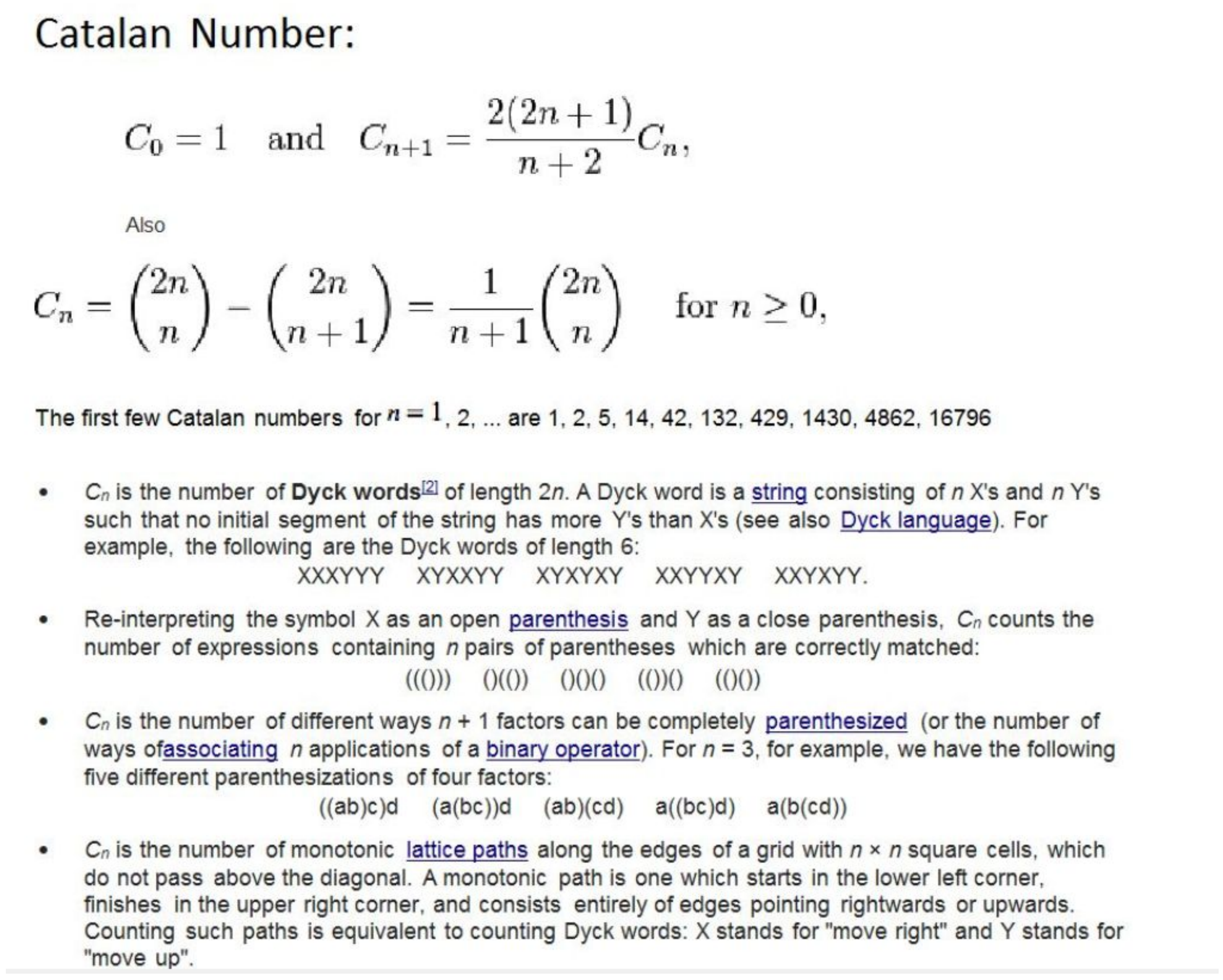
Bridge Tree:

```
/// Decomposes the graph into Bridge Tree and finds the bridges too
/// n is the number of nodes
/// bridge will contain the bridge edges
/// tree is the new graph with bridge trees
/// 1 based indexing
/// block_num[u] will contain the node whose part is the u-th node.

vector<int> E[MAX+10], stck, tree[MAX+10];
vector <pii> bridge;
bool vis[MAX+10];
int parent[MAX+10], disc[MAX+10], low[MAX+10], block_num[MAX+10], block_cnt,
n;
void dfs(int node)
{
    stck.push_back(node);
    vis[node] = 1;
    if(parent[node] == -1)
        disc[node] = low[node] = 1;
    else
        disc[node] = low[node] = disc[parent[node]]+1;
    int i, u, v;
    u = node;
    for(i = 0; i<E[u].size(); i++)
    {
        v = E[u][i];
        if(!vis[v])
        {
            parent[v] = u;
            dfs(v);
            low[u] = min( low[u], low[v] );
            if(low[v] > disc[u] )
            {
                bridge.pb( {min(u,v), max(u,v)} );
                block_cnt++;
                while(true)
                {
                    int nw = stck.back();
                    stck.pop_back();
                    block_num[nw] = block_cnt;
                    if(nw == v)
                        break;
                }
            }
        }
        else if(v!=parent[u])
            low[u] = min( low[u], disc[v] );
    }
}

void form_bridge_tree()
{
    for(int i = 0; i<=MAX; i++)
        tree[i].clear();
    block_cnt = 0;
    mem(parent, -1);
    mem(vis, 0);
    bridge.clear();
    for(int i = 1; i<=n; i++)
    {
        if(!vis[i])
        {
            dfs(i);
            block_cnt++;
            while(!stck.empty())
            {
                block_num[stck.back()] = block_cnt;
                stck.pop_back();
            }
        }
    }
    /// add edges in the bridge tree here.
    for(int i = 1; i<=n; i++)
    {
        for(int j = 0; j<E[i].size(); j++)
        {
            int v = E[i][j];
            if(block_num[i] != block_num[v])
                tree[block_num[i]].pb(block_num[v]);
        }
    }
}
```

Catalan Numbers:



Catalan Number:

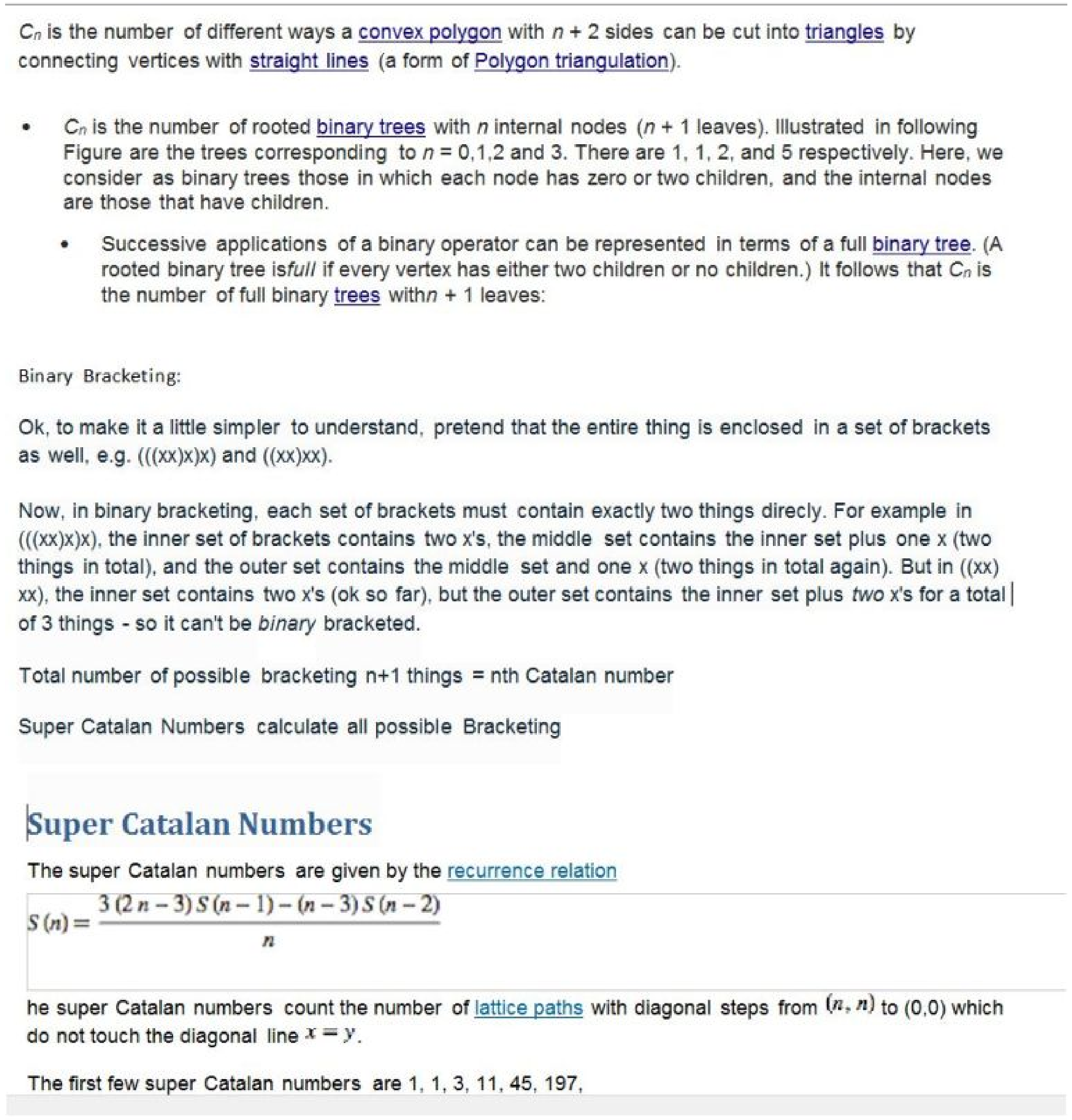
$$C_0 = 1 \quad \text{and} \quad C_{n+1} = \frac{2(2n+1)}{n+2} C_n,$$

Also

$$C_n = \binom{2n}{n} - \binom{2n}{n+1} = \frac{1}{n+1}\binom{2n}{n} \qquad \text{for } n \geq 0,$$

The first few Catalan numbers for $n = 1$, 2, ... are 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796

- C_n is the number of **Dyck words**[2] of length 2*n*. A Dyck word is a string consisting of *n* X's and *n* Y's such that no initial segment of the string has more Y's than X's (see also Dyck language). For example, the following are the Dyck words of length 6:
  XXXYYY XYXXYY XYXYXY XXYYXY XXYXYY.
- Re-interpreting the symbol X as an open parenthesis and Y as a close parenthesis, C_n counts the number of expressions containing *n* pairs of parentheses which are correctly matched:
  ((())) ()(()) ()()() (())() (()())
- C_n is the number of different ways *n* + 1 factors can be completely parenthesized (or the number of ways ofassociating *n* applications of a binary operator). For *n* = 3, for example, we have the following five different parenthesizations of four factors:
  ((ab)c)d (a(bc))d (ab)(cd) a((bc)d) a(b(cd))
- C_n is the number of monotonic lattice paths along the edges of a grid with *n* × *n* square cells, which do not pass above the diagonal. A monotonic path is one which starts in the lower left corner, finishes in the upper right corner, and consists entirely of edges pointing rightwards or upwards. Counting such paths is equivalent to counting Dyck words: X stands for "move right" and Y stands for "move up".



C_n is the number of different ways a convex polygon with $n + 2$ sides can be cut into triangles by connecting vertices with straight lines (a form of Polygon triangulation).

- C_n is the number of rooted binary trees with n internal nodes ($n + 1$ leaves). Illustrated in following Figure are the trees corresponding to $n = 0,1,2$ and 3. There are 1, 1, 2, and 5 respectively. Here, we consider as binary trees those in which each node has zero or two children, and the internal nodes are those that have children.
    - Successive applications of a binary operator can be represented in terms of a full binary tree. (A rooted binary tree is*full* if every vertex has either two children or no children.) It follows that C_n is the number of full binary trees with$n + 1$ leaves:

Binary Bracketing:

Ok, to make it a little simpler to understand, pretend that the entire thing is enclosed in a set of brackets as well, e.g. (((xx)x)x) and ((xx)xx).

Now, in binary bracketing, each set of brackets must contain exactly two things direcly. For example in (((xx)x)x), the inner set of brackets contains two x's, the middle set contains the inner set plus one x (two things in total), and the outer set contains the middle set and one x (two things in total again). But in ((xx)xx), the inner set contains two x's (ok so far), but the outer set contains the inner set plus *two* x's for a total of 3 things - so it can't be *binary* bracketed.

Total number of possible bracketing n+1 things = nth Catalan number

Super Catalan Numbers calculate all possible Bracketing

Super Catalan Numbers

The super Catalan numbers are given by the recurrence relation

$$S(n) = \frac{3(2n-3)S(n-1)-(n-3)S(n-2)}{n}$$

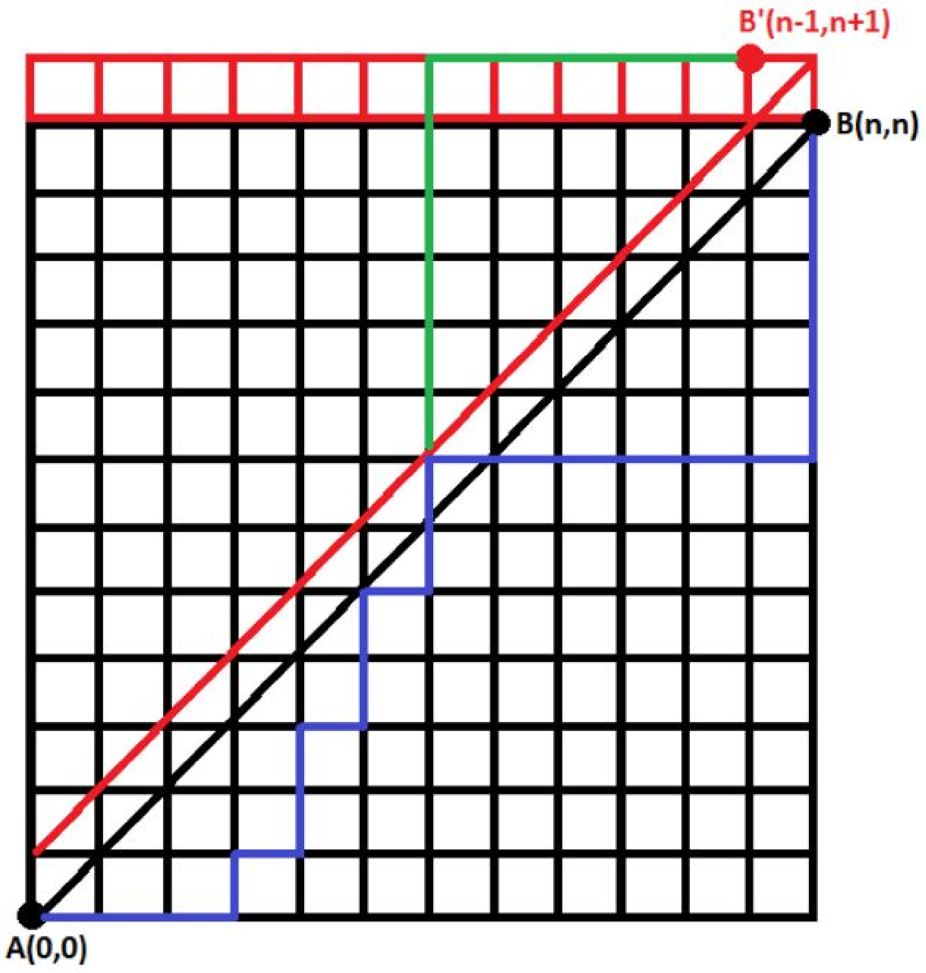
he super Catalan numbers count the number of lattice paths with diagonal steps from (n, n) to (0,0) which do not touch the diagonal line $x = y$.

The first few super Catalan numbers are 1, 1, 3, 11, 45, 197,

<u>Catalan Numbers(Numbers of ways to go to (n,n) from (0,0) without crossing main diagonal):</u>

2

We know that there are $\binom{2n}{n}$ ways of going (n,n) from $(0,0)$ when there is no restriction. So the answer should be $\binom{2n}{n} - B$ where B is the number of "bad paths", that is, number of paths that go above the diagonal line.



Now, in order to calculate B, we should notice something: When we go above the diagonal line, we will pass through another diagonal line, which is shown as *red* in the figure. And for each bad path (passes through the red line), let us create a new path which has the same moves until we go above the diagonal line (or pass through the red line), and then does the symmetric moves to our original path (original path is shown as *blue*, symmetric moves are shown in *green*). Therefore, number of bad paths become the number of paths from $(0,0)$ to $(n-1,n+1)$, which means $B = \binom{2n}{n+1}$. So the answer becomes

$$\binom{2n}{n} - \binom{2n}{n-1}$$

which is also called the n^{th} *Catalan Number*.

Centroid Decomposition:

```
/*
    1 based indexing
    Centroid Decomposition tree. (CDT)
    Complexity - O(Vlog(V)) , V is the number of nodes in the graph
    MAX is the maximum number of nodes.
    Just put the edges in E and call centroid_decomposition().
    Root is the root of the new tree.
    tree[] will hold the new tree.
    dist[i][v] = distance of v from its i'th level ancestor in CDT.
    myLevel[i] = level of i'th node in CDT.
    par[i] = parent of i'th node in CDT.
*/
#define MAX         ?
#define MAXLG       ?

vector<int> E[MAX+10], tree[MAX+10];
int taken[MAX+10], dist[MAXLG+5][MAX+10], myLevel[MAX+10], par[MAX+10];
pii mx[MAX+10];

int dfs1(int node, int par)
{
    int ret = 1;
    mx[node] = {0, node};
    for(auto v: E[node])
    {
        if(par == v || taken[v])
            continue;
        int nw = dfs1(v, node);
        mx[node] = max(mx[node], (pii){nw, v});
        ret += nw;
    }
    return ret;
}
int dfs2(int node, int nNode)
{
    if(mx[node].xx <= nNode/2)
        return node;
    return dfs2(mx[node].yy, nNode);
}
int getCentroid(int node)
```

```
{
    /// nNode = number of nodes in current tree
    int nNode = dfs1(node, 0);
    return dfs2(node, nNode);
}

void setDist(int node, int level, int d, int par)
{
    dist[level][node] = d;
    for(auto v: E[node])
    {
        if(taken[v] || v == par)
            continue;
        setDist(v, level, d+1, node);
    }
}

void build(int root, int L)
{
    setDist(root, L, 0, -1);
    myLevel[root] = L;
    for(auto v: E[root])
    {
        if(taken[v])
            continue;
        int nw = getCentroid(v);

        tree[root].pb(nw);
        par[nw] = root;
        taken[nw] = 1;
        build(nw, L+1);
    }
}
void centroidDecomposition(int &Root)
{
    mem(taken, 0);
    Root = getCentroid(1);
    taken[Root] = 1;
    build(Root, 1);
}

int getLCA(int u, int v)
{
    if(myLevel[u] < myLevel[v])
        swap(u,v);
    while(myLevel[u] != myLevel[v])
        u = par[u];
    while(u != v)
        u = par[u], v = par[v];
    return u;
}

int queryDist(int u, int v)
{
    int lca = getLCA(u,v);
    return dist[myLevel[lca]][u] + dist[myLevel[lca]][v];
}
```

Circle Rectangle Intersection Area:

```
//Circle Rectangle Intersection
// x1,y1 is the left bottom of the rectangle
// x2,y2 is the right top

double areaArc( double r, double x1, double y1 )
{
    double x2 = sqrt( r*r - y1*y1 );
    double y2 = sqrt( r*r - x1*x1 );
    double theta = acos( ( 2*r*r - (x2-x1)*(x2-x1) - (y2-y1)*(y2-y1) ) / (
2*r*r ) );
    return (theta*r*r - y1 * (x2 - x1) - x1 * (y2 - y1)) / 2;
}

double circleRectangleIntersection( int r, int x1, int y1, int x2, int y2 )
{
    if( x1 < 0 && x2 > 0 ) return circleRectangleIntersection( r, 0, y1, x2,
y2 ) + circleRectangleIntersection( r, x1, y1, 0, y2 );
    if( y1 < 0 && y2 > 0 ) return circleRectangleIntersection( r, x1, 0, x2,
y2 ) + circleRectangleIntersection( r, x1, y1, x2, 0 );
    if( x1 < 0 ) return circleRectangleIntersection( r, -x2, y1, -x1, y2 );
    if( y1 < 0 ) return circleRectangleIntersection( r, x1, -y2, x2, -y1 );
    if( x1 >= r || y1 >= r ) return 0.0;
    if( x2 > r ) return circleRectangleIntersection( r, x1, y1, r, y2 );
    if( y2 > r ) return circleRectangleIntersection( r, x1, y1, x2, r );
    if( x1*x1 + y1*y1 >= r*r ) return 0.0;
    if( x2*x2 + y2*y2 <= r*r ) return (x2 - x1) * (y2 - y1);
    int outCode = ( x2*x2 + y1*y1 >= r*r ) + 2 * ( x1*x1 + y2*y2 >= r*r );

    if( outCode == 3 ) return areaArc( r, x1, y1 );
    else if( outCode == 1 )
    {
        double x = sqrt( r*r - y2*y2 + 0.0 );
        return (x - x1) * (y2 - y1) + areaArc( r, x, y1 );
    }
    else if( outCode == 2 )
    {
        double y = sqrt( r*r - x2*x2 + 0.0 );
        return (x2 - x1) * (y - y1) + areaArc( r, x1, y );
    }
    else
    {
        double x = sqrt( r*r - y2*y2 + 0.0 );
        double y = sqrt( r*r - x2*x2 + 0.0 );
        return (x2 - x1) * (y - y1) + (x - x1) * (y2 - y) + areaArc( r, x, y
);
    }
}

const double pi = 2 * acos(0.0);

int cases, caseno;

struct circle
{
    int x, y, r;
};

double circleRectangleIntersection( circle C, int x1, int y1, int x2, int y2
)
{
    return circleRectangleIntersection( C.r, x1 - C.x, y1 - C.y, x2 - C.x, y2
- C.y );
}
```

Closest Pair:

```
struct point{
    double x, y;
};
point P[MAX+10], temp[MAX+10], S[MAX+10];
const double inf = ?;
double dist(point a, point b)
{
    return sqrt((a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y));
}
bool operator < (point a, point b)
{
    if(a.x == b.x)
        return a.y < b.y;
    return a.x < b.x;
}
double closestPair(int st, int ed)
{
    if(st == ed)
        return inf;

    int mid = (st+ed)/2;
    point midPoint = P[mid];

    double d = min(closestPair(st,mid), closestPair(mid+1, ed));
    int L = st, R = mid+1, cnt = 0;
    for(int id = 0; id<(ed-st)+1; id++ )
    {
        if(L > mid)
            temp[id] = P[R++];
        else if(R > ed)
            temp[id] = P[L++];
        else
        {
            if(P[L].y < P[R].y)
                temp[id] = P[L++];
            else
                temp[id] = P[R++];
        }
        if(fabs(midPoint.x - temp[id].x) < d)
            S[cnt++] = temp[id];
    }
    for(int id = 0, i = st; id<(ed-st)+1; id++, i++)
        P[i] = temp[id];
    for(int i = 0; i<cnt; i++)
    {
        for(int j = i+1; j<cnt; j++)
        {
            if(fabs(S[i].y - S[j].y) > d)
                break;
            d = min(d, dist(S[i], S[j]));
        }
    }
    return d;
}
```

CRT With Lucas:

```
1<= M <= 10^9
1 <= r <= n <= 10^9
M is squarefree and its prime factors are less than 50.

/*
  CRT:
      X % m1 = a_1
      X % m2 = a_2
      X % m3 = a_3
      m_1,m_2,m_3 are pair wise co-prime
      M = m_1*m_2*m_3
      u_i = Modular inverse of (M/m_i) with respect m_i
      X = ( a_1 * (M/m_1) * u_1 + a_2 * (M/m_2) * u_2 + a_3 * (M/m_3) * u_3 )
% M

  Lucas:
    If we need to find nCr % P where P is a prime but P can be
    less than n or r, we can use Lucas Theorem.
    nCr = ((n_0 C r_0) * (n_1 C r_1) * (n_2 C r_2) * ... * (n_k C r_k)) % P
    Where n_i is the i'th digit in P based representation of n
    and r_i is the i'th digit in P based representation of r
```

```
*/
vector<int>prime;
typedef long long int LL;
bool isPrime(int a)
{
    for(int i = 2; i<a; i++)
    {
        if(a%i == 0)
            return false;
    }
    return true;
}

LL exp(int a, int n, int MOD)
{
    if(n == 0)
        return 1;
    if(n&1)
        return (a*exp(a,n-1,MOD))%MOD;
    LL ret = exp(a,n/2,MOD);
    return (ret * ret)%MOD;
}

int lucas(int n, int r, int p)
{
    LL fact[55];
    fact[0] = 1;
    for(int i = 1; i<p; i++)
        fact[i] = (fact[i-1]*i)%p;
    vector<int>digN, digR;
    while(n)
    {
        digN.push_back(n%p);
        n/=p;
    }
    while(r)
    {
        digR.push_back(r%p);
        r/=p;
    }
    LL ret = 1;
    for(int i = 0; i<digR.size(); i++)
    {
        LL tmp;
        if(digN[i] < digR[i])
            tmp = 0;
        else
        {
            tmp = (fact[digN[i]-digR[i]] * fact[digR[i]])%p;
            tmp = (fact[digN[i]]*exp(tmp,p-2,p))%p;
        }
        ret = (ret * tmp)%p;
    }
    return ret;
}

int CRT(vector<int>&a, vector<int>&prime, int M)
{
    vector<int>u;
    for(int i = 0; i<prime.size(); i++)
        u.push_back(exp(M/prime[i], prime[i]-2, prime[i]));
    LL ret = 0;
    for(int i =0; i<prime.size(); i++)
    {
        LL tmp = (a[i] * (M/prime[i]))%M;
        tmp = (tmp * u[i])%M;
        ret = (ret + tmp);
        if(ret >= M)
            ret -= M;
    }
    return ret;
}

int nCr(int n, int r, int MOD)
{
    if(r > n)
        return 0;
    prime.clear();
    for(int i = 2; i<=50; i++)
    {
        if(MOD%i == 0 && isPrime(i))
            prime.push_back(i);
    }
    vector<int>a;
    for(int i = 0; i<prime.size(); i++)
        a.push_back(lucas(n,r,prime[i]));
    return CRT(a, prime, MOD);
}

int main()
{
    int t, n, r, MOD;
    scanf("%d",&t);
    for(int cs = 1; cs<=t; cs++)
    {
        scanf("%d %d %d",&n,&r,&MOD);
        printf("%d\n",nCr(n,r,MOD));
    }
}
```

Derangement:

```
int derangement(int n)
{
    if(!n) return n;
    if(n <= 2) return n-1;
    return (n-1)*(derangement(n-1) + derangement(n-2));
}
```

Diameter of Convex Polygon:

```
double find_diameter(vector<point>&A)
{
    sort(all(A));
    un(A);
    if(A.size() == 1)
        return 0;
    else if(A.size() == 2)
        return dist_point_point(A[0], A[1]);

    vector<point> hull;
    convex_hull(A, hull);
    int n = hull.size();

    int idx = 1;
    line nw = line(hull.back(),hull[0]);
    double mx = -inf;
    int i = 1;
    while(true)
    {
        double dist = dist_point_line(hull[i], nw);
        if(dist > mx)
            mx = dist, idx = i, i++;
        else
            break;
    }

    double ans = 0;
    for(i = 0; i<hull.size(); i++)
    {
        if(i == n-1)
            nw = line(hull[i], hull[0]);
        else
            nw = line(hull[i], hull[i+1]);

        double mx = -inf;
        for(int j = idx; ;j++)
        {
            if(j == n) j = 0;
            double dist = dist_point_line(hull[j], nw);
            if(dist > mx)
                mx = dist, idx = j, ans = max(ans, dist_point_point(hull[i],
hull[j]));
            else
                break;
        }
    }
    return ans;
}
```

Dijkstra:

```
/// O((V+E)log2(V))
/*
    Min Priority queue
    priority_queue<int, vector<int>, greater <int> > q;
```

```
*/

/*
    In dijkstra, always remember this optimization.
    if(dist[pq.front().xx] < pq.front().yy)
        continue;
    You don't need to re-check with a worse distance for the same node
*/
```

Dinic Maxflow Mine:

```
/// works for both 0 based and 1 based indexing
/// O(min(F,V^2)E)
/// O(sqrt(E)*E) for unit capacity graphs
/// O(sqrt(V)*E) for BPM
/// Edges are directed in this code. To make edges undirected, change the add
function
const int MAXN = ?;
int src, snk, nxt[MAXN+5], inf = ?, dist[MAXN+5];
struct edge
{
    int v, cap, opposite, flow;
};

vector<edge>E[MAXN+5];

void init(int _src, int _snk, int _n)
{
    src = _src, snk = _snk;
    for(int i = 0; i<=_n; i++)
        E[i].clear();
}

void add(int u, int v, int cap)
{
    E[u].push_back({v,cap,E[v].size(),0});
    E[v].push_back({u,0,E[u].size()-1,0});
}

bool bfs()
{
    memset(dist,-1,sizeof(dist));
    queue<int>q;
    dist[src] = 0;
    q.push(src);
    while(!q.empty())
    {
        int u = q.front();
        if(u == snk)
            return true;
        q.pop();
        for(int i = 0; i<E[u].size(); i++)
        {
            if(E[u][i].cap > E[u][i].flow)
            {
                int v = E[u][i].v;
                if(dist[v] == -1)
                {
                    dist[v] = dist[u] +1;
                    q.push(v);
                }
            }
        }
    }
    return dist[snk] != -1;
}

int dfs(int u, int sentFlow)
{
    if(u == snk)
        return sentFlow;
    for(; nxt[u]<E[u].size(); nxt[u]++)
    {
        int v = E[u][nxt[u]].v;
        int c = E[u][nxt[u]].cap;
        int f = E[u][nxt[u]].flow;
        int opposite = E[u][nxt[u]].opposite;
        if(dist[v] == dist[u]+1 && c > f)
        {
            int tmp = dfs(v,min(sentFlow,c-f));
            if(tmp)
            {
                E[u][nxt[u]].flow += tmp;
                E[v][opposite].flow -= tmp;
                return tmp;
            }
        }
    }
    return 0;
}

int dinitz()
{
    int totalFlow = 0;
    while(bfs())
    {
        memset(nxt,0,sizeof(nxt));
        int sentFlow;
        while(true)
        {
            sentFlow = dfs(src,inf);
            if(sentFlow == 0)
                break;
            totalFlow += sentFlow;
        }
    }
    return totalFlow;
}

int main()
{
    init(1, 4, 4);
    add(1, 2, 10);
    add(1, 3, 6);
    add(2, 3, 2);
    add(2, 4, 7);
    add(3, 4, 100);
    D(dinitz()); /// should be 15
}
```

Directed MST:

```
/*
*     Jaan Vai's code
*     Finds cost of forming DMST
*     Runs under V^2log(V) where V is the number of nodes
*     0 based indexing
*     MM is the number of nodes
*     Put all the outgoing edges from u in E[u]
*     Just call Find_DMST(root, number of nodes) and it will return the total
cost of forming DMST
*     if it returns inf, then initial graph was disconnected
*/
const int MM = ?
const int inf = ?

struct edge {
    int v, w;
    edge() {}
    edge( int vv, int ww ) { v = vv, w = ww; }
    bool operator < ( const edge &b ) const { return w < b.w; }
};

vector <edge> E[MM], inc[MM];
int DirectedMST( int n, int root, vector <edge> inc[MM] ) {
    int pr[MM];
    inc[root].clear();

    /// if any node is not reachable from root, then no mst can be found
    for( int i = 0; i < n; i++ ) {
        sort( inc[i].begin(), inc[i].end() );
        pr[i] = i;
    }
    bool cycle = true;
    while( cycle ) {
        cycle = false;
        int vis[MM] = {0}, W[MM];
        vis[root] = -1;
        for( int i = 0, t = 1; i < n; i++, t++ ) {
            int u = pr[i], v;
            if( vis[u] ) continue;
            for( v = u; !vis[v]; v = pr[inc[v][0].v] ) vis[v] = t;
            if( vis[v] != t ) continue;
            cycle = true;
            int sum = 0, super = v;
            for( ; vis[v] == t; v = pr[inc[v][0].v] ) {
                vis[v]++;
                sum += inc[v][0].w;
            }
            for( int j = 0; j < n; j++ ) W[j] = INT_MAX;
            for( ; vis[v] == t + 1; v = pr[inc[v][0].v] ) {
                vis[v]--;
                for( int j = 1; j < inc[v].size(); j++ ) {
                    int w = inc[v][j].w + sum - inc[v][0].w;
                    W[ inc[v][j].v ] = min( W[ inc[v][j].v ], w );
                }
                pr[v] = super;
            }
            inc[super].clear();
            for( int j = 0; j < n; j++ ) if( pr[j] != pr[ pr[j] ] ) pr[j] =
pr[ pr[j] ];
            for( int j = 0; j < n; j++ ) if( W[j] < INT_MAX && pr[j] != super
) inc[super].push_back( edge( j, W[j] ) );
            sort( inc[super].begin(), inc[super].end() );
        }
    }
    int sum = 0;
    for( int i = 0; i < n; i++ ) if( i != root && pr[i] == i ) sum +=
inc[i][0].w;
    return sum;
}

int Find_DMST(int root, int n) {
    bool visited[MM] = {0};
    queue <int> Q;
    for( int i = 0; i < n; i++ ) inc[i].clear();
    for( int i = 0; i < n; i++ ) for( int j = 0; j < E[i].size(); j++ ) {
        int v = E[i][j].v, w = E[i][j].w;
        inc[v].push_back( edge( i, w ) );
    }
    visited[root] = true;
    Q.push(root);
    while( !Q.empty() ) {
        int u = Q.front(); Q.pop();
        for( int i = 0; i < E[u].size(); i++ ) {
            int v = E[u][i].v;
            if( !visited[v] ) {
                visited[v] = true;
                Q.push(v);
            }
        }
    }
    /// The given graph is disconnected. So forming any MST is not possible.
    for( int i = 0; i < n; i++ ) if( !visited[i] ) return inf;
    return DirectedMST( n, root, inc );
}
```

Dominator Tree:

```
//1-Based directed graph input
#define MAX       200000
vector<int> g[MAX+5],tree[MAX+5],rg[MAX+5],bucket[MAX+5];
int sdom[MAX+5],par[MAX+5],dom[MAX+5],dsu[MAX+5],label[MAX+5];
int arr[MAX+5],rev[MAX+5],T,n, source;
void init(int _n, int _source)
{
    T = 0;
    n = _n;
    source = _source;
    for(int i = 1; i<=n; i++)
    {
        g[i].clear(), rg[i].clear(), tree[i].clear(), bucket[i].clear();
        arr[i] = sdom[i] = par[i] = dom[i] = dsu[i] = label[i] = rev[i] = 0;
    }
}
void dfs(int u)
{
    T++;
    arr[u]=T;
    rev[T]=u;
    label[T]=T;
    sdom[T]=T;
    dsu[T]=T;
    for(int i=0; i<g[u].size(); i++)
    {
        int w = g[u][i];
        if(!arr[w])
        {
            dfs(w);
            par[arr[w]]=arr[u];
        }
        rg[arr[w]].push_back(arr[u]);
    }
}
```

```
int Find(int u,int x = 0)
{
    if(u==dsu[u])return x?-1:u;
    int v = Find(dsu[u],x+1);
    if(v<0)return u;
    if(sdom[label[dsu[u]]]<sdom[label[u]])
        label[u] = label[dsu[u]];
    dsu[u] = v;
    return x?v:label[u];
}
void Union(int u,int v)  //Add an edge u-->v
{
    dsu[v]=u;
}

void build()
{
    dfs(source);
    for(int i=n; i>=1; i--)
    {
        for(int j=0; j<rg[i].size(); j++)
            sdom[i] = min(sdom[i],sdom[Find(rg[i][j])]);
        if(i>1)bucket[sdom[i]].push_back(i);
        for(int j=0; j<bucket[i].size(); j++)
        {
            int w = bucket[i][j],v = Find(w);
            if(sdom[v]==sdom[w]) dom[w]=sdom[w];
            else dom[w] = v;
        }
        if(i>1)Union(par[i],i);
    }

    for(int i=2; i<=n; i++)
    {
        if(dom[i]!=sdom[i])dom[i]=dom[dom[i]];
//        tree[rev[i]].push_back(rev[dom[i]]);
        tree[rev[dom[i]]].push_back(rev[i]);
    }
}
```

DSU On Tree:

```
vector<int>E[MAX+10];
int subtreeSize[MAX+10];

int getSize(int node, int par);

void add(int node, int par, int x, int bigChild = -1)
{
    /// Do whatever you have to do for this node with the command x
    for(auto v: E[node])
    {
        if(v == par || v == bigChild)
            continue;
        add(v, node, x);
    }
}

void dfs(int node, int par, bool keep)
{
    int bigChild = -1;
    for(auto v: E[node])
    {
        if(v == par)
            continue;
        if(bigChild == -1 || (subtreeSize[bigChild] < subtreeSize[v]))
            bigChild = v;
    }
    for(auto v: E[node])
    {
        if(v == par || v == bigChild)
            continue;
        dfs(v, node, 0);
    }
    if(bigChild != -1)
        dfs(bigChild, node, 1);
    add(node, par, 1, bigChild);
    /// my needed array is ready. Handle query next

    /// Clear solution as I'm not a bigchild. :(
    if(keep == 0)
    {
        add(node, par, -1);
    }
}
```

DP Optimization (D&C):

```
void pre()
{
    // set the cost function
    // Then set the base case
}

void call(int group, int L, int R, int optL, int optR)
{
    if(L > R)
        return;
    int mid = (L+R)/2;
    int ret = INT_MAX, idx;
    int lim = min(optR, mid);
    for(int i = optL; i<=lim; i++)
    {
        int cur = dp[group-1][i-1] + cost[i][mid];
        if(cur <= ret)
        {
            ret = cur;
            idx = i;
        }
    }
    dp[group][mid] = ret;
    call(group, L, mid-1, optL, idx);
    call(group, mid+1, R, idx, optR);
}

int solve(int n, int k)
{
    pre();
    for(int group = 1; group <= k; group++)
        call(group, 1, n, 1, n);
    return dp[k][n];
}
```

DP Optimization (Knuth) :

```
/// UVA optimal Binary Search Tree
int call(int st, int ed)
{
    if(st == ed)
    {
        path[st][ed] = ed;
        return freq[st];
    }
    if(dp[st][ed] != -1)
        return dp[st][ed];
    int ret = 1e9;

    call(st, ed-1);
    call(st+1, ed);

    int L = max(st, path[st][ed-1]);
    int R = min(ed, path[st+1][ed]);

    for(int i = L; i<=R; i++)
    {
        int nw = 0;
        if(i-1 >= st)
            nw += call(st, i-1);
        if(ed >= i+1)
            nw += call(i+1, ed);
        nw += csum[ed] - csum[st-1];
        if(nw < ret)
        {
            ret = nw;
            path[st][ed] = i;
        }
    }
    return dp[st][ed] = ret;
}
```

DP Optimization (Knuth Takes Over D&C):

```
// k is the total number of groups
LL call(int group, int pos)
```

```
{
    if(pos == 0)
        return dp[group][pos] = 0;
    if(group == 0)
        return dp[group][pos] = inf;

    if(dp[group][pos] != -1)
        return dp[group][pos];

    int L = 1, R = pos;
    if(pos-1 > 0)
    {
        call(group, pos-1);
        L = max(L, path[group][pos-1]);
    }
    if(group+1 <= k)
    {
        call(group+1, pos);
        R = min(R, path[group+1][pos]);
    }
    LL ret = inf;
    for(int i = L; i<=R; i++)
    {
        LL cur = call(group-1, i-1) + (cum[pos] - cum[i-1])*(pos-i+1);
        if(cur < ret)
        {
            ret = cur;
            path[group][pos] = i;
        }
    }
    return dp[group][pos] = ret;
}
```

Edmond Karp:

```
/// Edmond Karp Max Flow
/// O(V*E^2);
int E[MAX+10][MAX+10], src, dest;
queue<int> q;
bool vis[MAX*2+10];
int parent[MAX+10], n, m, B, S, D;
```

```
bool bfs()
{
    while(!q.empty())
        q.pop();
    mem(vis,0);
    vis[src] = 1;
    q.push(src);
    int i, nw, tmp;
    while(!q.empty())
    {
        nw = q.front();
        q.pop();
        for(i = 1; i<=n; i++)
        {
            if(!E[nw][i])
                continue;
            tmp = i;
            if(!vis[tmp])
            {
                vis[tmp] = 1;
                q.push(tmp);
                parent[tmp] = nw;
            }
            if(tmp == dest)
                return 1;
        }
    }
    return 0;
}
int go(int nd)
{
    if(nd == src)
        return inf;

    return min(E[parent[nd]][nd], go(parent[nd]));
}
void update(int nd, int mn)
{
    if(nd == src)
        return;

    E[parent[nd]][nd] -= mn;
    E[nd][parent[nd]] += mn;
    update(parent[nd],mn);
}
int Edmond_Karp()
{
    int ans = 0, mn;
    while(bfs())
    {
        mn = go(dest);
        ans += mn;
        update(dest,mn);
    }
    return ans;
}
```

Euclidean Thingies:

```
///TRIANGLES

# cos(A) = (b*b+c*c-a*a)/(2*(b*c))
# area of triangle = 0.5 * a*b* sin(C)
# a/sin(A)=b/sin(B)=c/sin(C) = 2*r /// where r is the radius of the
circumcircle of the triangle
# For two similar triangles ABC and DEF, area(ABC)/area(DEF) = (similar side
ratio)^2

# Triangle
      Circum Radius = a*b*c/(4*area)
      In Radius = area/s, where s = (a+b+c)/2
      length of median to side c = sqrt(2*(a*a+b*b)-c*c)/2
      length of bisector of angle C = sqrt(ab[(a+b)*(a+b)-c*c])/(a+b)

# Ellipse
      Area = PI*a*b

# Rhombus
      side length=s;
      Area = altitude × s
      Area = s*s sin(A)
      Area = s*s sin(B)

      p and q are diagonal lengths.
      Area = (p × q)/2

9. PICKS THEOREM :
      Given a simple polygon constructed on a grid of equal-distanced points
(i.e., points with integer coordinates)
   such that all the polygon's vertices are grid points, Pick's theorem
provides a simple formula for calculating the area A of this polygon in terms
of the number i of lattice points in the interior located in the polygon and
the number b of lattice points on the boundary placed on the polygon's
perimeter.

   A = i + b/2.0 - 1.
```

Euler Path:

```
/*
    Euler path = path that visits all the edges in a graph.
    An undirected graph has a Euler cycle iff it is connected and each vertex
has an even degree.

    An undirected graph has an Euler tour iff it is connected, and each
vertex, except for exactly two vertices, has an even degree.
    The two vertices of odd degree have to be the endpoints of the tour.

    A directed graph has a Euler cycle iff it is strongly connected and the
in-degree of each vertex is equal to its out-degree.

    A directed graph has an Euler tour if and only if at most one vertex has
(out-degree) – (in-degree) = 1, at most one vertex has
    (in-degree) – (out-degree) = 1, every other vertex has equal in-degree
and out-degree, and all of its vertices with nonzero degree belong
    to a single connected component of the underlying undirected graph.

    This code finds euler circuit.
    To find an Eulerian tour, simply find one of the nodes which has odd
degree and call find_circuit with it.
    Multiple edges between nodes can be handled by the exact same algorithm.
    Self-loops can be handled by the exact same algorithm as well, if
self-loops are considered to add 2 (one in and one out) to the degree of a
node.
    A directed graph has a Eulerian circuit if it is strongly connected
(except for nodes with both in-degree and out-degree of 0) and the indegree
```

```
of each node equals its outdegree. The algorithm is exactly the same, except
that because of the way this code finds the cycle, you must traverse arcs in
reverse order.
*/

///The following code finds euler path for directed graph
/// For others, just change the existence of euler path check accordingly.
/// To find an Eulerian tour in an undirected graph, simply find one of the
nodes which has odd degree and make it the source.
/// For others, anyone with non-zero indegree can be source.

#define MAX     ?
vector<int>E[MAX+10], sltn;
int indeg[MAX+10], outdeg[MAX+10];
int n ;
bool vis[MAX+10];
void dfs(int nd)
{
    /// this vis[nd] only makes sure in the end that nodes with nonzero
degree are all under
    /// the same weakly connected component. It has nothing to do with
finding the actual euler path.
    vis[nd] = true;
    while(E[nd].size())
    {
        int v = E[nd].back();
        E[nd].pop_back();
        dfs(v);
    }
    sltn.pb(nd);
}

/// Returns false if there is no euler path, returns true and prints the path
if there is an euler path
bool Find_Euler_Path()
{
    int src , sink;
    bool found_src = false, found_sink = false;
    mem(indeg,0);
    mem(outdeg,0);
    sltn.clear();
    /// calculate indegree and outdegree of every node
    for(int i = 1; i<=n; i++)
    {
        for(int j = 0; j<E[i].size(); j++)
        {
            int v = E[i][j];
            outdeg[i]++;
            indeg[v]++;
        }
    }
    /// check if euler path exists
    for(int i = 1; i<=n; i++)
    {
        int diff = outdeg[i] - indeg[i];
        if(diff == 1)
        {
            if(found_src)
                return false;
            found_src = true;
            src = i;
        }
        else if (diff == -1)
        {
            if(found_sink)
                return false;
            found_sink = true;
            sink = i;
        }
        else if(diff != 0)
            return false;
    }
    if(!found_src)
    {
        /// there actually exists a euler cycle. So you need to pick a random
node with non-zero degrees.
        for(int i = 1; i<=n;i++)
        {
            if(outdeg[i])
            {
                found_src = true;
                src = i;
                break;
            }
        }
    }
    if(!found_src)
        return true;
    mem(vis,0);
    dfs(src);
    for(int i = 1; i<=n; i++)
    {
        /// the underlying graph is not even weakly connected.
        if(outdeg[i] && !vis[i])
            return false;
    }
    for(int i = (int)sltn.size()-1; i>=0; i--)
        printf("%d ",sltn[i]);
    puts("");
    return true;
}
```

Euler Phi Formula:

```
phi(n) is the number of integers <= n who are coprime with n.
phi(n) = n*(1-(1/p1))*(1-(1/p2))*....*(1-(1/pk))
where n = p1^x1 * p2^x2 * p3^x2 * ... * pk^xk
```

Exponial Formula:

```
/*
if ( b >= phi[m] )
    (a ^ b)% m = (a ^ (b% phi [m] + phi [m]))% m
*/

/* Problem:
```

$$\text{exponial}(n) = n^{(n-1)^{(n-2)^{\cdots^{2^1}}}}.$$

```
compute exponial(n) % m.
1 <= n, m <= 1000000000
*/
/// key insight: phi(m) converges to 1 within very few steps.
const int MAX = sqrt(1e9)+1;
vector<int>prime;
```

```
bool notPrime[MAX+10];
void sieve()
{
    prime.pb(2);
    for(int i = 3; i<=MAX; i+=2)
    {
        if(notPrime[i] == false)
        {
            prime.pb(i);
            for(int j = i*i; j<=MAX; j+=(i+i))
                notPrime[j] = true;
        }
    }
}

LL phi(int a)
{
    LL ret = a;
    for(int i = 0; i<prime.size() && prime[i]*prime[i] <= a; i++)
    {
        if(a%prime[i] == 0)
        {
            ret *= (prime[i]-1);
            ret /= prime[i];
            while(a%prime[i] == 0)
                a/=prime[i];
        }
    }
    if(a > 1)
        ret *= (a-1), ret/= a;
    return ret;
}

LL exp(LL a, LL n, LL MOD)
{
    if(n == 0)
        return (1%MOD);
    if(n&1)
        return (a*exp(a,n-1,MOD))%MOD;
    LL ret = exp(a,n/2,MOD);
    return (ret * ret)%MOD;
}
```

```
/*
if ( b >= phi[m] )
    (a ^ b)% m = (a ^ (b% phi [m] + phi [m]))% m
*/


LL solve(LL n, LL m)
{
    if(n < 5)
    {
        LL ret = 1;
        for(int i = 2; i<=n; i++)
            ret = exp(i,ret,(LL)1e15);
        return ret%m;
    }
    if(m == 1)
        return 0;

    LL p = phi(m);
    LL b = solve(n-1, p) + p;
    return exp(n,b,m);
}

int main()
{
    sieve();
    LL n, m;
    sll(n, m);
    printf("%lld\n",solve(n,m));
    return 0;
}
```

Extended Euclid:

```
/*
    Find the number of integer solutions to the problem:
        Ax + By = C
    x1<=x<=x2 and y1<=y<=y2
    Note:
    1.  Number of points with integer co-ordinates between (x,y) and (_x,_y)
is:
        gcd(dx,dy)+1 ( Including both the terminal points )
    2.  Next point of(_x, _y) with integer co-ordinates on the straight line
```

```
Ax+By=C
        (_x+b/g, _y-a/g).
*/

struct triple{
    LL x, y, g;
};

triple egcd(LL a, LL b)
{
    if(b == 0)
        return {1, 0, a};
    triple tmp = egcd(b, a%b);
    triple ret;
    ret.x = tmp.y;
    ret.y = (tmp.x - (a/b)*tmp.y);
    ret.g = tmp.g;
    return ret;
}

LL myCeil(LL a, LL b)
{
    LL ret = (a/b);
    if((a >= 0 && b>= 0) || (a < 0 && b < 0))
    {
        if(a%b)
            ret++;
    }
    return ret;
}

LL myFloor(LL a, LL b)
{
    LL ret = (a/b);
    if((a > 0 && b < 0) || (a < 0 && b > 0))
    {
        if(a%b)
            ret--;
    }
    return ret;
}

/// x1 <= _x + t*(b/g) <= x2
```

```
pll get_range(LL x1, LL x2, LL _x, LL b, LL g)
{
    if((b/g) < 0)
        swap(x1, x2);

    LL low = myCeil(((x1 - _x) * g), b);
    LL high = myFloor(((x2 - _x) * g), b);

    return {low, high};
}

/// ax + by = c
LL solve(LL a, LL b, LL c, LL x1, LL x2, LL y1, LL y2)
{
    triple sltn = egcd(a,b);
    LL ret;

    if(a == 0 && b == 0)
    {
        if(c == 0) ret = (x2 - x1 + 1) * (y2 - y1 + 1);
        else ret = 0;
    }
    else if(a == 0)
    {
        if(c%b) ret = 0;
        else
        {
            LL y = (c/b);
            if(y1 <= y && y <= y2)
                ret = (x2 - x1 + 1);
            else
                ret = 0;
        }
    }
    else if(b == 0)
    {
        if(c%a) ret = 0;
        else
        {
            LL x = (c/a);
            if(x1 <= x && x <= x2)
                ret = (y2 - y1 + 1);
            else
```

```
                ret = 0;
            }
        }
        else
        {
            if(c%sltn.g) ret = 0;
            else
            {
                sltn.x *= (c/sltn.g);
                sltn.y *= (c/sltn.g);
                pll rangeX = get_range(x1,x2,sltn.x,b,sltn.g);
                pll rangeY = get_range(y1,y2,sltn.y,-a,sltn.g);
                pll range;
                range = {max(rangeX.xx, rangeY.xx), min(rangeX.yy, rangeY.yy)};
                ret = range.yy - range.xx + 1;
                ret = max(0LL, ret);
            }
        }
        return ret;
    }
```

Fast I/O (Linux):

```
/// works for negative numbers
/// fastRead_string was never tested by me.
/// doesn't work on OS other than Linux
inline void Read(int &x)
{
    register int c = getchar_unlocked();
    x = 0;
    int neg = 0;
    for(; ((c<48 || c>57) && c != '-'); c = getchar_unlocked());

    if(c=='-'){
        neg = 1;
        c = getchar_unlocked();
    }

    for(; c>47 && c<58 ; c = getchar_unlocked()){
        x = (x<<1) + (x<<3) + c - 48;
    }
    if(neg)
        x = -x;
}

inline void Read_String(char *str)
{

    register char c = 0;
    register int i = 0;

    while (c < 33)
        c = getchar_unlocked();

    while (c != '\n')
    {
        str[i] = c;
        c = getchar_unlocked();
        i = i + 1;
    }
    str[i] = '\0';
}

inline void print(int a)
{
    if(a < 0)
    {
        putchar_unlocked('-');
        a = -a;
    }
    char s[11];
    int t = -1;
    do
    {
        s[++t] = a % 10 + '0';
        a /= 10;
    }
    while(a > 0);
    while(t >= 0)putchar_unlocked(s[t--]);
}
```

Fast I/O any OS:

```
/// works for negative numbers
/// fastRead_string was never tested by me.
/// Remember to fix data type

inline void Read(int &a)
{
    int cc = getc(stdin);
    for (; (cc < '0' || cc > '9') && (cc!='-');)  cc = getc(stdin);
    bool neg = false;
    if(cc == '-')
    {
        neg = true;
        cc = getc(stdin);
    }
    int ret = 0;
    for (; cc >= '0' && cc <= '9';)
    {
        ret = ret * 10 + cc - '0';
        cc = getc(stdin);
    }
    if(neg)
        ret = -ret;
    a = ret;
}

inline void Read_string(char *str)
{

    register char c = 0;
    register int i = 0;

    while (c < 33)
        c = getc(stdin);

    while (c != '\n' && c != ' ')
    {
        str[i] = c;
        c = getc(stdin);
        i = i + 1;
    }
    str[i] = '\0';
}

inline void print(int a)
{

    if(a < 0)
    {
        putc('-',stdout);
        a = -a;
    }
    char s[11];
    int t = -1;
    do
    {
        s[++t] = a % 10 + '0';
        a /= 10;
    }
    while(a > 0);
    while(t >= 0)putc(s[t--],stdout);
}
```

FFT:

```
/*
    multiply (7x^2 + 8x^1 + 9x^0) with (6x^1 + 5x^0)
    ans = (42x^3 + 83x^2 + 94x^1 + 45x^0)
    A = (9,8,7), B = (5,6)
    multiply(A,B,res)
    res will be (45 94 83 42)
*/
typedef complex<double> base;
void fft (vector<base> & a, bool invert) {
      int n = (int) a.size();

      for (int i=1, j=0; i<n; ++i) {
            int bit = n >> 1;
            for (; j>=bit; bit>>=1)
                  j -= bit;
            j += bit;
            if (i < j)
                  swap (a[i], a[j]);
      }

      for (int len=2; len<=n; len<<=1) {
            double ang = 2*PI/len * (invert ? -1 : 1);
            base wlen (cos(ang), sin(ang));
            for (int i=0; i<n; i+=len) {
                  base w (1);
                  for (int j=0; j<len/2; ++j) {
                        base u = a[i+j],  v = a[i+j+len/2] * w;
                        a[i+j] = u + v;
                        a[i+j+len/2] = u - v;
                        w *= wlen;
                  }
            }
      }
      if (invert)
            for (int i=0; i<n; ++i)
                  a[i] /= n;
}
void multiply (const vector<int> & a, const vector<int> & b, vector<int>
&res) {
      vector<base> fa (a.begin(), a.end()),  fb (b.begin(), b.end());
      size_t n = 1;
      while (n < max (a.size(), b.size()))  n <<= 1;
      n <<= 1;
      fa.resize (n),  fb.resize (n);

      fft (fa, false),  fft (fb, false);
      for (size_t i=0; i<n; ++i)
            fa[i] *= fb[i];
      fft (fa, true);
      res.resize (n);
      for (size_t i=0; i<n; ++i)
            res[i] = int (fa[i].real() + 0.5);
    /// Uncomment the following to Speed up.
    /// But don't do that for hamming distance or where leading zeros of a
polynomial is important
//    while(res.size() && res.back() == 0)
//        res.pop_back();
//    if(res.empty())
//        res.push_back(0);
```

```
}

int main()
{
    vector<int>A = {9, 8, 7}, B = {5, 6}, res;
    multiply(A, B, res);
    for(auto v: res)
        cout << v << " ";
    cout << endl;
    return 0;
}
```

Fusion Principle:

Given is a connected graph (the bush) on **N** vertices. The vertices are numbered from **1** to **N.** Edges represent branches of the bush, vertex **1** represents the ground. Players alternate in making moves, Alice moves first. A move consists of two steps. In the first step the player selects an edge and removes it from the graph. In the second step he/she removes all the edges that are no longer connected to the ground. (The player cuts a branch of the bush and removes the part of the bush he cut away.) The first player who has no legal move (nothing to cut) loses. You may assume that both Alice and Bob play optimally.

Solution:

```
/// MAX should be equal to (V+E) for safety
#define MAX     ?
struct edgeData{
    vector<int> E[MAX+10];
}A[2];
int root;
vector<int>stck, tmp;
bool forbidden[MAX+10], vis[MAX+10];
int n;
void getCycle(int u)
{
    tmp.clear();
    for(int i = (int)stck.size()-1; i>=0; i--)
    {
        tmp.pb(stck[i]);
        if(stck[i] == u)
        {
            stck = tmp;
            return;
        }
    }
    assert(false);
}

bool dfs(int node, int par, vector<int>E[])
{
    vis[node] = true;
    stck.pb(node);
    int cnt = 0;
    for(auto v: E[node])
    {
        if(v == par)
            cnt++;
    }
    if(cnt > 1)
    {
        getCycle(par);
        return true;
    }
    for(auto v: E[node])
    {
        if(par != v)
        {
            if(vis[v])
            {
                getCycle(v);
                return true;
            }
            if(dfs(v, node, E))
                return true;
        }
    }
    stck.pop_back();
    return false;
}

int green(int node, int par, vector<int>E[])
{
    int x = 0;
    for(auto v: E[node])
    {
        if(v != par)
            x ^= (green(v, node, E) + 1);
    }
    return x;
}
string solve()
{
    bool kechal = true;
    root = 1;
    int nw = 0, nxt = 1;
    while(kechal)
    {
        mem(vis, 0);
        stck.clear();
        kechal = dfs(root, -1, A[nw].E);
        if(kechal)
        {
            n++;
            assert(n <= MAX);
            int current = n, cnt = 0;
            for(auto u: stck)
            {
                forbidden[u] = true;
                if(u == root)
                    root = n;
            }
            for(int i= 1; i<current; i++)
            {
                for(auto v: A[nw].E[i])
                {
                    if(forbidden[i] && forbidden[v])
                        cnt++;
                    else if(forbidden[i])
                        A[nxt].E[current].pb(v);
                    else if(forbidden[v])
                        A[nxt].E[i].pb(current);
                    else
                        A[nxt].E[i].pb(v);
                }
            }
            cnt /= 2;
            if(cnt & 1)
```

```
                A[nxt].E[current].pb(++n), A[nxt].E[n].pb(current);
            stck.clear();
            for(int i = 1; i<=n; i++)
            {
                A[nw].E[i].clear();
            }
            swap(nw, nxt);
        }
    }
    return green(root, -1, A[nw].E) ? "Alice" : "Bob";
}

int cnt[MAX+10];
int main()
{
    int t, u, v, m;
    sf(t);
    for(int cs = 1; cs<=t; cs++)
    {
        sff(n,m);
        mem(cnt, 0);
        for(int i = 1; i<=m; i++)
        {
            sff(u,v);
            if(u == v)
                cnt[u]++;
            else
            {
                A[0].E[u].pb(v);
                A[0].E[v].pb(u);
            }
        }
        int cur = n;
        for(int i = 1; i<=cur; i++)
        {
            if(cnt[i] > 1)
            {
                if(cnt[i] & 1)
                    A[0].E[i].pb(++n), A[0].E[n].pb(i);
            }
        }
        cout << solve() << '\n';
        for(int i = 1; i<=n; i++)
```

```
            A[0].E[i].clear(), A[1].E[i].clear();
        mem(forbidden, 0);
    }
    return 0;
}
```

Gambler's Ruin:

```
/// you are at co-ordinate x on a line with integer co-ordinates only
/// In one move, you have to go to (x+1) or (x-1). You cannot stay at x.
/// p is the probability of moving to (x+1) from x
/// (1-p) is the probability of moving to (x-1) from x
/// if you reach a or b you cannot move anymore.
/// MUST: a <= st <= b, a != b,  0 < p < 1
/// randomWalk returns the probability of ending at b if you start from st.

/// Gambler's Ruin problem can be transformed to the above problem.
/// Two gamblers are playing and the first player has u1 and the 2nd player
has u2 units of money. In a round the probability of first player winning is
p.
/// The loser has to give 1 unit to the winner after a round. When a player
has 0 units, the game is over.
/// Find the probability of the first player winning.
/// a = 0, b = u1+u2, st = u1. Isn't it obvious then? :P


long double randomWalk(long double p, int a, int b, int st)
{
    if(isEq(p,0.5))
        return (st-a)/(long double)(b-a);
    assert(0 < p && p < 1);
    assert(a <= st && st <= b);
    if(st == a)
        return 0;
    if(st == b)
        return 1;

    long double q = 1-p;
    long double c = q/p;
    long double up, down, ret;
    if(c > 1)
    {
        up = pow(c, st-b)-1;
        down = pow(c, a-b)-1;
        ret = 1 - (up/down);
    }
    else
    {
        up = pow(c, st-a)-1;
        down = pow(c, b-a) - 1;
        ret = (up/down);
    }
    return ret;
}

long double randomWalkExpLen(long double p, int a, int b, int st)
{
    if(isEq(p,0.5))
        return ((long double)(st-a)) * (b-st);
    long double win = randomWalk(p, a, b, st);
    long double ret = (win*(b-st) + (1-win) * (a-h))/(p-q);
    return ret;
}
```

Gaussian Elimination:

```
#define MAX         ?
/// 0 based indexing
/// n rows, m columns
/// n equations, m variables
/// MAX is the maximum number of equations or variables
/// X holds the solution
/// returns 1 if unique solution, -1 if inconsistent, inf if infinite
solutions
struct matrix{
    double arr[MAX+10][MAX+10];
    int n, m;
}A;

int where[MAX+10];
double X[MAX+10];
const int inf = INT_MAX;
int gauss(int totalRow, int totalCol)
```

```
{
    A.n = totalRow, A.m = totalCol;
    mem(where, -1);
    int row, col;
    for(row = col = 0; row<A.n && col<A.m; col++)
    {
        int pivot = row;
        for(int i = row+1; i<A.n; i++)
        {
            if(fabs(A.arr[pivot][col]) < fabs(A.arr[i][col]))
                 pivot = i;
        }

        if(fabs(A.arr[pivot][col]) < eps)
            continue;
        if(pivot != row)
        {
            for(int i = 0; i<=A.m; i++)
                swap(A.arr[row][i], A.arr[pivot][i]);
        }
        where[col] = row;
        for(int i = row+1; i<A.n; i++)
        {
            if(fabs(A.arr[i][col]) > eps)
            {
                double c = -A.arr[i][col]/A.arr[row][col];
                for(int j = col+1; j<=A.m; j++)
                    A.arr[i][j] += c*A.arr[row][j];
            }
            A.arr[i][col] = 0;
        }
        row++;
    }


    for(int i = 0; i<A.n; i++)
    {
        bool nothing = true;
        for(int j = 0; j<A.m;j++)
        {
            if(fabs(A.arr[i][j]) > eps)
                nothing = false;
        }
        if(nothing && fabs(A.arr[i][A.m]) > eps)
            return -1;
    }

    for(int i = A.m-1; i>=0; i--)
    {
        if(where[i] == -1)  return inf;
        int row = where[i];
        double sltn = A.arr[row][A.m];
        for(int j = i+1; j<A.m; j++)
            sltn -= A.arr[row][j]*X[j];
        X[i] = sltn/A.arr[row][i];
    }

    return 1;
}
```

Hackenbush:

Almost Everything About Hackenbush

Green Hackenbush And Colon Principle:

Game Rules:

1. Players move in alternating turns, and both players always move optimally.
2. During each move, a player removes an edge from the tree, disconnecting one of its leaves or branches. The leaf or branch that was disconnected from the rooted tree is removed   from the game.
3. The first player to be unable to make a move loses the game.

Solution:

_ 1. A single stalk is like a chain.
2. Size of a stalk is the number of edges in it.
3. A single stalk can be considered as a nim pile.
4. All kind of trees can be transformed to a single stalk using colon principle.
5. Colon Principle:

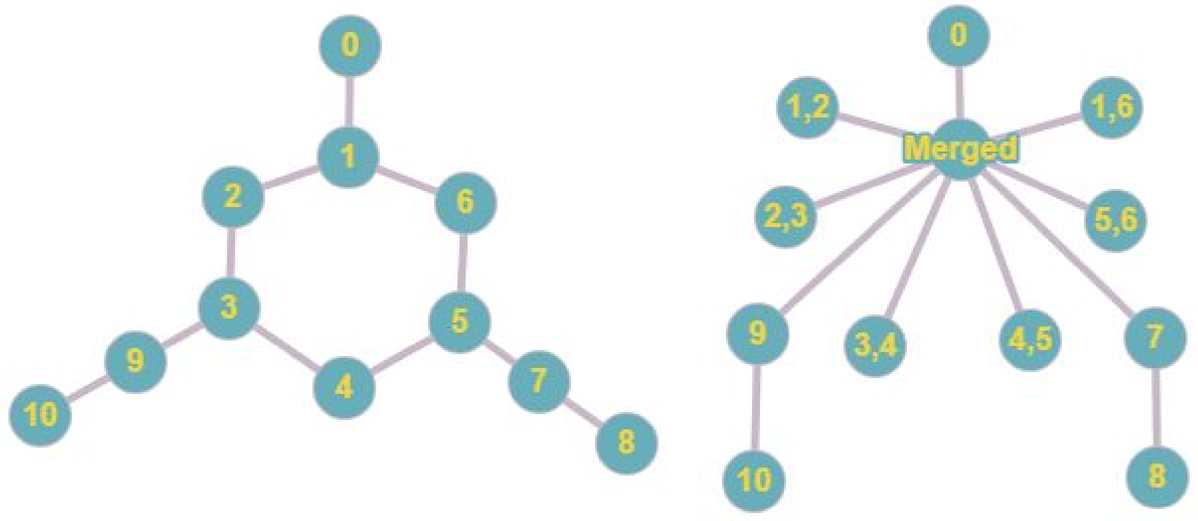
    If node u has 3 children a, b, c. then

            stalkSize(u) = (stalkSize(a) + 1) ^ (stalkSize(b) + 1) ^ (stalkSize(c) + 1)

6. Look at the nim pile size of the stalk obtained from the tree and decide who wins the game.

Fusion Principle:

The Fusion Principle states that you can fuse all the nodes in any cycle of a Green Hackenbush and get a tree structure.



Node 1,2,3,4,5,6 are in a cycle cycle, they are merged to a single node. For every edge in the cycle a new node arises. Thus, we can get a tree from any graph and apply colon principle there.

If number of edges in the cycle is even, then the cycle can be replaced by a single node. If
number of edges in the cycle is odd, then the cycle can be replaced by a single special node with an additional edge. In both cases all the other edges of the nodes that just got fused, get attached to the new node.

Blue Red Stalk:

1. Blue player moves first
2. Blue player only cuts blue edges, Red player only cuts red edges. Substalk or subtree which has no attachment to the ground/root gets eliminated just like green hackenbush.
3. First player who cannot move loses.

Solution:

Blue is positive, Red is negative.

Until the color changes for the first time, each edge is worth +V or -V (depending on whether it is blue or red, respectively). Once color change occurs, each subsequent edge (regardless of being blue or red), is worth half of the previous value, with a +/- corresponding to the color.

For example: BBBRB = +V +V +V -V/2 +V/4. Now sum them to get value of each stalk.

For multiple stalk, instead of using XOR, we add them normally. If positive then Blue wins and if negative Red wins.

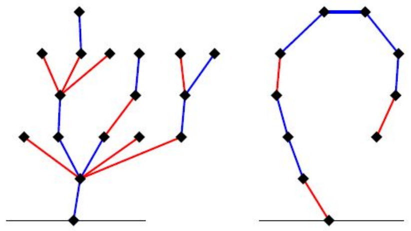
Blue Red Hackenbush Tree:

Same as blue red stalk, but with a tree.

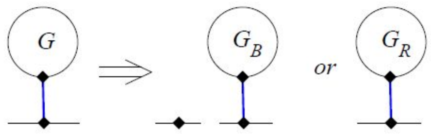
We can convert a tree to a stalk. Keep reading with patience.

11.1 Stalks and Trees

A "tree" is a connected position where there is a unique path from any node to the "ground" node. In other words, in a tree, there are no loops. A "stalk" is a very special kind of tree where there are no branches: every node has at most two segments connected to it. In both a tree and a stalk, nodes can be colored arbitrarily. In the figure below, the position on the left is a tree and the position on the right is a stalk (and also a tree, of course).



It turns out that there are some nice recursive methods to calculate the value of a tree or a stalk, and those recursive methods can sometimes be used to help calculate the values of more complex positions. The idea is based on the following theorem based on the figure below:



If G is some Red-Blue Hackenbush position having value x (for example, the position on the far left of the figure above), then if the grounded segments of G are connected instead to a single segment of color blue, then the value of "G on a stick" is equal to the first value from the series:

$$\frac{x+1}{1}, \frac{x+2}{2}, \frac{x+3}{4}, \frac{x+4}{8}, \frac{x+5}{16}, \frac{x+6}{32}, \ldots$$

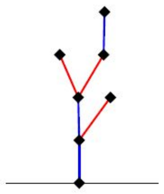
for which the numerator of the expression exceeds 1.

(If the "stick" under the game G with value x is instead red, then the value of that game on a stick is given by the first term in the series:

$$\frac{x-1}{1}, \frac{x-2}{2}, \frac{x-3}{4}, \frac{x-4}{8}, \frac{x-5}{16}, \frac{x-6}{32}, \ldots$$

for which the numerator of the expression is exceeded by -1.)

Let's now use this theorem to calculate the value of the position below:



The topmost single blue segment has value 1. The node below that has two red segments. One is a single segment with value -1, but the one on the right is blue on red (which should have value $-1/2$). To see that the $-1/2$ agrees with the

formulas, we need to look at the numerators: $x-1$, $x-2$, $\ldots$, where $x = 1$ (the value of the single blue strut) and find the first one exceeded by -1. We see that $1-1$ doesn't work, nor does $1-2$ (which is equal to -1, but is not exceeded it), but finally, $1-3 = -2$ is smaller than -1 therefore the stalk with blue on top of red is $\frac{1-3}{4} = -1/2$.

The next node down, which is one step above the ground, has a single red strut with value -1, plus a blue strut with a position equal to $-3/2 = -1 + (-1/2)$ on top of it. Now x will be $-3/2$ and we need to look at $x+1$, $x+2$, $\ldots$ until one is larger than 1. We see that $x+3$ works, so the value of the position is $(+3/2)/4 = +3/8$. That, plus the red segment with value -1 yields a game of value $-5/8$ on top of a blue stick. We have to add 2 to that to exceed 1, so the value of the full game will be: $((-5/8)+2)/2 = 11/16$.

Harmonic Sum:

```
/// Error is in the region of 2e-12 for values greater than 1000.
/// Run for loop for less than 1000.
const double Gamma = 0.5772156649;
double Hn(LL n)
{
     double r = 0;
     r = log(n) + Gamma + .5/n - (1/(12.0 * n*n));
     return r;
}
```

Hashing:

```
/*
    Call initHashing() before doing anything for god's sake.
    Change query return type to int or pii depending on number of hashes.
    Vector is really slow.
    String->abcde, base 10, a is 1
    hash array will be-> [1,12,123,1234,12345]
    hash value will be ->12345
    query(2,3)->34
    update will only work for hash value. Array will not be modified and will
be inconsistent.
    So you should not query for a subarray after updating a hash value.
    update(2,'e')-> new hash value is 12545
*/
#define N 2
#define MAX     100000
LL base[N], mod[N], power[N][MAX+10];
int totalHash;
void initHashing()
{
    totalHash = 2;
    base[0] = 3407;
    base[1] = 4721;
    mod[0] = 1000003999;
    mod[1] = 1000000861;
    for(int i = 0; i<totalHash; i++)
    {
        power[i][0] = 1;
        for(int j = 1; j<=MAX; j++)
            power[i][j] = (power[i][j-1] * base[i])%mod[i];
    }
}

struct HashData{
    LL ara[N][MAX+10], Hash[N];
    char str[MAX+10];
    int len;
```

```
void init(char *s)
{
    strcpy(str, s);
    len = strlen(str);
    for(int i = 0; i<totalHash; i++)
    {
        ara[i][0] = str[0];
        for(int j = 1; j<len; j++)
        {
            ara[i][j] = (ara[i][j-1]*base[i])%mod[i];
            ara[i][j] +=  str[j];
            if(ara[i][j] >= mod[i])
                ara[i][j] -= mod[i];
        }
        Hash[i] = ara[i][len-1];
    }
}

inline pii query(int st, int ed)
{
    int ret[2];
    for(int i = 0; i<totalHash; i++)
    {
        LL nw = ara[i][ed];
        if(st > 0)
        {
            nw -= (ara[i][st-1]*power[i][ed-st+1])%mod[i];
            if(nw < 0)
                nw += mod[i];
        }
        ret[i] = nw;
    }
    return {ret[0],ret[1]};
}

inline void append(char c)
{
    len++;
    for(int i = 0; i<totalHash; i++)
    {
        if(len > 1)
            ara[i][len-1] = (ara[i][len-2]*base[i])%mod[i];
        else
```

```
            ara[i][len-1] = 0;
        ara[i][len-1] += (c);
        if(ara[i][len-1] >= mod[i])
            ara[i][len-1] -= mod[i];
        Hash[i] = ara[i][len-1];
    }
    str[len-1] = c;
    str[len] = 0;
}

inline bool isEqual(const HashData &b)
{
    for(int i = 0; i<totalHash; i++)
    {
        if(Hash[i] != b.Hash[i])
            return false;
    }
    return true;
}

inline void update(int idx, char c)
{
    for(int i = 0; i<totalHash; i++)
    {
        Hash[i] -= (power[i][len-idx-1] * str[idx])%mod[i];
        if(Hash[i] < 0)
            Hash[i] += mod[i];
        Hash[i] += (power[i][len-idx-1] * c)%mod[i];
        if(Hash[i] >= mod[i])
            Hash[i] -= mod[i];
    }
    str[idx] = c;
}

void print()
{
    D(len);
    D(str);
    for(int i = 0; i<totalHash; i++)
    {
        puts("printing hash array");
        for(int j = 0; j<len; j++)
            cout << ara[i][j] << " ";
            cout << endl;
            D(Hash[i]);
        }
    }
}H;

int main()
{
    initHashing();
    H.init("abcde");
    DD(H.Hash[0], H.Hash[1]);
    pii a = H.query(1,3);
    DD(a.xx, a.yy);
    H.update(2,'z');
    DD(H.Hash[0], H.Hash[1]);
}
/*
    H.Hash[0] = 485411926   H.Hash[1] = 433979241
    a.xx = 137882996   a.yy = 184674175
    H.Hash[0] = 752387853   H.Hash[1] = 946599584
*/
```

HavelHakimi:

```
/// 1 based indexing
/// Given the degree of each of the node, check if a simple undirected graph
can be constructed.
/// A simple graph is an undirected graph without multiple edges or loops.
/// sltn contains the list of edges if graph construction is possible.
/// It can be the case that the graph built is not connected.
/// But this algorithm will always build a connected graph if possible.
/// For checking if a connected graph is built:
/// If graph can be built and sum of degrees of all vertices is more than or
equal to 2*n-2 and
/// each vertex degree is > 0, then a connected graph can be built with that
degree sequence.

#define MAX      ?
struct nodeData{
    int degree, id;
}A[MAX+10];

bool operator < (nodeData a, nodeData b)
{
    return a.degree < b.degree;
}

priority_queue<nodeData> PQ;

bool havelHakimi(int n, vector< pii >&sltn)
{
    while(!PQ.empty())
        PQ.pop();
    int mx = 0, mn = n+1;
    int oddCount = 0;
    for(int i = 1; i<=n; i++)
    {
        mx = max(mx, A[i].degree);
        mn = min(mn, A[i].degree);
        oddCount += (A[i].degree&1);
        if(A[i].degree > 0)
            PQ.push(A[i]);
    }
    if((oddCount&1) || (mx >= n) || (mn < 0))
        return false;
    while(!PQ.empty())
    {
        vector<nodeData>tmp;
        nodeData u = PQ.top();
        PQ.pop();
        while(u.degree > 0)
        {
            if(PQ.empty())
                return false;
            nodeData v = PQ.top();
            PQ.pop();
            sltn.pb({u.id,v.id});
            u.degree--;
            v.degree--;
            if(v.degree)
                tmp.pb(v);
        }
        for(int i = 0; i<tmp.size(); i++)
            PQ.push(tmp[i]);
    }
    return true;
}

int main()
{
    for(int i = 1; i<=4; i++)
        A[i].id = i, A[i].degree = 1;
    A[5].id = 5;
    A[5].degree = 2;
    vector<pii>res;
    assert(havelHakimi(5, res));
    for(auto s: res)
        DD(s.xx, s.yy);
}
```

Heavy Light Decomposition:

```
Problem:
A tree where each node's value is initially 0. Then two types of  operations.
1.  The value of node u is increased by v.
2.  Report the maximum value in the path from u to v.

Solution:
#define MAXLG 18
#define MAXN 100000
int parent[MAXN+10], sparseTable[MAXLG+2][MAXN+10], level[MAXN+10], n,
subtreeSize[MAXN+10];
vector<int> E[MAXN+10];
int tree[MAXN*4+10];

int dfs(int node, int par, int currentLevel)
{
    level[node] = currentLevel;
    parent[node] = par;
    int ret = 1;
    for(int i = 0; i<E[node].size(); i++)
    {
        int v = E[node][i];
        if(v != par)
            ret += dfs(v,node,currentLevel+1);
    }
    return subtreeSize[node] = ret;
}
void initSparseTable(int Root)
{
    dfs(Root,-1,1);
    for(int i = 1; i<=n; i++)
        sparseTable[0][i] = parent[i];
    for(int p = 1; p <= MAXLG; p++)
    {
        for(int u = 1; u <= n; u++)
            sparseTable[p][u] = sparseTable[p-1][sparseTable[p-1][u]];
    }
}

int findLCA(int u, int v)
{
    /// keep u as the deeper node
    if(level[v] > level[u])
        swap(u,v);
    for(int i = MAXLG; i>=0; i--)
    {
        if((1<<i) <= level[u] - level[v])
            u = sparseTable[i][u];
    }
    if(u == v)
        return v;
    for(int i = MAXLG; i>=0; i--)
    {
        if(sparseTable[i][u] != sparseTable[i][v])
            u = sparseTable[i][u], v = sparseTable[i][v];
    }
    return parent[v];
}


/// Each node in the tree has a value
/// Update a node's value
/// query for the summation of values of the nodes on the path from u to v
/// chainNo is the total number of chains
/// chainHead[u] is the head of node u's chain
/// chainSize[i] is the size of the chain i
/// chainId[u] is the id/number of u's chain
/// idxInBaseArray[u] is the index of the base array where data about u is
stored
/// baseArray is the array upon which BIT/Segment tree is created
/// It's expected that subtreeSize array is filled up while doing dfs for
initSparseTable
int chainNo = 0, chainHead[MAXN+10], chainSize[MAXN+10],
    chainId[MAXN+10], idxInBaseArray[MAXN+10], cnt = 0;

/// HLD start
void HLD(int node, int par)
{
    cnt++;
    idxInBaseArray[node] = cnt;
    if(chainHead[chainNo] == -1)
        chainHead[chainNo] = node;
    chainId[node] = chainNo;
    chainSize[chainNo]++;
    int specialChild = -1, maxSubtreeSize = 0;
    for(int i = 0; i<E[node].size(); i++)
    {
        int v = E[node][i];
        if(v != par && subtreeSize[v] > maxSubtreeSize)
        {
            maxSubtreeSize = subtreeSize[v];
            specialChild = v;
        }
    }
    if(specialChild != -1)
        HLD(specialChild, node);
    for(int i = 0; i<E[node].size(); i++)
    {
        int v = E[node][i];
        if(v != par && v != specialChild)
        {
            chainNo++;
            HLD(v,node);
        }
    }
}

inline void HLDConstruct()
```

```
{
    cnt = 0;
    chainNo = 0;
    mem(chainHead,-1);
    mem(chainSize,0);
    HLD(1,-1);
    for(int i = 1; i<=n*4; i++)
        tree[i] = 0;
}
/// HLD End

void update(int node, int st, int ed, int i, int v)
{
    if(st == ed && st == i)
    {
        tree[node] += v;
        return;
    }

    int lft = 2*node, rght = lft+1, mid = (st+ed)/2;
    if(i <= mid)
        update(lft, st, mid, i, v);
    if(i > mid)
        update(rght, mid+1, ed, i, v);
    tree[node] = max(tree[lft], tree[rght]);
}

int query(int node, int st, int ed, int i, int j)
{
    if( st >= i && ed <= j)
        return tree[node];

    int lft = 2*node, rght = lft+1, mid = (st+ed)/2;
    int ret = -inf;
    if(i <= mid)
        ret = max(ret, query(lft, st, mid, i, j));
    if(j > mid)
        ret = max(ret, query(rght, mid+1, ed, i, j));
    return ret;
}

inline int solve(int u, int v)
{
    /// v is below u, level[v] > level[u];
    int ret = -inf;
    while(true)
    {
        if(chainId[u] == chainId[v])
        {
            int L = idxInBaseArray[u];
            int R = idxInBaseArray[v];
            ret = max(ret, query(1,1,n,L,R));
            return ret;
        }
        int head = chainHead[chainId[v]];
        int L = idxInBaseArray[head];
        int R = idxInBaseArray[v];
        ret = max(ret, query(1,1,n,L,R));
        v = parent[head];
    }
}
inline int query(int u, int v)
{
    int lca = findLCA(u,v);
    return max(solve(lca,u), solve(lca, v));
}

inline void update(int a, int v)
{
    int idx = idxInBaseArray[a];
    update(1,1,n,idx,v);
}

int main()
{
    int i, j, cs, t, u, v, q;
    sf(n);
    FRL(i,1,n)
    {
        sff(u,v);
        E[u].pb(v);
        E[v].pb(u);
    }
    initSparseTable(1);
    HLDConstruct();
    sf(q);
    char s[5];
    while(q--)
    {
        scanf("%s %d %d",s,&u,&v);
        if(s[0] == 'I')
            update(u,v);
        else
            printf("%d\n",query(u,v));
    }
    return 0;
}
```

Highly Composite Number:

```
/*
    The following code finds the largest highly composite number less than
limit along with its number of divisors.
    Takes 156 ms for 100 test cases.
    A highly composite number is a positive integer with more divisors than
any smaller positive integer has.

    List:
    limit                HCN  Divisors
    1e1                    6,      4
    1e2                   60,     12
    1e3                  840,     32
    1e4                 7560,     64
    1e5                83160,    128
    1e6               720720,    240
    1e7              8648640,    448
    1e8             73513440,    768
    1e9            735134400,   1344
    1e10          6983776800,   2304
    1e11         97772875200,   4032
    1e12        963761198400,   6720
    1e13       9316358251200,  10752
    1e14      97821761637600,  17280
    1e15     866421317361600,  26880
    1e16    8086598962041600,  41472
    1e17   74801040398884800,  64512
    1e18  897612484786617600, 103680
*/
```

```
LL n;
#define MAX 12
LL primes[MAX] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37};
pll sltn;
void call(int pos, int last, LL curNum, LL curDiv)
{
    if(sltn.yy < curDiv)
        sltn = {curNum, curDiv};
    else if(sltn.yy == curDiv && sltn.xx > curNum)
        sltn = {curNum, curDiv};

    if(pos == MAX)
        return;

    for(int i = 1; i<=last; i++)
    {
        if(curNum > n/primes[pos])
            break;
        curNum *= primes[pos];
        call(pos+1, i, curNum, curDiv*(i+1));
    }
}
int main()
{
    int i = 1;
    for(n = 10; ; n*=10)
    {
        sltn.xx = sltn.yy = 0;
        call(0, 100, 1, 1);
        printf("1e%d ",i);
        if(i < 10)
            printf(" ");
        printf("%19lld, %6lld\n",sltn.xx, sltn.yy);
        if(n == 1e18)
            break;
        i++;
    }
    return 0;
}
```

IDA* Search:

```
/*
    IDAStar to solve 15 puzzle
    Finds the minimum steps to reach solution. Finds the lexicographical
minimum among the best.
    For 15 puzzle, there can be inputs that might need around 200 moves.
    ID_DFS() terminates after 35 steps.
    Manhattan distance from actual position is used as heuristics.
    Remember, heuristics should always give less than or equal to the actual
answer.
    Heuristics can never overestimate the answer.

    For 8 puzzle, highest number of steps needed is 31 ( probably ) .
    Critical(31 steps) case for 8 puzzle:
    8 6 7
    2 5 4
    3 0 1
    Impossible checking for 8 puzzle:
    if(inv %2 ) return -1; ( impossible to solve if inversion count is odd )
*/
/// op[i] is the opposite move of move i.
int A[5][5], dest[5][5], R[20], C[20], op[5];

/// D L R U
int dy[] = {+0,-1,+1,+0};
int dx[] = {+1,+0,+0,-1};
int depth;
vector<int>sltn;
char Move[10];

bool ok(int nr, int nc)
{
    if(nr > 0 && nc > 0 && nr <= 4 && nc <= 4)
        return true;
    return false;
}

/// heuristics function
int heuristics(int nw, int r, int c)
{
    return abs(R[nw] - r) + abs(C[nw] - c);
```

```
}

int dfs(int r, int c, int d, int lst,  int h)
{
    int i, j, nr, nc;

    if(h == 0)
        return true;
    if(h + d > depth)
        return false;

    for(i = 0; i<4; i++)
    {
        /// Don't go to parent state
        if(lst >= 0 && op[lst] == i)
            continue;
        nr = r, nc = c;
        nr += dx[i];
        nc += dy[i];
        if( ok(nr,nc))
        {
            int nw = A[nr][nc];

            h -= heuristics(nw, nr, nc);
            swap(A[r][c],A[nr][nc]);
            h += heuristics(nw,r,c);
            sltn.pb(i);
            if(dfs(nr,nc,d+1,i,h))
                return true;

            sltn.pop_back();
            h += heuristics(nw,nr,nc);
            swap(A[r][c],A[nr][nc]);
            h -= heuristics(nw,r,c);
        }
    }
    return false;
}
int ID_DFS()
{
    int i, j, x, y, inv = 0, cnt = 0, h = 0;
    /// calculate heuristics for given input and count inversions
    for(i = 1; i<=4; i++)
    {
        for(j = 1; j<=4; j++)
        {
            if(A[i][j] == 0)
                x = i, y = j;
            if(A[i][j])
                h += heuristics(A[i][j], i, j);
            for(int k = i; k<=4; k++)
            {
                int l = 1;
                if(k == i) l = j+1;
                for(; l <= 4; l++)
                    if(A[i][j] && A[k][l] && A[i][j] > A[k][l])
                        inv++;
            }
        }
    }

    /// Impossible checking for 15 puzzle
    if( (inv %2 && (4-x+1)%2 == 1) || ( inv%2 == 0 && (4-x+1)%2 == 0))
        return -1;

    /// Increase depth by 1 of IDDFS
    for(depth = h; depth <= 35; depth++)
    {
        if(dfs(x,y,0,-10,h))
            return depth;
    }
    /// Solution not found within 35 moves
    return -1;
}

void pre()
{
    Move[0] = 'D';
    Move[1] = 'L';
    Move[2] = 'R';
    Move[3] = 'U';

    op[0] = 3;
    op[3] = 0;
    op[1] = 2;
    op[2] = 1;
    int cnt = 1;
    for(int i = 1; i<=4; i++)
    {
        for(int j = 1; j<=4; j++)
            dest[i][j] = cnt, R[cnt] = i, C[cnt] = j, cnt++;
    }
}

void solve()
{
    sltn.clear();
    int ans = ID_DFS();
    if(ans == -1)
        printf("This puzzle is not solvable.");
    else
    {
        for(int i = 0; i<sltn.size(); i++)
            printf("%c",Move[sltn[i]]);
    }
    puts("");
}

int main()
{
    int cs, t, i, j, k;

    pre();
    sf(t);
    FRE(cs,1,t)
    {
        for(i = 1; i<=4; i++)
        {
            for(j = 1; j<=4; j++)
                scanf("%d",&A[i][j]);
        }
        printf("Case %d: ",cs);
        solve();
    }
    return 0;
}
```

Implicit Segment Tree:

```
/// Implicit Seg Tree
/// For long range
/// Range update, Range query AC code
/// When indices can be negative,
/// int mid = floor((st+ed)/2.00);
/// will be useful. That's why it's written like this

struct node{
    LL sum, lazy;
    node *lft,*rght;
    node()
    {
        sum = lazy = 0;
        lft = rght = NULL;
    }
};

void propagate(node *nd, int st, int ed)
{
    int mid = floor((st+ed)/2.00);
    nd->lft->sum += (mid-st+1)*nd->lazy;
    nd->rght->sum += (ed-mid)* nd->lazy;
    nd->lft->lazy += nd->lazy;
    nd->rght->lazy += nd->lazy;
    nd->lazy = 0;
}

void update(node *nd, int st, int ed, int i, int j, LL v)
{
    if(st >= i && ed <= j)
    {
        nd->sum+= (ed-st+1)*v;
        nd->lazy+= v;
        return;
    }

    if(nd->lft == NULL)
        nd->lft = new node();
    if(nd->rght == NULL)
        nd->rght = new node();
    if(nd->lazy)
        propagate(nd, st, ed);

    int mid = floor((st+ed)/2.00);


    if(i <= mid)
        update(nd->lft, st, mid, i, j, v);

    if(j > mid)
        update(nd->rght, mid+1, ed, i, j, v);
    nd->sum = nd->lft->sum + nd->rght->sum;
}

LL query(node *nd, int st, int ed, int i, int j)
{
    if(st >= i && ed <= j)
        return nd->sum;

    if(nd->lft == NULL)
        nd->lft = new node();
    if(nd->rght == NULL)
        nd->rght = new node();

    if(nd->lazy)
        propagate(nd, st, ed);

    int mid = floor((st+ed)/2.00);
    LL ret = 0;
    if(i <= mid)
        ret+= query(nd->lft, st, mid, i, j);

    if(j > mid)
        ret+= query(nd->rght, mid+1, ed, i, j);

    return ret;
}

void rmv(node *cur)
{
    if(!cur) return;
    rmv(cur->lft);
    rmv(cur->rght);
    delete(cur);
}


int main()
{
//    freopen("in.txt","r",stdin);
//    freopen("out.txt","w",stdout);
    int i, j, cs, t, type, n, q, v;
    sf(t);
    FRE(cs,1,t)
    {
        sff(n,q);
        node *root = new node();
        while(q--)
        {
            sfff(type, i, j);
            if(type == 0)
                sf(v), update(root, 1, n, i, j, v);
            else
                printf("%lld\n",query(root, 1, n, i, j));
        }
        rmv(root);
    }
    return 0;
}
```

Iterative BigMod:

```
inline LL bigMod(LL a, LL n, LL MOD)
{
    LL ret = 1%MOD;
    a %= MOD;
    while(n)
    {
        if(n&1) ret *= a, ret %= MOD;
        a *= a;
        a %= MOD;
        n >>= 1;
    }
    return ret;
}
```

Josephus:

```
//Josephus [ 0 indexed ] O (n) Recursive
int josephus(int n, int k)
{
    if(n == 1) return 0;
    return (josephus(n-1, k) + k)%n;
}

//  [ 0 Indexed] O(n) Iterative
int josephus(int n,int k)
{
    int bachbe=0,first_morbe,i;
    for(i=2;i<=n;i++)
    {
        first_morbe=(k)%i;
        bachbe=(first_morbe+bachbe)%i;
    }
    return bachbe;
}
```

KMP:

```
int P[MAX+10];
/// P[i] is the length of the longest proper prefix which is also a proper
suffix of the string S[0..i]
int prefixFunction(char *S)
{
    int now;
    P[0] = now = 0;
    int len = strlen(S);
    for(int i = 1; i<len; i++)
    {
        while(now != 0 && S[now] != S[i])
            now = P[now-1];
        if(S[now] == S[i])
            P[i] = ++now;
        else
            P[i] = now = 0;
    }
}

/// checks if pattern is a substring of S
bool KMPMatcher(char *S, char *pattern)
{
    int now = 0;
    int lenStr = strlen(S);
    int lenPat = strlen(pattern);
    prefixFunction(pattern);
    for(int i = 0; i<lenStr; i++)
    {
        while(now != 0 && pattern[now] != S[i])
            now = P[now-1];
        if(pattern[now] == S[i])
            now++;
        else
            now = 0;
        if(now == lenPat)
            return 1;
    }
    return 0;
}
```

Lagrange Multiplier:

```
/*
    A constraint equation Cons()
    Cons() must be equation, not inequality. If problem gives you inequality
like,
    t1 + t2 <= T
    then try to prove that there is no point in keeping (t1+t2) < T and solve
it for (t1+t2) = T. Lagrange can only solve for equations, not inequalities.

    A objective function ( minimization or maximization ) Obj()
    It should be the case that either minimization or maximization for the
given objective function is very easy and the other is very tough and has to
be solved by lagrange multiplier.
    lambda * D(Cons()) = D(Obj) ,where D(F(x)) is d/dxi(F(xi)) [partial
differentiation with respect to xi]
    Find out the lower or upper bound of lambda from the above function if
```

```
needed ( using fact such as no probability can be negative etc), otherwise
set infinity to lower and upper bounds.
    Then do a binary search over lambda to find out which value of lambda
forces the variables to be equal to constraint. That lambda value is the
lagrange multiplier and now you know the individual values for the variables
which optimizes the given objective function
    If you have additional constraints, try bitmasking or a greedy choice to
pick optimal set of variables so that the weaker variables don't violate
additional constraint.
*/
```

Problem:
Suppose you are in a 2-Dimensional world. Now, you are in a system of ‘N’ parallel zones of same or different speed, numbered from 0 to N-1. In each zone you can move in some given constant speed (Si amount per second in i-th zone) at any direction. Each zone is parallel to X axis, starting from the X axis (and then on the positive X and positive Y part only). Width of each zone is 100 (along the Y axis). You are currently in the origin (0, 0). You need to reach (100*N, D) coordinate. But, you want to do that in minimum possible time (seconds).

Solution:

The problem can be reworded as: Choose $x_i \in \mathbb{R}$, such that

$$\sum_{i=0}^{N-1} \frac{\sqrt{100^2+x_i^2}}{s_i}$$

is minimum, and

$$\sum_{i=0}^{N-1} x_i = D$$

An easy way to solve this is by using Lagrange multipliers:

$$\frac{\partial}{\partial x_i}(\sum_{i=0}^{N-1} \frac{\sqrt{100^2+x_i^2}}{s_i} - \lambda(\sum_{i=0}^{N-1} x_i - D)) = 0$$

$$\frac{1}{s_i}\frac{1}{2\sqrt{100^2+x_i^2}}2x_i - \lambda = 0$$

$$x_i = \frac{100\lambda s_i}{1-\lambda^2 s_i^2}$$

If we write this as:

$$x_i = \frac{100 s_i}{\frac{1}{\lambda^2} - s_i^2}$$

We can see that as λ increases(decreases), x_i increases(decreases), so we can do a binary search for the value of λ that satisfies

$$\sum_{i=0}^{N-1} x_i = D$$

Problem:

You have a total of T hours that you can split among different subjects. For each subject i, the expected grade with t hours of studying is given by the function $f_i(t) = a_i t^2 + b_i t + c_i$, satisfying the following properties:

- $f_i(0) \geq 0$;
- $f_i(T) \leq 100$;
- $a_i < 0$;
- $f_i(t)$ is a non-decreasing function in the interval $[0, T]$.

You may allocate any fraction of an hour to a subject, not just whole hours. What is the maximum average grade you can obtain over all n subjects?

Input

The first line of each input contains the integers N $(1 \leq N \leq 10)$ and T $(1 \leq T \leq 240)$ separated by a space. This is followed by N lines, each containing the three parameters a_i, b_i, and c_i describing the function $f_i(t)$. The three parameters are separated by a space, and are given as real numbers with 4 decimal places. Their absolute values are no more than 100.

Output

Output in a single line the maximum average grade you can obtain. Answers within 0.01 of the correct answer will be accepted.

Solution:

```
struct data{
    double a, b, c;
    double eval(double t)
    {
        return a*t*t + b*t + c;
    }
}F[15];

int n, T;
const double inf = 1e9;

double getUpper(vector<data>&F)
{
    double lambda = inf;
    for(auto u: F)
        lambda = min(lambda, u.b);
    return lambda;
}

double constraint(double lambda, vector<data>&F)
{
    double ret = 0;
    for(auto u: F)
    {
        ret += (lambda - u.b)/(2*u.a);

    }
    return ret;
}

double solve(int mask)
{
    vector<data>myF;
    double others = 0;
    for(int i = 0; i<n; i++)
    {
        if(checkBit(mask, i))
            myF.pb(F[i]);
        else
            others += F[i].eval(0);
    }
    double lo = -inf, hi = getUpper(myF), mid;
    if(constraint(hi, myF)+eps > T)
        return -inf;

    for(int i = 1; i<=100; i++)
    {
        double mid = (lo+hi)/2;
        if(constraint(mid, myF) < T)
            hi = mid;
        else
            lo = mid;
    }
    double ret = others, lambda = lo;
    for(auto u: myF)
        ret += u.eval((lambda - u.b)/(2*u.a));
    return ret;
}

double solve()
```

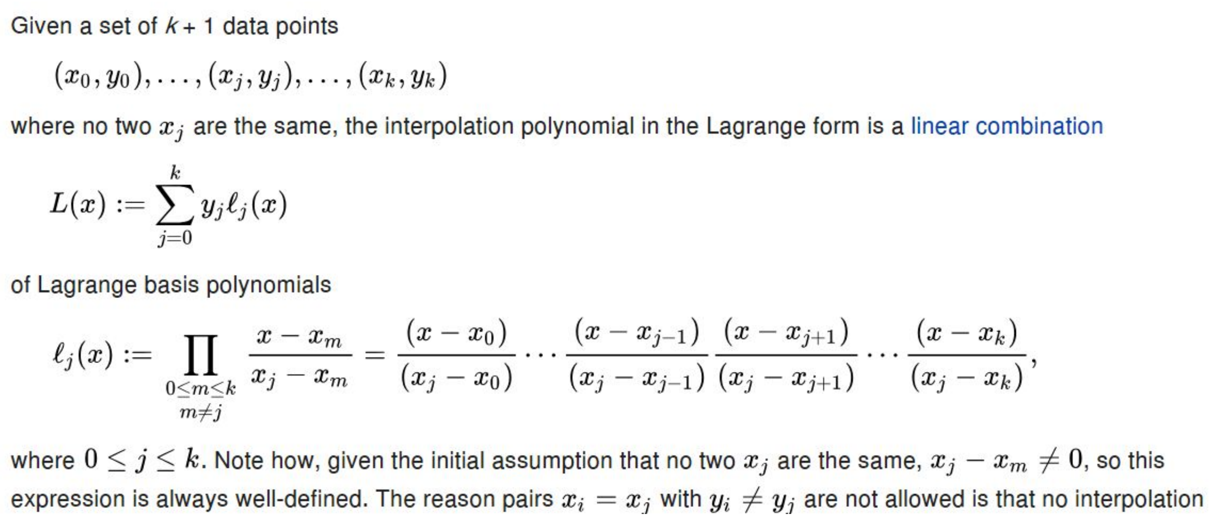
```
{
    int lim = (1<<n)-1;
    double mx = 0;
    for(int i = 1; i<=lim; i++)
        mx = max(mx, solve(i));
    return mx;
}

int main()
{
    sff(n,T);
    for(int i = 0; i<n;i++)
        scanf("%lf %lf %lf",&F[i].a, &F[i].b, &F[i].c);
    printf("%.6f\n",solve()/n);
    return 0;
}
```

Lagrange Polynomial:

```
/*
    F is a k-degree polynomial but you don't know the co-efficients.
    Given the value of F(i) for (k+1) points, Find F(n).
    If the (k+1) points have a pattern or are sequential, it can be solved in
    O(klogk). If not, then it can be solved in O(k^2).

    If you need to find the unique k-degree polynomial, it can be done in
O(k^3) naively.
    It can also be done using D&C with FFT in O(nlognlogn).
*/
Basic Idea:
```



Given a set of $k + 1$ data points

$$(x_0, y_0), \ldots, (x_j, y_j), \ldots, (x_k, y_k)$$

where no two x_j are the same, the interpolation polynomial in the Lagrange form is a linear combination

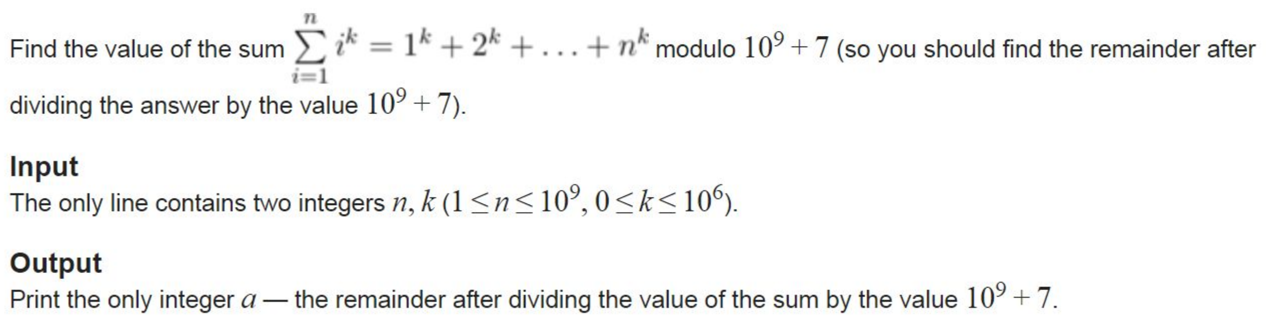
$$L(x) := \sum_{j=0}^{k} y_j \ell_j(x)$$

of Lagrange basis polynomials

$$\ell_j(x) := \prod_{\substack{0 \le m \le k \\ m \ne j}} \frac{x - x_m}{x_j - x_m} = \frac{(x - x_0)}{(x_j - x_0)} \cdots \frac{(x - x_{j-1})}{(x_j - x_{j-1})} \frac{(x - x_{j+1})}{(x_j - x_{j+1})} \cdots \frac{(x - x_k)}{(x_j - x_k)},$$

where $0 \le j \le k$. Note how, given the initial assumption that no two x_j are the same, $x_j - x_m \ne 0$, so this expression is always well-defined. The reason pairs $x_i = x_j$ with $y_i \ne y_j$ are not allowed is that no interpolation

Problem Statement:



Find the value of the sum $\sum_{i=1}^{n} i^k = 1^k + 2^k + \ldots + n^k$ modulo $10^9 + 7$ (so you should find the remainder after dividing the answer by the value $10^9 + 7$).

Input
The only line contains two integers n, k $(1 \le n \le 10^9, 0 \le k \le 10^6)$.

Output
Print the only integer a — the remainder after dividing the value of the sum by the value $10^9 + 7$.

```
LL n;
int k;
const LL MOD = 1000000007;

LL expo(LL a, LL n)
{
    if(n == 0)
        return 1;
    if(n&1)
        return (a*expo(a,n-1))%MOD;
    LL ret = expo(a, n/2);
    return (ret * ret)%MOD;
}
LL F[1000010];
LL lagrange(LL F[], int total, LL n)
{
    if(1<=n && n <= total)
```

```
        return F[n];
    LL up = 1, down = 1;
    for(int i = 2; i<=total; i++)
    {
        LL nw = n - i;
        if(nw < 0)
            nw += MOD;
        up *= (nw);
        up %= MOD;

        nw = (1-i);
        if(nw < 0)
            nw += MOD;
        down *= nw;
        down %= MOD;
    }

    LL ret = (up * F[1])%MOD;
    ret *= expo(down, MOD-2);
    ret %= MOD;
    for(int i = 2; i<=total; i++)
    {
        LL nw = n - (i-1);
        if(nw < 0)
            nw += MOD;
        up *= (nw);
        up %= MOD;

        nw = (n-i);
        if(nw < 0)
            nw += MOD;
        down *= nw;
        down %= MOD;

        down *= (i-1);
        down %= MOD;

        nw = ((i-1)-total);
        if(nw < 0)
            nw += MOD;
        up *= nw;
        up %= MOD;
        LL tmp = (up * F[i])%MOD;
        tmp *= expo(down, MOD-2);
        tmp %= MOD;

        ret += tmp;
        if(ret >= MOD)
            ret -= MOD;
    }
    return ret;
}

int main()
{
    sl(n);
    sf(k);
    for(int i = 1; i<=k+2; i++)
    {
        F[i] = F[i-1] + expo(i,k);
        if(F[i] >= MOD)
            F[i] -= MOD;
    }

    printf("%lld\n",lagrange(F,k+2,n));
    return 0;
}
```

Link Cut Tree For Rooted Trees (No lazy propagation) :

```
/*
    1 based indexing
    can link or cut edge of rooted trees in O(logn). Usually has a constant
factor equal to 2*constant factor of segment tree or similar.
    Change the update function as you need. Remember the node has only
information about the BBST ( splay tree here ) it is part of.
    It cannot store anything related to its subtree in the original tree.
    Call Access(u) to make a splay tree with the nodes from the root to u. u
will also be made the root of the splay tree.
    Whenever you need to change a node, access it and then update it. Since
it will be at the root of its splay tree, updating its value
    will not affect anyone else. You can also just splay it to get the same
effect. But it is better to access it.
```

```
    create an object of the class link cut tree to use it. Alternatively, to
make the code faster you can change the code for dynamically allocating
memory for
    the node pointers in that class and erase the function overloading
default destructor.
    ***The link function makes u a child of v. It must be the case that u is
a root of a tree.Also u and v
    must be in different trees.
    Don't write your own functions completely inside the class. Rather write
the complete function globally with pointers as
    parameters and write helper function inside class where you send only the
node ids and call the global function from there.
    Just like how link is written here.
*/
struct Node
{
    int sz, label; /* size, label */ /// keep additional information here as
you need
    Node *p, *pp, *l, *r; /* parent, path-parent, left, right pointers in
splay tree*/
    Node()
    {
        p = pp = l = r = 0; /// initialize some of the custom variables here
if needed. ( like mx)
    }
};

void update(Node *x) /// change here as you need.
{
    x->sz = 1;
    if(x->l) x->sz += x->l->sz;
    if(x->r) x->sz += x->r->sz;
}

void rotr(Node *x)
{
    Node *y, *z;
    y = x->p, z = y->p;
    if((y->l = x->r)) y->l->p = y;
    x->r = y, y->p = x;
    if((x->p = z))
    {
        if(y == z->l) z->l = x;
        else z->r = x;
    }
    x->pp = y->pp;
    y->pp = 0;
    update(y);
}

void rotl(Node *x)
{
    Node *y, *z;
    y = x->p, z = y->p;
    if((y->r = x->l)) y->r->p = y;
    x->l = y, y->p = x;
    if((x->p = z))
    {
        if(y == z->l) z->l = x;
        else z->r = x;
    }
    x->pp = y->pp;
    y->pp = 0;
    update(y);
}

void splay(Node *x)
{
    Node *y, *z;
    while(x->p)
    {
        y = x->p;
        if(y->p == 0)
        {
            if(x == y->l) rotr(x);
            else rotl(x);
        }
        else
        {
            z = y->p;
            if(y == z->l)
            {
                if(x == y->l) rotr(y), rotr(x);
                else rotl(x), rotr(x);
            }
            else
            {
                if(x == y->r) rotl(y), rotl(x);
                else rotr(x), rotl(x);
            }
        }
    }
    update(x);
}

Node *access(Node *x)
{
    splay(x);
    if(x->r)
    {
        x->r->pp = x;
        x->r->p = 0;
        x->r = 0;
        update(x);
    }

    Node *last = x;
    while(x->pp)
    {
        Node *y = x->pp;
        last = y;
        splay(y);
        if(y->r)
        {
            y->r->pp = y;
            y->r->p = 0;
        }
        y->r = x;
        x->p = y;
        x->pp = 0;
        update(y);
        splay(x);
    }
    return last;
}

Node *root(Node *x)
{
    access(x);
    while(x->l) x = x->l;
    splay(x);
    return x;
}

void cut(Node *x)
{
    access(x);
    x->l->p = 0;
    x->l = 0;
    update(x);
}

void link(Node *x, Node *y)
{
    access(x);
    access(y);
    x->l = y;
    y->p = x;
    update(x);
}

Node *lca(Node *x, Node *y)
{
    access(x);
    return access(y);
}

int depth(Node *x)
{
    access(x);
    return x->sz - 1;
}

class LinkCut
{
public:
    Node *x;

    LinkCut(int n)
    {
        x = new Node[n+5];
        for(int i = 1; i <= n; i++)
        {
            x[i].label = i;
            update(&x[i]); /// initialize your custom variables here.
        }
    }

    virtual ~LinkCut()
    {
        delete[] x;
    }

    /// ***The link function makes u a child of v. It must be the case that u
is a root of a tree.
    ///    Also u and v must be in different trees.
    void link(int u, int v)
    {
        ::link(&x[u], &x[v]);
    }

    void cut(int u)
    {
        ::cut(&x[u]);
    }

    int root(int u)
    {
        return ::root(&x[u])->label;
    }

    int depth(int u)
    {
        return ::depth(&x[u]);
    }

    int lca(int u, int v)
    {
        return ::lca(&x[u], &x[v])->label;
    }
    void access(int u)
    {
        ::access(&x[u]);
    }
};
```

Link Cut Tree forUnrooted Trees (With Lazy Propagation) :

```
/*
    1 based indexing
    can link or cut edge of unrooted trees in O(logn). You have a current
root of a tree but you can make any node u, the root of its tree
    by calling makeroot(u). Makeroot uses lazy propagation technique.
normalize is the function for propagating your lazy value
    to your right and left child in the splay tree. If you only need the lazy
propagation part but not the undirected tree part
    of the code, write your normalize function and call it at the same places
as here.
    When cut(a,b) is called it must be the case that a and b has a direct
edge between them which will be cut.
*/
struct Node
{
    int sz, label, flip; /* size, label, flip for evert operation */ /// keep
additional information here as you need
    Node *p, *pp, *l, *r; /* parent, path-parent, left, right pointers in
splay tree*/
    Node()
    {
        p = pp = l = r = 0; /// initialize some of the custom variables here
if needed. ( like mx)
        flip = 0;
    }
};

void normalize(Node *x)
{
    if(x->flip)
    {
        if(x->l)
        {
            x->l->flip ^= 1;
            swap(x->l->l, x->l->r);
        }
        if(x->r)
        {
            x->r->flip ^= 1;
            swap(x->r->l, x->r->r);
        }
        x->flip = 0;
    }
}



void update(Node *x) /// change here as you need.
{
    x->sz = 1;
    if(x->l) x->sz += x->l->sz;
    if(x->r) x->sz += x->r->sz;
}

void rotr(Node *x)
{
    Node *y, *z;
    y = x->p, z = y->p;
    normalize(y);
    normalize(x);
    if((y->l = x->r)) y->l->p = y;
    x->r = y, y->p = x;
    if((x->p = z))
    {
        if(y == z->l) z->l = x;
        else z->r = x;
    }
    x->pp = y->pp;
    y->pp = 0;
    update(y);
}

void rotl(Node *x)
{
    Node *y, *z;
    y = x->p, z = y->p;
    normalize(y);
    normalize(x);
    if((y->r = x->l)) y->r->p = y;
    x->l = y, y->p = x;
    if((x->p = z))
    {
        if(y == z->l) z->l = x;
        else z->r = x;
    }
    x->pp = y->pp;
    y->pp = 0;
    update(y);
}

void splay(Node *x)
{
    Node *y, *z;
    while(x->p)
    {
        y = x->p;
        if(y->p == 0)
        {
            if(x == y->l) rotr(x);
            else rotl(x);
        }
        else
        {
            z = y->p;
            if(y == z->l)
            {
                if(x == y->l) rotr(y), rotr(x);
                else rotl(x), rotr(x);
            }
            else
            {
                if(x == y->r) rotl(y), rotl(x);
                else rotr(x), rotl(x);
            }
        }
    }
    normalize(x);
    update(x);
}

Node *access(Node *x)
{
    splay(x);
    if(x->r)
    {
        x->r->pp = x;
        x->r->p = 0;
        x->r = 0;
        update(x);
    }

    Node *last = x;
    while(x->pp)
    {
        Node *y = x->pp;
        last = y;
        splay(y);
        if(y->r)
        {
            y->r->pp = y;
            y->r->p = 0;
        }
        y->r = x;
        x->p = y;
        x->pp = 0;
        update(y);
        splay(x);
    }
    return last;
}

void makeRoot(Node *x)
{
    access(x);
    x->flip ^= 1;
    swap(x->l, x->r);
}

int depth(Node *x)
{
    access(x);
    return x->sz - 1;
}

Node *root(Node *x)
{
    access(x);
    while(x->l) x = x->l;
    splay(x);
    return x;
}

void cut(Node *x, Node *y)
{
    if(depth(x) < depth(y))
        swap(x, y);
    access(x);
    x->l->p = 0;
    x->l = 0;
    update(x);
}

void link(Node *x, Node *y)
{
    makeRoot(x);
    access(x);
    access(y);
    x->l = y;
    y->p = x;
    update(x);
}

Node *lca(Node *x, Node *y)
{
    access(x);
    return access(y);
}



class LinkCut
{
public:
    Node *x;

    LinkCut(int n)
    {
        x = new Node[n+5];
        for(int i = 1; i <= n; i++)
        {
            x[i].label = i;
            update(&x[i]); /// initialize your custom variables here.
        }
    }

    virtual ~LinkCut()
    {
        delete[] x;
    }

    /// ***The link function makes u a child of v. It must be the case that u
is a root of a tree.
    ///    Also u and v must be in different trees.
    void link(int u, int v)
    {
        ::link(&x[u], &x[v]);
    }

    void cut(int u, int v)
    {
        ::cut(&x[u], &x[v]);
    }

    int root(int u)
    {
        return ::root(&x[u])->label;
    }

    int depth(int u)
    {
        return ::depth(&x[u]);
    }

    int lca(int u, int v)
    {
        return ::lca(&x[u], &x[v])->label;
    }
    void access(int u)
    {
        ::access(&x[u]);
    }
};
```

Problem:
A forest of unrooted trees initially consists of N (1 ≤ N ≤ 100,000) single-vertex trees. The vertices are numbered from 1 to N.

Your task is to maintain that forest and answer connectivity queries.

All edges in the problem are **undirected**.

You will receive the following queries, where (1 ≤ A, B ≤ N) :

- **add** A B : add an edge between vertices A and B, where initially there is no path between A and B.
- **rem** A B : remove edge between vertices A and B, where initially there is an edge between A and B.
- **conn** A B : print **YES** if there is a path between A and B and **NO** otherwise, where A and B are different.

Solution:

```
struct Node
{
    int sz, label, flip; /* size, label */ /// keep additional information
here as you need
    Node *p, *pp, *l, *r; /* parent, path-parent, left, right pointers in
splay tree*/
    Node()
    {
        p = pp = l = r = 0; /// initialize some of the custom variables here
if needed. ( like mx)
        flip = 0;
    }
};

void normalize(Node *x)
{
    if(x->flip)
    {
        if(x->l)
        {
            x->l->flip ^= 1;
            swap(x->l->l, x->l->r);
        }
        if(x->r)
        {
            x->r->flip ^= 1;
            swap(x->r->l, x->r->r);
        }
        x->flip = 0;
    }
}



void update(Node *x) /// change here as you need.
{
    x->sz = 1;
    if(x->l) x->sz += x->l->sz;
    if(x->r) x->sz += x->r->sz;
}

void rotr(Node *x)
{
    Node *y, *z;
    y = x->p, z = y->p;
    normalize(y);
    normalize(x);
    if((y->l = x->r)) y->l->p = y;
    x->r = y, y->p = x;
    if((x->p = z))
    {
        if(y == z->l) z->l = x;
        else z->r = x;
    }
    x->pp = y->pp;
    y->pp = 0;
    update(y);
}

void rotl(Node *x)
{
    Node *y, *z;
    y = x->p, z = y->p;
    normalize(y);
    normalize(x);
    if((y->r = x->l)) y->r->p = y;
    x->l = y, y->p = x;
    if((x->p = z))
    {
        if(y == z->l) z->l = x;
        else z->r = x;
    }
    x->pp = y->pp;
    y->pp = 0;
    update(y);
}

void splay(Node *x)
{
    Node *y, *z;
    while(x->p)
    {
        y = x->p;
        if(y->p == 0)
        {
            if(x == y->l) rotr(x);
            else rotl(x);
        }
        else
        {
            z = y->p;
            if(y == z->l)
            {
                if(x == y->l) rotr(y), rotr(x);
                else rotl(x), rotr(x);
            }
            else
            {
                if(x == y->r) rotl(y), rotl(x);
                else rotr(x), rotl(x);
            }
        }
    }
    normalize(x);
    update(x);
}

Node *access(Node *x)
{
    splay(x);
    if(x->r)
    {
        x->r->pp = x;
        x->r->p = 0;
        x->r = 0;
        update(x);
    }

    Node *last = x;
    while(x->pp)
    {
        Node *y = x->pp;
        last = y;
        splay(y);
        if(y->r)
        {
            y->r->pp = y;
            y->r->p = 0;
        }
        y->r = x;
        x->p = y;
        x->pp = 0;
        update(y);
        splay(x);
    }
    return last;
}

void makeRoot(Node *x)
{
    access(x);
    x->flip ^= 1;
    swap(x->l, x->r);
}

int depth(Node *x)
{
    access(x);
    return x->sz - 1;
}

Node *root(Node *x)
```

```
{
    access(x);
    while(x->l) x = x->l;
    splay(x);
    return x;
}

void cut(Node *x, Node *y)
{
    if(depth(x) < depth(y))
        swap(x, y);
    access(x);
    x->l->p = 0;
    x->l = 0;
    update(x);
}

void link(Node *x, Node *y)
{
    makeRoot(x);
    access(x);
    access(y);
    x->l = y;
    y->p = x;
    update(x);
}

Node *lca(Node *x, Node *y)
{
    access(x);
    return access(y);
}



class LinkCut
{
public:
    Node *x;

    LinkCut(int n)
    {
        x = new Node[n+5];
        for(int i = 1; i <= n; i++)
        {
            x[i].label = i;
            update(&x[i]); /// initialize your custom variables here.
        }
    }

    virtual ~LinkCut()
    {
        delete[] x;
    }

    /// ***The link function makes u a child of v. It must be the case that u
is a root of a tree.
    ///    Also u and v must be in different trees.
    void link(int u, int v)
    {
        ::link(&x[u], &x[v]);
    }

    void cut(int u, int v)
    {
        ::cut(&x[u], &x[v]);
    }

    int root(int u)
    {
        return ::root(&x[u])->label;
    }

    int depth(int u)
    {
        return ::depth(&x[u]);
    }

    int lca(int u, int v)
    {
        return ::lca(&x[u], &x[v])->label;
    }
    void access(int u)
    {
        ::access(&x[u]);
    }
```

```
};


char s[50];
int main()
{
    int n, m, u, v;
    sf(n);
    LinkCut *tree = new LinkCut(n);
    sf(m);
    while(m--)
    {
        scanf("%s %d %d",s,&u,&v);
        if(s[0] == 'a')
            tree->link(u,v);
        else if(s[0] == 'r')
            tree->cut(u,v);
        else
        {
            if(tree->root(u) == tree->root(v))
                puts("YES");
            else
                puts("NO");
        }
    }
}
```

Longest Increasing Subsequence (O(nlogk)):

```
/// O(nlogk) LIS
/// k is the length of the LIS
/// Code by: Leonardo Boshell ( LOJ Wavio Sequence )
int L[MAX+10];
int lis(int *seq, int *dp, int N)
{
    int len = 0;
    for (int i = 1; i <= N; ++i) {
        int lo = 1, hi = len;
        while (lo <= hi) {
            int m = (lo + hi) / 2;
            if (L[m] < seq[i]) lo = m + 1;
            else hi = m - 1;
        }
        L[lo] = seq[i], dp[i] = lo;
        if (len < lo) len = lo;
    }
    return len;
}
```

Lowest Common Ancestor:

```
/// 1 based indexing
#define MAXLG ?
#define MAXN ?
int parent[MAXN+10], sparseTable[MAXLG+2][MAXN+10], level[MAXN+10], n;

void dfs(int node, int par, int currentLevel)
{
    parent[node] = par;
    parentVal[node] = val[par];
    level[node] = currentLevel+1;
    for(auto v: E[node])
    {
        if(v != par)
            dfs(v, node, currentLevel+1);
    }
}
void initSparseTable(int Root)
{
    dfs(Root,0,1);
    for(int i = 1; i<=n; i++)
        sparseTable[0][i] = parent[i];
    for(int p = 1; p <= MAXLG; p++)
    {
        for(int u = 1; u <= n; u++)
            sparseTable[p][u] = sparseTable[p-1][sparseTable[p-1][u]];
    }
}

int findLCA(int u, int v)
{
    /// keep u as the deeper node
    if(level[v] > level[u])
        swap(u,v);
    for(int i = MAXLG; i>=0; i--)
    {
        if((1<<i) <= level[u] - level[v])
            u = sparseTable[i][u];
    }
    if(u == v)
        return v;
    for(int i = MAXLG; i>=0; i--)
    {
        if(sparseTable[i][u] != sparseTable[i][v])
            u = sparseTable[i][u], v = sparseTable[i][v];
    }
    return parent[v];
}
```

Manacher:

```
#define MAXN ?
/**
*** centerAt : len of longest palindrome center at i
*** centerAfter : len of longest palindrome center after i
*** centerAt is valid for i = 0 to len-1
*** centerAfter is valid for i = 0 to len-2
*** memset these two arrays to zero if you dont want to remember so many
things
**/
int centerAt[MAXN],centerAfter[MAXN];
vector <int> manacher(char *str)
{
    int i = 0, j = 0, k = 0, len = strlen(str), n = len << 1;
    vector <int> pal(n);
    for ( ; i < n; j = max( 0, j - k ), i += k)
    {
        while (j <= i && (i + j + 1) < n && str[(i - j) >> 1] == str[(i + j +
1) >> 1]) j++;
        for (k = 1, pal[i] = j; k <= i && k <= pal[i] && (pal[i] - k) !=
pal[i - k]; k++)
        {
            pal[i + k] = min(pal[i - k], pal[i] - k);
        }
    }
    pal.pop_back();
    return pal;
}
void buildManacher(char *str)
{
    vector < int >  v = manacher(str);
    for(int i=0; i<v.size(); i++)
    {
        if(i&1)
            centerAfter[i/2]=v[i];
        else
            centerAt[i/2]=v[i];
    }
    v.clear();
}
bool isPal(int L, int R)
{
    int zog = L +R,mid = zog/2;
    if(zog&1) return (centerAfter[mid]>=(R-L+1));
    else return centerAt[mid]>=(R-L+1);
}
```

Matrix Exponentiation:

```
/*
    M^(n-1) * A = B
*/
struct mat{
    int r, c;
    LL arr[?][?];
}M, A, id;

mat mul(mat a, mat b)
{
    mat ret;
    ret.r = a.r, ret.c = b.c;
    for(int i = 0; i<a.r; i++)
    {
        for(int j = 0; j<b.c; j++)
        {
            ret.arr[i][j] = 0;
            for(int k = 0; k<a.c; k++)
            {
                ret.arr[i][j] += (a.arr[i][k] * b.arr[k][j])%MOD;
                ret.arr[i][j] %= MOD;
            }
        }
    }
    return ret;
}

void init()
{
    id.r = id.c = ?;
    mem(id.arr,0);
    for(int i = 0; i<id.r; i++)
        id.arr[i][i] = 1;

//    do rest of initiating here.

}

mat mat_exp(mat M, LL n)
{
    mat ret = id;
    while(n)
    {
        if(n&1) ret = mul(ret, M);
        n>>=1;
        M = mul(M,M);
    }
    return ret;
}
```

MaxFlow + BPM Important Notes:

```
/*
    Finding Maximum independent set from BPM ( Kuhn and Karp )
    Run BPM. After that do a dfs from every node in the left side that are
unmatched. Then take an edge from u to v only
    and only if
    i) u is on the left side and match[u] != v
    ii) u is on the right side and match[v] == u

    After this, all visited nodes on the left and all unvisited nodes on the 
right will form the
    maximum independent set.

*/

/*
    Finding the edges in the minimum cut
    After running max flow, do a DFS from source on the residual graph. Then 
an edge from u to v will be in the
    minimum cut if and only if,
    i) u is visited and v is unvisited
    ii) u is unvisited and v is visited

*/

/*
    Fixing lower bound on edges
    Create a super source, a super sink. If u->v has a lower bound of LB, 
give an edge from super source to v with capacity LB.
    Give an edge from u to super sink with capacity LB.
    Give an edge from normal sink to normal source with capacity infinity.
    If maxflow is equal to LB, then the lower bound can be satisfied.
*/

/*
    Finding minimum flow that will make sure lower bound for every edge.
    Usually for lower bound an edge with infinite capacity is given from 
destination to source.
    But now we will binary search on the value of the capacity of the edge 
from the destination
    to the source.
*/

/*
    Finding Maximum flow that will make sure lower bound for every edge.
    Just add a super source and binary search on the upper bound of the edge 
from super source to
    normal source to find a solution satisfying upper bounds of all other 
edges.
*/
```

```
/*
    To solve a maximization problem with max flow, convert it to a
minimization problem somehow and
    then try to find the solution from min cut.
*/

/*
    Project Selection Problem
    Maximize total profit.
    Doing i'th project profits you by P[i]
    Doing i'th project requires you to buy a list of instruments each of
which has different cost.
    Different projects may require the same instrument in which case, buying
one instrument is ok.
    Make a flow graph with projects on the left and instruments on the right.
    cap[source][i'th project] = P[i]
    cap[j'th instrument][sink] = Cost[j]
    cap[i'th project][j'th instrument] = inf ( if i'th project requires j'th
instrument )
    here mincut will minimize this function ( sacrifice profits of projects +
cost of instruments to be bought )
    So ans is = Total profit of all projects - mincut.
*/

/*
    Image Segmentation Problem
    There are n pixels. Each pixel i can be assigned a foreground value  fi
or a background value bi.
    There is a penalty of pij if pixels i, j are adjacent and have different
assignments. The problem is to assign pixels to foreground or background
    such that the sum of their values minus the penalties is maximum.
    Let P be the set of pixels assigned to foreground and Q be the set of
points assigned to background, then the problem can be formulated as
    maximize  ( totalF + totalB - sacrificeFore for Q - sacrificeBack for P -
penalty pij)
    or, minimize( sacrificeFore for Q + sacrificeBack for P + penalty pij)
    The above minimization problem can be formulated as a minimum-cut problem
by constructing a network
    where the source is connected to all the pixels with capacity  fi, and
the sink is connected by all the pixels with capacity bi.
    Two edges (i, j) and (j, i) with pij capacity are added between two
adjacent pixels.
```

```
    The s-t cut-set then represents the pixels assigned to the foreground in
P and pixels assigned to background in Q.
*/
```

Maximum Matching in general graph O(V^2*E):

Resonance

Mincost Maximum Flow:

```
/// 1 based indexing
namespace mcmf{
    const int MAXN = 100, inf = 1e9;
    int src, snk, nNode, dist[MAXN+5], flow[MAXN+5], parent[MAXN+5];
    bool vis[MAXN+5];
    struct edgeData{
        int u, v, cost, cap, flow;
    };
    vector<edgeData> edges;
    vector<int>E[MAXN+5];
    void init(int _src, int _snk, int _n)
    {
        src = _src, snk = _snk, nNode = _n;
        edges.clear();
        for(int i = 1; i<=nNode; i++)
            E[i].clear();
    }

    void inline add(int u, int v, int _cost, int _cap)
    {
        E[u].push_back(edges.size());
        edges.push_back({u,v,_cost,_cap,0});
        E[v].push_back(edges.size());
        edges.push_back({v,u,-_cost,0,0});
    }

    bool inline SPFA()
    {
        queue<int>q;
        mem(vis,false);
        for(int i = 1; i<=nNode; i++)
            flow[i] = dist[i] = inf;
        vis[src] = true;
        dist[src] = 0;
        q.push(src);
        while(!q.empty())
        {
            int u = q.front();
            q.pop();
            vis[u] = false;
            for(int i = 0; i<E[u].size(); i++)
            {
                int idx = E[u][i];
                int v = edges[idx].v;
                int cost = edges[idx].cost;
                int cap = edges[idx].cap;
                int f = edges[idx].flow;
                if( f < cap &&  dist[v] > dist[u] + cost)
                {
                    dist[v] = dist[u]+cost;
                    parent[v] = idx;
                    flow[v] = min(flow[u], cap - f);
                    if(!vis[v])
                    {
                        q.push(v);
                        vis[v] = true;
                    }
                }
            }
        }
        return dist[snk] != inf;
    }

    pair<int,int> solve()
    {
        int totalFlow = 0, cost = 0;
        while(SPFA())
        {
            int u = snk;
            totalFlow += flow[snk];
            while(u != src)
            {
                edges[parent[u]].flow += flow[snk];
                edges[parent[u]^1].flow -= flow[snk];
                u =  edges[parent[u]].u;
            }
            cost += dist[snk] * flow[snk];
        }
        return {totalFlow, cost};
    }
}

int main()
{
    mcmf::init(1,5,5);
    mcmf::add(1,2,-1,1);
    mcmf::add(1,3,-5,2);
    mcmf::add(2,4,-10,100);
    mcmf::add(3,4,-1,100);
    mcmf::add(4,5,0,2);
    pii a = mcmf::solve();
    DD(a.xx, a.yy);
    /// should be 2 flow, -17 cost.
    return 0;
}
```

Minimum Enclosing Sphere:

```
struct point3D{
    double x, y, z;
    point3D(){}
    point3D(double xx, double yy, double zz):x(xx),y(yy),z(zz){}
}P[MAX+5];

struct sphere{
    point3D center;
    double r;
    sphere(){}
    sphere(point3D p, double rr):center(p), r(rr){}
};

double abs(double x, double y, double z)
{
    return (x*x+y*y+z*z);
}

sphere minimumEnclosingSphere(point3D arr[], int n) // 1 based indexing
{
    point3D piv = point3D(0,0,0);

    int i, j;
    for(i = 1; i <= n; i++)
    {
        piv.x += arr[i].x;
        piv.y += arr[i].y;
        piv.z += arr[i].z;
    }

    piv.x /= n;
    piv.y /= n;
    piv.z /= n;

    double p = 0.1, e, d;

    for (i = 0; i < 70000; i++) // better to have 50K+
    {
      int f = 0;
      d = numeric_limits<double>::min();
      for (j = 1; j <= n; j++)
      {
            e = abs(piv.x - arr[j].x, piv.y - arr[j].y, piv.z - arr[j].z);
            if (d < e) {
                  d = e;
                  f = j;
            }
      }

      piv.x += (arr[f].x - piv.x)*p;
      piv.y += (arr[f].y - piv.y)*p;
      piv.z += (arr[f].z - piv.z)*p;
      p *= 0.998;
    }

    return sphere(piv, sqrt(d));
}
```

Modular Inverse with Extended Euclid (Iterative):

```
/// Finds Modular inverse of a with respect to MOD
inline int inv(int a, int MOD)
{
    int x = 0,y = 1,u = 1,v = 0;
    int e = MOD,f = a;
    int c,d,q,r;
    while(f != 1)
    {
        q = e/f;
        if(e >= f)
            r = e%f;
        else
            r = e;
        c = x-q*u;
        d = y-q*v;
        x = u;
        y = v;
        u = c;
        v = d;
        e = f;
        f = r;
    }
    if(u < 0)
        u += MOD;
    if( u >= MOD)
        u -= MOD;
    return u;
}
```

Modular Inverse of 1 to n in O(n):

```
LL inv[?];
void buildInverse(int lim)
{
    inv[1] = 1;
    for(int i = 2; i<=lim; i++)
    {
        inv[i] = (MOD - ((MOD/i) * inv[MOD%i]) % MOD);
        if(inv[i] < 0)
```

```
            inv[i] += MOD;
    }
}
```

Mo's Algorithm:

```
/*
    Complexity: O(max(n*sqrt(n), q*sqrt(n), q*log(q)))
    Optimizations done:
    1. keep l_bucket
    2. use inline functions
    3. use even odd different sorting
*/
? A[?], ans,  sltn[?];
int freq[?], bucket_size = sqrt(?);

struct data{
    int st, ed, idx, l_bucket;
    bool even;
    data(int a, int b, int i)
    {
        st = a, ed = b, idx = i;
        l_bucket = st/bucket_size;
        if(l_bucket & 1)
            even = false;
        else
            even = true;
    }
    data(){}
};

vector< data > Q;

inline bool cmp(data a, data b)
{
    if(a.l_bucket == b.l_bucket)
    {
        if(a.even)
            return a.ed < b.ed;
        else
            return a.ed > b.ed;
    }
    return a.l_bucket < b.l_bucket;
}

inline void add(int idx)
{

}

inline void rmv(int idx)
{

}


void Mo()
{
    sort(all(Q), cmp);
    ans = 0;
    add(1);
    int l = 1, r = 1;
    for(int i = 0; i<Q.size(); i++)
    {
        while(l < Q[i].st)
            rmv(l), l++;
        while(l > Q[i].st)
            l--, add(l);
        while(r < Q[i].ed)
            r++, add(r);
        while(r > Q[i].ed)
            rmv(r), r--;
        sltn[Q[i].idx] = ans;
    }
}
```

Mo On Tree:

Problem:

This story is happening in a town named BubbleLand. There are n houses in BubbleLand. In each of these n houses lives a boy or a girl. People there really love numbers and everyone has their favorite number f. That means that the boy or girl that lives in the *i*-th house has favorite number equal to *fi*.

The houses are numerated with numbers 1 to *n*.

The houses are connected with *n* - 1 bidirectional roads and you can travel from any house to any other house in the town. There is exactly one path between every pair of houses.

A new dating had agency opened their offices in this mysterious town and the citizens were very excited. They immediately sent *q* questions to the agency and each question was of the following format:

- *a b* — asking how many ways are there to choose a couple (boy and girl) that have the same favorite number and live in one of the houses on the unique path from house *a* to house *b*.

Help the dating agency to answer the questions and grow their business.

Solution:
```
/// Run a dfs and fill up level and parent arrays first.
/// 1 based indexing
#define MAXLG       20
#define MAXN        100000
#define MAXQ        100000
int val[MAXN+10];
vector<int>E[MAXN+10];

int parent[MAXN+10], sparseTable[MAXLG+2][MAXN+10], level[MAXN+10], n,
counter, ara[MAXN*2+10];
int st[MAXN+10], ed[MAXN+10];

void dfs(int node, int par, int cur)
{
    st[node] = ++counter;
    ara[counter] = node;
    parent[node] = par;
    level[node] = cur;
    for(auto v: E[node])
    {
        if(v!=par)
            dfs(v, node, cur+1);
    }
    ed[node] = ++counter;
    ara[counter] = node;
}
void initSparseTable(int Root)
{
    dfs(Root,0,1);
    for(int i = 1; i<=n; i++)
        sparseTable[0][i] = parent[i];
    for(int p = 1; p <= MAXLG; p++)
    {
        for(int u = 1; u <= n; u++)
            sparseTable[p][u] = sparseTable[p-1][sparseTable[p-1][u]];
    }
}

int findLCA(int u, int v)
{
    /// keep u as the deeper node
    if(level[v] > level[u])
        swap(u,v);
    for(int i = MAXLG; i>=0; i--)
    {
        if((1<<i) <= level[u] - level[v])
            u = sparseTable[i][u];
    }
    if(u == v)
        return v;
    for(int i = MAXLG; i>=0; i--)
    {
        if(sparseTable[i][u] != sparseTable[i][v])
            u = sparseTable[i][u], v = sparseTable[i][v];
    }
    return parent[v];
}

/*
    Complexity: O(max(n*sqrt(n), q*sqrt(n), q*log(q)))
    Optimizations done:
    1. keep l_bucket
    2. use inline functions
    3. use even odd different sorting
*/

LL ans,  sltn[MAXQ+10], on[MAXN+10];
int freq[2][MAXN+10], bucket_size = sqrt(MAXQ);
int sex[MAXN+10], both[MAXN+10];
struct data{
    int st, ed, idx, l_bucket, u, v, lca;
    bool even;
    data(int a, int b, int i, int uu, int vv, int LCA)
    {
        u = uu, v = vv, lca = LCA;
        st = a, ed = b, idx = i;
        l_bucket = st/bucket_size;
        if(l_bucket & 1)
            even = false;
        else
            even = true;
    }
    data(){}
};

vector< data > Q;
inline bool cmp(data a, data b)
{
    if(a.l_bucket == b.l_bucket)
    {
        if(a.even)
            return a.ed < b.ed;
        else
            return a.ed > b.ed;
    }
    return a.l_bucket < b.l_bucket;
}

inline void add(int node)
{
    LL nw = (LL)freq[0][val[node]] * freq[1][val[node]];
    ans -= nw;
    freq[sex[node]][val[node]]++;
    nw = (LL)freq[0][val[node]] * freq[1][val[node]];
    ans += nw;
}
```

```
inline void rmv(int node)
{
    LL nw = (LL)freq[0][val[node]] * freq[1][val[node]];
    ans -= nw;
    freq[sex[node]][val[node]]--;
    nw = (LL)freq[0][val[node]] * freq[1][val[node]];
    ans += nw;
}

inline void check(int idx)
{
    int node = ara[idx];
    if(on[node])
        rmv(node);
    else
        add(node);
    on[node] ^= 1;
}

void Mo()
{
    sort(all(Q), cmp);
    ans = 0;
    check(1);
    int l = 1, r = 1;
    for(int i = 0; i<Q.size(); i++)
    {
        while(l < Q[i].st)
            check(l), l++;
        while(l > Q[i].st)
            l--, check(l);
        while(r < Q[i].ed)
            r++, check(r);
        while(r > Q[i].ed)
            check(r), r--;
        if(Q[i].lca != Q[i].u && Q[i].lca != Q[i].v)
            check(st[Q[i].lca]);
        sltn[Q[i].idx] = ans;

        if(Q[i].lca != Q[i].u && Q[i].lca != Q[i].v)
            check(st[Q[i].lca]);
    }
```

```
}

map<int, int>M;

void queryAndSolve()
{
    counter = 0;
    initSparseTable(1);
    int q;
    sf(q);
    for(int i = 1; i<=q; i++)
    {
        int u, v;
        sff(u,v);
        if(st[u] > st[v])
            swap(u, v);
        int lca = findLCA(u,v);
        if(u == lca)
            Q.pb(data(st[u], st[v], i, u, v, lca));
        else
            Q.pb(data(ed[u], st[v], i, u, v, lca));
    }
    Mo();
    for(int i = 1; i<=q; i++)
        printf("%lld\n",sltn[i]);
}

int main()
{
    int u, v, m;
    while(sf(n) == 1)
    {
        mem(freq, 0);
        mem(on, 0);
        Q.clear();
        M.clear();
        for(int i = 1; i<=n; i++)
            sf(sex[i]);
        for(int i = 1; i<=n; i++)
            sf(val[i]), M[val[i]];
        int cnt = 1;
        for(auto &m: M)
            m.yy=cnt++;
        for(int i = 1; i<=n; i++)
            val[i] = M[val[i]];
        for(int i = 1; i<n; i++)
            sff(u,v), E[u].pb(v), E[v].pb(u);

        queryAndSolve();

        for(int i = 1; i<=n; i++)
            E[i].clear();
    }
    return 0;
}
```

Mo Special With No Deletion:

Problem:
You are given a sequence of **n** integers **a1 , a2 , ... , an** in non-decreasing order. In addition to that, you are given several queries consisting of indices **i** and **j** (*1 ≤ i ≤ j ≤ n*). For each query, print one line with one integer: The number of occurrences of the most frequent value within the given range.

Solution:

```
struct queryData{
    int st, ed, id;
};

struct queryBucket{
    int bucketEd;
    vector<queryData>V;
}bucket[320];

bool operator < ( queryData a, queryData b)
{
    return a.ed < b.ed;
}

int A[MAX+10], bucketSize, bucketTotal;

void preProcess(int n)
```

```
{
    bucketSize = sqrt(n);
    bucketTotal = 0;
    for(int i = bucketSize-1; ;i+=bucketSize,bucketTotal++)
    {
        bucket[bucketTotal].bucketEd = min(i,n);
        if(i >= n)
        {
            bucketTotal++;
            break;
        }
    }
}

int freq[MAX*2+10], tmpFreq[MAX*2+10], sltn[MAX+10];
int bruteFreq[MAX*2+10];

int brute(int st, int ed)
{
    int ans = 0;
    for(int i = st; i<=ed; i++)
    {
        int nw = A[i];
        bruteFreq[nw]++;
        ans = max(ans, bruteFreq[nw]);
    }
    for(int i = st; i<=ed; i++)
    {
        int nw = A[i];
        bruteFreq[nw]--;
    }
    return ans;
}

void solveBucket(int idx)
{
    int bucketEd = bucket[idx].bucketEd;
    int curRight = bucketEd+1;
    int ans = 0;
    for(int i = 0; i<bucket[idx].V.size(); i++)
    {
        int st = bucket[idx].V[i].st;
        int ed = bucket[idx].V[i].ed;
        int id = bucket[idx].V[i].id;
        if(ed <= bucket[idx].bucketEd)
        {
            sltn[id] = brute(st,ed);
            continue;
        }
        /// shift rightPointer to ed
        while(curRight <= ed)
        {
            int nw = A[curRight];
            freq[nw]++;
            tmpFreq[nw]++;
            ans = max(ans, tmpFreq[nw]);
            curRight++;
        }

        int tmpAns = ans;

        /// shift leftPointer to st
        for(int j = bucketEd; j>= st; j--)
        {
            int nw = A[j];
            tmpFreq[nw]++;
            tmpAns = max(tmpAns, tmpFreq[nw]);
        }
        sltn[id] = tmpAns;
        /// keep persistency
        for(int j = bucketEd; j>= st; j--)
        {
            int nw = A[j];
            tmpFreq[nw]--;
        }
    }
}

void solve()
{
    for(int i = 0; i<bucketTotal; i++)
    {
        mem(freq,0);
        mem(tmpFreq,0);

        sort(all(bucket[i].V));
        solveBucket(i);
   }
}

int sltn2[MAX+10];
pii bla[MAX+10];
int main()
{
    int n, q, st, ed, cs = 1;
    while(scanf("%d",&n) && n)
    {
        scanf("%d",&q);
        preProcess(n);
        for(int i = 1; i<=n; i++)
            scanf("%d",&A[i]),A[i]+=MAX;

        for(int i = 1; i<=q; i++)
        {
            scanf("%d %d",&st,&ed);
            bla[i] = {st,ed};
            bucket[st/bucketSize].V.push_back({st,ed,i});
        }
        solve();
        for(int i = 1; i<=q; i++)
            printf("%d\n",sltn[i]);
        for(int i = 0; i<=bucketTotal; i++)
            bucket[i].V.clear();
    }
}
```

Mo with Update:

Problem:
You have an array of problems **A** with **N** problems. Each problem has a Difficulty level. **i-th** has dificulty **A[i]**.

You want to give this problems to HrSiam. But you know, HrSiam doesn't like easy problems, so he will get angry. Also he doesn't want to solve problems with same difficulty again and again.

So he also has an array Angry, with N elements.

When he visits a problem with certain difficulty **d**, his angriness will increase by **d * angry[1]**.

After some time if he again visits a problem with same difficulty, for the second time, then his angriness will increase by **d * angry[2]**, and so on.

Basically, if he visit a problem with certain difficulty **d** for the **i-th** time, his angrines will increase by **d * angry[i].**

In this circumstances, you want to do 2 things -

1. You are afrad that HrSiam may get too angry, so you want to present a part of the array to him. But you need to quickly determine what will be his total angriness if you present the sub array of problems **A[l...r]** to HrSiam?

2. After certain time you may want to change difficulty of a problem, i.e set difficulty of **x-th** problem to **y**.

solution:
```
/// O(n^(5/3)) only point update, range query
#define MAX         ?
int k = cbrt(MAX*MAX);
int n, Q, A[MAX+10], B[MAX+10], angry[MAX+10];
struct data{
    int st, ed, stBlock, edBlock, t, idx;
    data(int a, int b, int u, int i)
    {
        idx = i;
        st = a;
        ed = b;
        stBlock = st/k;
        edBlock = ed/k;
        t = u;
    }
};

bool operator < (data a, data b)
{
    if(a.stBlock != b.stBlock)
        return a.stBlock < b.stBlock;
    if(a.edBlock != b.edBlock)
        return a.edBlock < b.edBlock;
    if(a.edBlock&1)
        return a.t > b.t;
    else
        return a.t < b.t;
}

vector<data>query;

struct upData{
    int idx, prev, cur;
}U[MAX+10];


LL ans = 0, freq[MAX+10];

inline void add(int idx)
{
    freq[A[idx]]++;
    ans += (LL)angry[freq[A[idx]]] * A[idx];
}

inline void rmv(int idx)
{
    ans -= (LL)angry[freq[A[idx]]] * A[idx];
    freq[A[idx]]--;
}

int L, R;

inline void apply(int idx, int x)
{
    if(L <= idx && idx <= R)
        rmv(idx);
    A[idx] = x;
    if(L <= idx && idx <= R)
        add(idx);
}
LL res[MAX+10];
void Mo()
{
    ans = 0;
    add(1);
    L = R = 1;
    int t = 0;
    for(int i = 0; i<query.size(); i++)
    {
        while(t < query[i].t) t++, apply(U[t].idx, U[t].cur);
        while(t > query[i].t) apply(U[t].idx, U[t].prev), t--;

        while(L < query[i].st)
            rmv(L), L++;
        while(L > query[i].st)
            L--, add(L);
        while(R < query[i].ed)
            R++, add(R);
        while(R > query[i].ed)
            rmv(R), R--;
        res[query[i].idx] = ans;
    }
}

int main()
{
    sf(n);
    for(int i = 1; i<=n; i++)
        sf(A[i]), B[i] = A[i];
    for(int i = 1; i<=n; i++)
        sf(angry[i]);
    sf(Q);
    int type, x, y, t = 0;
    int cnt = 1;
    for(int i = 1; i<=Q; i++)
    {
        sfff(type, x, y);
        if(type == 1)
            query.pb(data(x,y,t,cnt++));
        else
            U[++t] ={x, B[x], y}, B[x] = y;
    }

    sort(all(query));
    Mo();
    for(int i = 1; i<cnt; i++)
        printf("%lld\n",res[i]);
    return 0;
}
```

N Choose R all possible scenarios:

```
If n,r <= 5000,
    solution DP
Otherwise the following cases.
1. M is a prime but n, r >= M
Use simple Lucas
2. M is a squarefree number
Use Lucas with CRT
3. M is any number
Use RM Library
```

Number of Divisors in O(n^(1/3)):

```
/// n <= 1e18

N = input()
primes = array containing primes till 10^6
ans = 1
for all p in primes :
            if p*p*p > N:
                  break
            count = 1
            while N divisible by p:
                  N = N/p
                  count = count + 1
            ans = ans * count
if N is prime:
            ans = ans * 2
else if N is square of a prime:
```

```
            ans = ans * 3
else if N != 1:
            ans = ans * 4
```

PBDS:

```
#include <ext/pb_ds/assoc_container.hpp> // Common file
#include <ext/pb_ds/tree_policy.hpp> // Including
tree_order_statistics_node_update
using namespace __gnu_pbds;
/// for set

typedef tree<
int,
null_type,
less<int>,
rb_tree_tag,
tree_order_statistics_node_update>
orderedSet;

orderedSet X;
X.find_by_order(a); /// returns iterator to the a'th ( 0 based ) largest
value of the set
X.order_of_key(a); /// returns number of elements with value less than a.



/// for multiset
typedef tree < pair<int,int>, null_type, less<pair<int,int>>, rb_tree_tag,
tree_order_statistics_node_update > ordered_set;

struct ordered_multi_set
{
    ordered_set st;
    int inf = 2100000000;
    int T;
    int Size() { return st.size(); }
    void init() { st.clear(); T = 1; }
    void Insert(int x) { st.insert( {x,++T} ); }
    void Erase(int x) { st.erase( st.lower_bound( {x,0} ) ); }
    bool isThere(int x) {
        auto it = st.lower_bound({x, 0});
        if(it == st.end() || it->xx != x)
            return false;
        else
            return true;
    }
    ///0 based index, returns -inf if invalid
    int kth(int k)
    {
        if( k < 0 || k >= st.size()) return -inf;
        auto it = st.find_by_order(k);
        return it->xx;
    }
    ///number of values less than x or index(0 based) of first x if it is in
here
    int less_than(int x) { return st.order_of_key( {x,0} ); }
};
```

Persistent Segment Tree Without Pointers:

Problem:
You have an array of N integers, named **Version-0** array.

You need to do **Q** queries. There are 2 type of queries.

1. **idx pos v**: Take **Version-idx** array and copy it into another array. Name the new array **Version-K** array where **K** = (number of queries of 1st type before this query + 1). Then add **v** the element at index **pos** in **Version-K** array.
2. **idx l r**: In **Version-idx** array, sum the elements from index **l** to **r**. Print the sum of the range

Solution:

```
#define MAX     ?

struct data{
    int lft, rght;
    int sum;
    data()
    {
        lft = rght = -1;
        sum = 0;
    }
    inline void Copy(data& copyFrom)
    {
        lft = copyFrom.lft;
        rght = copyFrom.rght;
        sum = copyFrom.sum;
    }
}tree[MAX*4*20+10], dummy;

int total, version[MAX+10];

inline int create(data& copyFrom)
{
      total++;
      tree[total].Copy(copyFrom);
      return total;
}

int A[MAX+10];

void init(int node, int st, int ed)
{
    if(st == ed)
    {
        tree[node].sum = A[st];
        return;
    }
    int mid = (st+ed)/2;

    tree[node].lft = create(dummy);
    init(tree[node].lft, st, mid);

    tree[node].rght = create(dummy);
    init(tree[node].rght, mid+1, ed);

    int lft = tree[node].lft, rght = tree[node].rght;
    tree[node].sum = tree[lft].sum + tree[rght].sum;
}

void update(int node, int st, int ed, int pos, int val)
```

```
{
    if(st == ed)
    {
        tree[node].sum += val;
        return;
    }

    int lft = tree[node].lft, rght = tree[node].rght, mid = (st+ed)/2;

    if(pos <= mid)
    {
        lft = tree[node].lft = create(tree[lft]);
        update(lft, st, mid, pos, val);
    }
    else
    {
        rght = tree[node].rght =  create(tree[rght]);
        update(rght, mid+1, ed, pos, val);
    }
    tree[node].sum = tree[lft].sum + tree[rght].sum;
}

int query(int node, int st, int ed, int i, int j)
{
    if(i <= st && ed <= j)
        return tree[node].sum;

    int mid = (st+ed)/2;
    int ret = 0;
    if(i <= mid)
        ret += query(tree[node].lft, st, mid, i, j);
    if(j > mid)
        ret += query(tree[node].rght, mid+1, ed, i, j);
    return ret;
}

int main()
{
    int n;
    total = 0;
    sf(n);
    for(int i = 1; i<=n; i++)
        sf(A[i]);
    version[0] = create(dummy);
    init(version[0],1,n);
    int q;
    sf(q);
    int type, idx, pos, st, ed, v;
    int cnt = 0;
    for(int i = 1; i<=q; i++)
    {
        sff(type, idx);
        if(type == 1)
        {
            sff(pos, v);
            version[++cnt] = create(tree[version[idx]]);
            update(version[cnt], 1, n, pos, v);
        }
        else
        {
            sff(st, ed);
            printf("%d\n",query(version[idx], 1, n, st, ed));
        }
    }
    return 0;
}
```

Persistent Segment Tree With Pointers:

```
#define MAX     ?

struct data{
    data *lft, *rght;
    int sum;
    data()
    {
        lft = rght = NULL;
        sum = 0;
    }
    data(data* copyFrom)
    {
        lft = copyFrom->lft;
        rght = copyFrom->rght;
        sum = copyFrom->sum;
    }
```

```
};
data *version[MAX+10];

int A[MAX+10];

void init(data *node, int st, int ed)
{
    if(st == ed)
    {
        node->sum = A[st];
        return;
    }
    int mid = (st+ed)/2;

    node->lft = new data();
    init(node->lft, st, mid);

    node->rght = new data();
    init(node->rght, mid+1, ed);

    node->sum = node->lft->sum + node->rght->sum;
}

void update(data* node, int st, int ed, int pos, int val, data *copyFrom)
{
    if(st == ed)
    {
        node->sum += val;
        return;
    }

    int mid = (st+ed)/2;
    if(pos <= mid)
    {
        node->lft = new data(copyFrom->lft);
        update(node->lft, st, mid, pos, val, copyFrom->lft);
    }
    else
    {
        node->rght = new data(copyFrom->rght);
        update(node->rght, mid+1, ed, pos, val, copyFrom->rght);
    }
    node->sum = node->lft->sum + node->rght->sum;
```

```
}

int query(data* node, int st, int ed, int i, int j)
{
    if(i <= st && ed <= j)
        return node->sum;

    int mid = (st+ed)/2;
    int ret = 0;
    if(i <= mid)
        ret += query(node->lft, st, mid, i, j);
    if(j > mid)
        ret += query(node->rght, mid+1, ed, i, j);
    return ret;
}

int main()
{
    int n;
    sf(n);
    for(int i = 1; i<=n; i++)
        sf(A[i]);
    version[0] = new data();
    init(version[0],1,n);
    int q;
    sf(q);
    int type, idx, pos, st, ed, v;
    int cnt = 0;
    for(int i = 1; i<=q; i++)
    {
        sff(type, idx);
        if(type == 1)
        {
            sff(pos, v);
            version[++cnt] = new data(version[idx]);
            update(version[cnt], 1, n, pos, v, version[idx]);
        }
        else
        {
            sff(st, ed);
            printf("%d\n",query(version[idx], 1, n, st, ed));
        }
    }
    return 0;
}
```

Pollard Rho:

```
const long long LIM = LLONG_MAX;

/// accurate, slow
long long mul(long long a, long long b, long long m){
    long long x, res;

    if (a < b) swap(a, b);
    if (!b) return 0;
    if (a < (LIM / b)) return ((a * b) % m);

    res = 0, x = (a % m);
    while (b){
        if (b & 1){
            res = res + x;
            if (res >= m) res -= m;
        }
        b >>= 1;
        x <<= 1;
        if (x >= m) x -= m;
    }

    return res;
}

/// floating point precision issues, faster
//inline long long mul(long long a, long long b, long long MOD)
//{
//    long double res = a;
//    res *= b;
//    long long c = (long long)(res / MOD);
//    a *= b;
//    a -= c * MOD;
//    if (a >= MOD) a -= MOD;
//    if (a < 0) a += MOD;
//    return a;
//}
```

```
long long expo(long long x, long long n, long long m){
    long long res = 1;

    while (n){
        if (n & 1) res = mul(res, x, m);
        x = mul(x, x, m);
        n >>= 1;
    }

    return (res % m);
}

const int small_primes[] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41,
43, 47, 51, 53, 59, 61, 67, 71};

bool miller_rabin(long long p, int lim){
    long long a, s, m, x, y;
    s = p - 1, y = p - 1;
    while (!(s & 1)) s >>= 1;

    while (lim--){
        x = s;
        a = (rand() % y) + 1;
        m = expo(a, x, p);

        while ((x != y) && (m != 1) && (m != y)){
            m = mul(m, m, p);
            x <<= 1;
        }
        if ((m != y) && !(x & 1)) return false;
    }

    return true;
}

void brent_pollard_rho(uint64_t n, vector <uint64_t> &v){
    if (miller_rabin(n, 10)){
        v.push_back(n);
        return;
    }

    uint64_t a, g, x, y;
    y = 1;
    a = rand() % n;
    x = rand() % n;

    for (int i = 0; ((i * i) >> 1) < n; i++){
        x = mul(x, x, n);
        x += a;
        if (x < a) x += (ULLONG_MAX - n) + 1;
        x %= n;

        g = __gcd(n, y - x);
        if ((g != 1) && (g != n)){
            n /= g;
            brent_pollard_rho(g, v);
            if (n != g) brent_pollard_rho(n, v);
            else if (miller_rabin(n, 10)) v.push_back(n);
            return;
        }

        if (!(i & (i - 1))) y = x;
    }
    brent_pollard_rho(n, v);
}

void factorize(uint64_t n, vector <uint64_t> &v){
    srand(time(0));
    int i, j, x;

    for (i = 0; i < 21; i++){
        x = small_primes[i];
        while ((n % x) == 0){
            n /= x;
            v.push_back(x);
        }
    }

    if (n > 1) brent_pollard_rho(n, v);
    sort(v.begin(), v.end());
}
```

Rabin Miller Primality Testing:

```
/*
    Rabin Miller Primality Testing
    Russian Peasant Multiplication with Nafis vai's addition is used for
multiplying two large numbers.
    O(iterations * log(p) * log(p) )
    Probabilistic algorithm.
    If N iterations are performed on a composite number, then the probability
that it passes each test is 1/4^N or less.
    But there's also another good thing that says Rabin Miller will not fail
for any p < 2^64 if 12 iterations are done with the first 12 primes.
    " The primality of numbers < 2^64 can be determined by asserting strong
pseudoprimality to all prime bases <= 37 (=prime(12)).
    Testing to prime bases <=31 does not suffice, as a(11) < 2^64 and a(11)
is a strong pseudoprime to all prime bases <= 31"

    Smallest odd number for which Miller-Rabin primality test on bases <=
n-th prime does not reveal compositeness.
    2047, 1373653, 25326001, 3215031751, 2152302898747, 3474749660383,
341550071728321, 341550071728321, 3825123056546413051,
    3825123056546413051, 3825123056546413051, 318665857834031151167461,
3317044064679887385961981

*/

inline ULL add(ULL a, ULL b, ULL m)
{
    a%= m, b%= m;
    if(m-b > a)
        return (a+b);
    else
        return (a-(m-b));
}

inline ULL mul(ULL a, ULL b, ULL m)
{
    a%= m, b%= m;
    int i = 0;
    ULL ret = 0;
    while(b)
    {
        if(b&1)
            ret = add(ret, a, m);
        a = add(a, a, m);
        b/=2;
    }
    return ret;
}

int A[20] = {2,3,5,7,11,13,17,19,23,29,31,37,41,43};


inline LL ip(LL a, LL b, LL mod)
{
    LL ret;
    ret = 1;
    while(b)
    {
        if(b&1)
            ret = (mul(ret , a, mod)) ;
        b = b >> 1;
        a = mul(a,a,mod) ;
    }
    return ret;
}
bool RabinMiller(LL p, int iterations)
{
    if(p < 2)
        return false;
    if( p == 2)
        return true;
    LL d, a;
    int s = 0;
    d = p-1;
    while((d&1) == 0)
    {
        s++;
        d = d >> 1;
    }
    for(int i = 0; i< iterations; i++)
    {
        a = A[i] % (p-1);
        if(a == 0)
            a = p-1;
        a = ip(a,d,p);
        if(a == 1 || a == p-1)
            continue;
        for(int i = 1; i<s; i++)
        {
            a = mul(a,a,p);
            if(a == 1)
                return false;
            if(a == p-1)
                break;
        }
        if(a != p-1)
            return false;
    }
    return true;
}
```

Rectangle Union with Segment Tree:

```
/*
    Finding number of non-zero points in a range will only work if there is
no update in any of the parent/ancestor nodes of the node of the range.
*/
struct nodeData{
    int command, on;
    nodeData *lft, *rght;
    nodeData()
    {
        command = on = 0;
        lft = rght = NULL;
    }
}*root;

void update(nodeData* node, int st, int ed, int i, int j, int v)
{
    int mid = (st+ed)/2;
    if(st >= i && ed <= j)
    {
        node->command+=v;
        if(node->command > 0)
            node->on = (ed-st+1);
        else
        {
            node->on = 0;
            if(node->lft != NULL)
                node->on += node->lft->on;
            if(node->rght != NULL)
                node->on += node->rght->on;
        }
        return;
    }

    if(i <= mid)
    {
        if(node->lft == NULL)
            node->lft = new nodeData();
        update(node->lft, st, mid, i, j, v);
    }
    if(j > mid)
    {
        if(node->rght == NULL)
            node->rght = new nodeData();
        update(node->rght, mid+1, ed, i, j, v);
    }
    if(node->command == 0)
    {
        node->on = 0;
        if(node->lft != NULL)
            node->on += node->lft->on;
        if(node->rght != NULL)
            node->on += node->rght->on;
    }
}

int query()
{
    return root->on;
}
struct segment{
    int ySt, yEd, x;
    bool st;
};

bool operator < (segment a, segment b)
```

```
{
    return a.x < b.x;
}

vector<segment>V;

LL rectangleUnion(vector<segment>&V)
{
    sort(V.begin(), V.end());
    int mx = 0;
    for(int i = 0; i<V.size(); i++)
        mx = max(mx, V[i].yEd);
    int n = mx;

    int last = -1;
    LL ret = 0;
    for(int i = 0; i<V.size(); i++)
    {
        if(last != -1)
            ret += (LL)(V[i].x - last)* query();
        last = V[i].x;
        if(V[i].st == true)
            update(root,1,n,V[i].ySt+1,V[i].yEd,1);
        else
            update(root,1,n,V[i].ySt+1,V[i].yEd,-1);
    }
    return ret;
}
```

RMQ Static Data:

```
///RANGE_MINIMUM_QUEREY[STATIC DATA]
///Preprocessing O(n log n)
///Query O(1)
///Shanto vai's book has a nice explanation

#define MAXLG   17
#define MAXN    30000

struct data{
    int val, idx;
    data(){;}
    data(int v, int i){
        val = v;
        idx = i;
    }
}rmq[MAXLG+5][MAXN+10];
data min(data a, data b) { return (a.val <= b.val)?a:b;}


void preprocess(int in[], int n) // -> 1 based indexing [must be]
{
    int stp, i;
    for(stp = 0; (1<<stp) <= n; stp++)
        for(i = 1; i <= n; i++)
        {
            if(!stp) rmq[stp][i] = data(in[i], i);
            else if(i + (1<<stp) -1 > n) break;
            else rmq[stp][i] = min(rmq[stp-1][i],
rmq[stp-1][i+(1<<(stp-1))]);
        }
}

data query(int l, int r)
{
    int mxs = sizeof(int) * 8 - 1 - __builtin_clz(r+1-l);
    return min(rmq[mxs][l], rmq[mxs][r-(1<<mxs)+1]);
}
```

Rope:

```
#include <ext/rope> //header with rope
using namespace __gnu_cxx;
/// push_back(element), substr(st,len), erase(st,len), insert(st,rope), + for
concatenation

rope <char> R;
string str = "12345678";
/// inserting a string character by character into a rope.
for(auto s: str)
    R.push_back(s);
/// substr(st,len) works in log(n) time. It returns a rope starting from st
with len elements.
```

```
/// sub and R will share memory for some time till sub or R is not modified.
rope<char>sub = R.substr(2,3);

/// you can access and even modify a rope at random positions in O(log(n))
cout << R[2] << endl;
R.mutable_reference_at(2) = 'a';
cout << R[2] << endl;

/// erase(st,len) works in log(n) time too. It erases len amount of elements
starting from position st.
R.erase(2, 3);
/// insert(st,rope) works in log(n). It inserts the rope given in the
parameter at position st.
R.insert(4, sub);

/// concatenation of ropes. Guessed it. It's also log(n)
R = R + sub + R;
/// iterate over the rope. You can use auto it = R.begin(). But in that case
you
/// will only get a const iterator and won't be able to modify the element.
for(auto it= R.mutable_begin(); it!=R.mutable_end(); it++)
    *it = 'q';
```

Root of Quadratic equation:

```
ax^2 + bx + c = 0
x = ( -b {+ or -} sqrt(b^2-4ac))/(2a);
```

Russian Peasant Multiplication:

```
/*
    Russian Peasant Multiplication with Nafis vai's addition.
    O(log(b))
    Gives (a*b)%m
    where (a,b,m) < 2^64
*/

ULL add(ULL a, ULL b, ULL m)
{
    a%= m, b%= m;
    if(m-b > a)
        return (a+b);
    else
        return (a-(m-b));
}

ULL mul(ULL a, ULL b, ULL m)
{
    a%= m, b%= m;
    int i = 0;
    ULL ret = 0;
    while(b)
    {
        if(b&1)
            ret = add(ret, a, m);
        a = add(a, a, m);
        b/=2;
    }
    return ret;
}
```

Segmented Sieve:

```
/*
    Segmented Sieve
    This code was for 1 <= a <= b <= 2^31-1
    Change variable types appropriately.
*/
bool notPrime[ ? ];
void segmented_sieve(int a, int b)
{
    int p, f;
    mem(notPrime, 0);
    for(int i = 0; i<tot_prime; i++)
    {
        p = prime[i];
        if(a%p == 0) f = a;
        else f = (a - (a%p)+ p);

        f = max( p*p, f);
        for(unsigned j = f; j<=b; j+=p)
            notPrime[j-a] = true;
    }
    if(a == 1)
        notPrime[0] = 1;
}
```

SOS DP:

```
/*
    O(n*2^n) dp with O(2^n) space
    dp[i][mask] denotes that you can go to any submask of the given mask but
you cannot change any bits before the i-th bit.
*/

void SOS(int N)
{
    /// base case ??
    for(int i = 0;i < N; ++i)
    {
        for(int mask = 0; mask < (1<<N); ++mask)
        {
            if(mask & (1<<i))
                F[mask] += F[mask^(1<<i)];
        }
    }
}
```

Problem:
Two integers *x* and *y* are compatible, if the result of their bitwise "AND" equals zero, You are given an array of integers *a*1, *a*2, ..., *an*. Print *n* integers *ansi*. If *ai* isn't compatible with any other element of the given array *a*1, *a*2, ..., *an*, then *ansi* should be equal to -1. Otherwise *ansi* is any such number, that *ai* & *ansi* = 0, and also *ansi* occurs in the array *a*1, *a*2, ..., *an*.

Solution:

```
int F[1<<22];
void SOS(int N)
{
    for(int i = 0;i < N; ++i)
    {
        for(int mask = 0; mask < (1<<N); ++mask)
        {
            if(mask & (1<<i))
                F[mask] = max(F[mask], F[mask^(1<<i)]);
        }
    }
}

int ulta(int a)
{
    for(int i = 0; i<22; i++)
        if(checkBit(a,i))
            a = resetBit(a, i);
        else
            a = setBit(a, i);
    return a;
}
#define MAX     1000000
int A[MAX+10];
int main()
{
    int n;
    sf(n);
    mem(F,-1);
    for(int i = 1; i<=n; i++)
        sf(A[i]), F[A[i]] = A[i];
    SOS(22);
    for(int i = 1; i<=n; i++)
        printf("%d ",F[ulta(A[i])]);
    puts("");
    return 0;
}
```

Stable Marriage:

```
/*
there are n companies who require one employee each and there are n
candidates. All the candidates interviewed in each of the companies
and eventually they have different preferences over the companies and the
companies have different preferences over the candidates.

So, you are given the task to assign each candidate to each company such that
the employment scheme is stable. A scheme is stable if there is no pair
```

```
(candidatei, companyj) and (candidatex, companyy) where

a)      Candidatei prefers companyy more than companyj and
b)      Companyy prefers candidatei more than candidatex.
*/

/// Code: Zobayer Vai
/// Complexity: O(N^2)
const int MAX = 128;

int m, L[MAX][MAX], R[MAX][MAX], L2R[MAX], R2L[MAX], p[MAX];

void stableMarriage()
{
    int i, man, wom, hubby;
    memset(R2L, -1, sizeof R2L);
    memset(p, 0, sizeof p);
    for(i = 0; i < m; i++ )
    {
        man = i;
        while(man >= 0)
        {
            while(true)
            {
                wom = L[man][p[man]++];
                if(R2L[wom] < 0 || R[wom][man] > R[wom][R2L[wom]]) break;
            }
            hubby = R2L[wom];
            R2L[L2R[man] = wom] = man;
            man = hubby;
        }
    }
}

int main()
{
    int test, cs, i, j, pref;
    scanf("%d", &test);
    for(cs = 1; cs <= test; cs++)
    {
        scanf("%d", &m);
        for(i = 0; i < m; i++)
        {
            for(j = 0; j < m; j++)
            {
                scanf("%d", &L[i][j]);
                L[i][j] -= (m+1);
            }
        }
        for(j = 0; j < m; j++)
        {
            for(i = 0; i < m; i++)
            {
                scanf("%d", &pref);
                R[j][pref-1] = m-i;
            }
        }
        stableMarriage();
        printf("Case %d:", cs);
        for(i = 0; i < m; i++) printf(" (%d %d)", i+1, L2R[i]+m+1);
        printf("\n");
    }
    return 0;
}
```

Stirling Numbers:

```
/// Finds the number ways to put n balls into k indistinguishable boxes such
that no box is empty`.
int stirling2(int n, int k)
{
    if(n < k)
        return 0;
    if(k == 1)
        return 1;
    if(dp[n][k] == dp[n][k])
        return dp[n][k];
    return dp[n][k] = stirling2(n-1,k-1) + stirling2(n-1,k)*k;

}

/// Finds the number of ways to put n elements into k cycles where no cycle
is empty
```

```
int stirling1(int n, int k)
{
    dp[n][k] = stirling1(n-1,k-1) + stirling(n-1,k)*n-1;
}
```

STL:

```
/// bit manipulation
__builtin_popcount(mask); ( number of on bits in the mask )
__builtin_parity(x): This function is used to check the parity of a number.
This function returns true
if the number has odd parity else it returns false for even parity.
__builtin_clz(x): this function is used to count the leading zeros of the
integer. Note : clz = count leading zero's
__builtin_ctz(x): This function is used to count the trailing zeros of the
given integer. Note : ctz = count trailing zeros.

/// C++ Fast IO:
ios_base::sync_with_stdio(false);
cin.tie(NULL);
/*
    numeric_limits<long double>::max()
    numeric_limits<long double>::min()
    numeric_limits<long double>::lowest()
    LLONG_MAX
    ULLONG_MAX
    UINT_MAX
    INT_MAX
/*

iterator:
    map<int,int>::iterator myIt;
new operator:
    int **ara;
    ara = new int*[5];
    ara[2] = new int[10];
Vector:
    Resize for Vector:
        V.resize(5);
    Assign for Vector:
        V.assign(5,1);
        V = [1,1,1,1,1];
    Erase for Vector:
        V.erase(V.begin()+2) ( Removes the third element )
        V.erase(V.begin()+2, V.begin()+5) ( Removes from the third to the
FIFTH element )
lower_bound and upper_bound: ( Never use with set!)
    lower_bound(V.begin(),V.end(),val);
    lower_bound returns an iterator pointing to the first element in the
range [first,last) which has a value not less than ‘val’.
    upper_bound returns an iterator pointing to the first element in the
range [first,last) which has a value greater than ‘val’.
    Complexity: On non-random-access iterators, the iterator advances produce
themselves an additional linear complexity in N on average.
Map:
    M.erase(s);
        removes the key value pair where key is s
    Map iterator is a key value pair.
Set:
    S.erase(40) ( erases the element 40 from the set )
    S.erase(it) ( erases element pointed by iterator it )
    S.erase(it, S.end()) From it till S.end(), the whole range is erased
    S.insert(v.begin()+2, v.begin()+5) inserts v[2] to v[4] to the set
    S.lower_bound(val) returns iterator to first element that is not less
than val in the set
    S.upper_bound(val) returns similar that is greater than val
Multiset:
    Same as set.
    Inserts same element at the end of the range of current same elements
**After erasing an iterator, that iterator becomes invalidated. Don't use it
anymore.
Functions:
    advance(it,n)
        equivalent to: it+=n.   Takes O(n*logn)time for non-random access
iterators.
                                Takes O(1) time for random access iterators.
    next_permutation(v.begin(),v.end()):
    it permutes the given range and then sees if it is sorted. If it is
sorted, then it returns false.That's why you write
do{}while(next_permutation..);
    max_element(v.begin(),v.end())
    min_element(v.begin(),v.end())
    max({3,6,354,12,45})
```

```
string:
    Raw strings. This ignores escape characters.
    string r =
R"(The next is a tab    That really was a tab! and now let's print double
quote "wow"
Wow this is a new line.
)";
    string to_string(double/int/LL);
    int stoi(string a)
    LL stoll(string a)
*/
```

Strongly Connected Component:

```
/*
    Topologically sort the nodes of the main graph.
    Start running dfs from nodes in sorted order in the reverse graph.
    Nodes visited in a single dfs form a SCC.
*/
```

Submasks of All Masks:

```
///O(3^n)
// iterate over all the masks
for (int mask = 0; mask < (1<<n); mask++){
    // iterate over all the subsets of the mask except for 0. Handle 0
manually
    for(int i = mask; i > 0; i = (i-1) & mask)
    {
        ??
    }
}
```

Sudoku Solver:

```
char s[20];
int A[11][11], Row[11][11], Col[11][11], Box[11][11];
int boxNumber(int i, int j)
{
    return ((i-1)/3 * 3) + ceil(j/3.0);
}
```

```
int check(int i, int j)
{
    bool notCandidate[11] = {0};
    int b = boxNumber(i,j), k;
    for(k = 1; k<=9; k++)
        notCandidate[k] |= Row[i][k], notCandidate[k] |= Col[j][k],
notCandidate[k] |= Box[b][k];
    int ret = 0;
    for(k = 1; k<=9; k++)
        ret += (notCandidate[k] == 0);
    return ret;
}

bool call()
{
    int i, j, k;
    pii nxt;
    int mn = inf;
    FRE(i,1,9)
    {
        FRE(j,1,9)
        {
            if(A[i][j] == 0)
            {
                int nw = check(i,j);
                if(nw < mn)
                {
                    mn = nw;
                    nxt = {i,j};
                }
            }
        }
    }
    if(mn == 0)
        return false;

    if(mn == inf)
        return true;

    bool notCandidate[11] = {0};
    i = nxt.xx, j = nxt.yy;
    int b = boxNumber(i,j);
    for(k = 1; k<=9; k++)
    {
        notCandidate[k] |= Row[i][k], notCandidate[k] |= Col[j][k],
notCandidate[k] |= Box[b][k];
        if(!notCandidate[k])
        {
            Row[i][k] = Col[j][k] = Box[b][k] = 1;
            A[i][j] = k;
            if(call())
                return true;
            Row[i][k] = Col[j][k] = Box[b][k] = 0;
            A[i][j] = 0;
        }
    }
    return false;
}

int main()
{
    int i, j, k, cs, t;
    sf(t);
    FRE(cs,1,t)
    {
        mem(Row,0);
        mem(Col,0);
        mem(Box,0);
        FRE(i,1,9)
        {
            scanf("%s",s);
            for(j = 0; j<9; j++)
            {
                if(s[j] == '.')
                    A[i][j+1] = 0;
                else
                {
                    A[i][j+1] = s[j] - '0';
                    Row[i][s[j]-'0'] = 1;
                    Col[j+1][s[j]-'0'] = 1;
                    k = boxNumber(i,j+1);
                    Box[k][s[j]-'0'] = 1;
                }
            }
        }
        call();
        pf("Case %d:\n",cs);
        FRE(i,1,9)
        {
            FRE(j,1,9)
                 pf("%d",A[i][j]) ;
            puts("");
        }
    }
    return 0;
}
```

Suffix Array O(nlognlogn):

```
char text[MAX+10];
struct data{
    int tupleRank[2];
    int idx;
}A[MAX+10];
int sparseTable[MAXLG+2][MAX+10];
bool operator < (data a, data b)
{
    return (a.tupleRank[0] == b.tupleRank[0]) ? (a.tupleRank[1] <
b.tupleRank[1]) : (a.tupleRank[0] < b.tupleRank[0]);
}
int step, SA[MAX+10], n;
void buildSA()
{
    n = strlen(text);
    if(n == 1)
    {
        sparseTable[0][0] = 0;
        SA[0] = 0;
        return;
    }
    for(int i = 0; i<n; i++)
        sparseTable[0][i] = text[i];
    step = 1;
    for(int jump = 1; jump < n; jump <<= 1, step++)
    {
        for(int i = 0; i<n; i++)
        {
            A[i].idx = i;
            A[i].tupleRank[0] = sparseTable[step-1][i];
            A[i].tupleRank[1] = (jump+i < n) ? (sparseTable[step-1][jump+i])
: -1;
        }
        sort(A,A+n);
        sparseTable[step][A[0].idx] = 0;
        for(int i = 1; i<n; i++)
        {
            if(A[i-1].tupleRank[0] == A[i].tupleRank[0] &&
A[i-1].tupleRank[1] == A[i].tupleRank[1])
                sparseTable[step][A[i].idx] = sparseTable[step][A[i-1].idx];
            else
                sparseTable[step][A[i].idx] = i;
        }
    }
    for(int i = 0; i<n; i++)
        SA[sparseTable[step-1][i]] = i;
}

int getLCP(int a, int b)
{
    int ret = 0;
    for(int i = step-1; i>=0; i--)
    {
        if(sparseTable[i][a] == sparseTable[i][b])
        {
            ret += (1<<i);
            a += (1<<i);
            b += (1<<i);
            if(a>=n || b>=n)
                break;
        }
    }
    return ret;
}
```

Suffix Array Others:

```
O(n) -> See Zahin vai
O(nlogn) -> See Zobayer Vai.
```

Template:

```
#include<bits/stdc++.h>
using namespace std;

#define mem(t, v)   memset ((t) , v, sizeof(t))
#define all(x)      x.begin(),x.end()
#define un(x)       x.erase(unique(all(x)), x.end())
#define sf(n)       scanf("%d", &n)
#define sff(a,b)    scanf("%d %d", &a, &b)
#define sfff(a,b,c) scanf("%d %d %d", &a, &b, &c)
#define sl(n)       scanf("%lld", &n)
#define sll(a,b)    scanf("%lld %lld", &a, &b)
#define slll(a,b,c) scanf("%lld %lld %lld", &a, &b, &c)
#define D(x)        cerr << __LINE__ << ": " << #x << " = " << (x) << '\n'
#define DD(x,y)     cerr << __LINE__ << ": " << #x << " = " << x << "   " <<
#y << " = " << y << '\n'
#define DDD(x,y,z)  cerr << __LINE__ << ": " << #x << " = " << x << "   " <<
#y << " = " << y << "   " << #z << " = " << z << '\n'
#define DBG         cerr << __LINE__ << ": Hi" << '\n'
#define pb          push_back
#define PI          acos(-1.00)
#define xx          first
#define yy          second
#define eps         1e-9

typedef unsigned long long int ULL;
typedef long long int LL;
typedef pair<int,int> pii;
typedef pair<LL,LL> pll;


inline int setBit(int N, int pos) { return N=N | (1<<pos); }
inline int resetBit(int N, int pos) { return N= N & ~(1<<pos); }
inline bool checkBit(int N, int pos) { return (bool)(N & (1<<pos)); }

//int fx[] = {+0, +0, +1, -1, -1, +1, -1, +1};
```

```
//int fy[] = {-1, +1, +0, +0, +1, +1, -1, -1}; //Four & Eight Direction

int main()
{
    //freopen("maxon.txt","r",stdin);
    //freopen("out.txt","w",stdout);

    return 0;
}
```

Tree Generation:

```
/// generates a random tree with 10^5 nodes under 1 second. Takes more than 6
seconds to generate random tree with 10^6 nodes
#define MAX     100000
namespace tree{
    int par[MAX+10];
    int Find(int a)
    {
        if(par[a] == a)
            return a;
        return par[a] = Find(par[a]);
    }

    bool Union(int u, int v)
    {
        int a = Find(u), b = Find(v);
        if(a == b)
            return false;
        par[a] = b;
        return true;
    }

    void generateTree(int _n)
    {
        int n = _n;
        srand(time(NULL));
        cout << n << endl;
        for(int i= 1; i<=n; i++)
            par[i] =i;
        for(int i = 1; i<n; i++)
        {
            int u = (rand()*rand())%n+1;
            int v = (rand()*rand())%n+1;
            if(Union(u,v))
                printf("%d %d\n",u,v);
            else
                i--;
        }
    }
}

int main()
{
    freopen("inputTree.txt", "w", stdout);
    tree::generateTree(MAX);
}
```

Unordered Map:

```
struct HASH{
  size_t operator()(const pair<int,int>&x)const{
    return (size_t) x.first * 2411U + (size_t) x.second;
  }
};
unordered_map< pair<int,int>, int, HASH> M;
/// alternatively
#include<bits/stdtr1c++.h>
tr1::unordered_map<int,int>M;
/// Speed up with more memory
M.reserve(1024);
M.max_load_factor(0.25);


/// Better hash tables with 60-70% of time taken by unordered_map
#include <ext/pb_ds/assoc_container.hpp>
using namespace __gnu_pbds;
const int RANDOM = rand();
unsigned hash_f(unsigned x) {
    x = ((x >> 16) ^ x) * 0x45d9f3b;
    x = ((x >> 16) ^ x) * 0x45d9f3b;
    x = (x >> 16) ^ x;
    return x;
}
```

```
struct chash {
    int operator()(int x) const { return hash_f(x); }
};
gp_hash_table<int, int, chash>G; /// preferable, reading/writing is 10%
slower than cc, but 3-4 times
                              /// faster in terms of insertion, clearing,
deletion
/// or alternatively,
cc_hash_table<int, int, chash>C;
```

Z Algorithm:

```
/// z[i] denotes the maximum prefix length of the string which is equal to
the prefix of the suffix starting from i.
int z[MAX+10];
void ZFunction(char *s)
{
    int n = strlen(s);
    int L = 0, R = 0;
    for (int i = 1; i < n; i++)
    {
        if (i > R)
        {
            L = R = i;
            while (R < n && s[R-L] == s[R]) R++;
            z[i] = R-L;
            R--;
        }
        else
        {
            int k = i-L;
            if (z[k] < R-i+1) z[i] = z[k];
            else
            {
                L = i;
                while (R < n && s[R-L] == s[R]) R++;
                z[i] = R-L;
                R--;
            }
        }
    }
}
```

2D BIT:

```
// 1 based indexing
/// Point Update, Range Query
int mxIdx_x, mxIdx_y, tree[MAX+10][MAX+10]; //An array, suppose arr[MAX][MAX]
void init(int nx, int ny)
{
    mxIdx_x = nx;
    mxIdx_y = ny;
    mem(tree,0);
}
void update(int x , int y , int val) //Updating arr[x][y]
{
    int y1;
    while (x <= mxIdx_x)
    {
        y1 = y;
        while (y1 <= mxIdx_y)
        {
            tree[x][y1] += val;
            y1 += (y1 & -y1);
        }
        x += (x & -x);
    }
}
int query(int x , int y) // Cumulative sum from arr[1][1] to arr[x][y]
{
    int y1, ret = 0;
    while (x)
    {
        y1 = y;
        while (y1)
        {
            ret += tree[x][y1];
            y1 -= (y1 & -y1);
        }
        x -= (x & -x);
    }
    return ret;
}
int query_rectangle(int x1, int y1, int x2, int y2) // sum of the values
enclosed by the rectangle x1,y1 and x2,y2 where x1,y1 is the lower corner.
```

```
{
    int ret = query(x2,y2);
    ret -= query(x2, y1-1);
    ret -= query(x1-1, y2);
    ret += query(x1-1,y1-1);
    return ret;
}
```

2D Segment Tree (RMQ):

```
/// Single point update, Range Query
struct node{
    int t[MAX*4 + 10];
}T[MAX*4 + 10];
int n;
int q(int tr, int nd, int st, int ed, int i, int j)
{
    if(st >= i && ed<= j)
        return T[tr].t[nd];
    int lft = 2*nd, rght = lft+1, mid = (ed+st)/2;
    int a, b, ret;
    if(i <= mid)
        a = ret = q(tr, lft, st, mid, i, j);
    if(j > mid)
        b = ret = q(tr, rght, mid+1, ed, i, j);
    if(i <= mid && j > mid)
        ret = max(a,b);
    return ret;
}
int query(int nd, int st, int ed, int i, int j, int _i, int _j)
{
    if(st >= i && ed<= j)
       return q(nd, 1, 1, n, _i, _j);
    int lft = 2*nd, rght = lft+1, mid = (ed+st)/2;
    int a, b, ret;
    if(i <= mid)
        a = ret = query(lft, st, mid, i, j, _i, _j);
    if(j > mid)
        b = ret = query(rght, mid+1, ed, i, j, _i, _j);
    if(i <= mid && j > mid)
        ret = max(a,b);
    return ret;
}
void u(int tr, int nd, int st, int ed, int i, int v)
{
    if(i == st && st == ed)
    {
        T[tr].t[nd] = v;
        return;
    }
    int lft = 2*nd, rght = lft+1, mid = (ed+st)/2;
    if(i <= mid)
        u(tr, lft, st, mid, i, v);
    if(i > mid)
        u(tr, rght, mid+1, ed, i, v);
    T[tr].t[nd] = max(T[tr].t[lft], T[tr].t[rght]);
}
void update(int nd, int st, int ed, int i, int _i, int v)
{
    if(i == st && st == ed)
    {
        u(nd, 1, 1, n, _i, v);
        return;
    }
    int lft = 2*nd, rght = lft+1, mid = (st+ed)/2;
    if(i <= mid)
        update(lft, st, mid, i, _i, v);
    if(i > mid)
        update(rght, mid+1, ed, i, _i, v);
    int a = q(lft, 1, 1, n, _i, _i);
    int b = q(rght, 1, 1, n, _i, _i);
    u(nd, 1, 1, n, _i, max(a,b));
}
```

2-SAT:

Problem:
The ministers of the remote country of Stanistan are having severe problems with their decision making. It all started a few weeks ago when a new process for deciding which bills to pass was introduced. This process works as follows. During each voting session, there are several bills to be voted on. Each minister expresses an opinion by voting either “yes” or “no” for some of these bills. Because of limitations in the design of the technical solution

used to evaluate the actual voting, each minister may vote on only at most four distinct bills (though this does not tend to be a problem, as most ministers only care about a handful of issues). Then, given these votes, the bills that are accepted are chosen in such a way that each minister gets more than half of his or her opinions satisfied. As the astute reader has no doubt already realized, this process can lead to various problems. For instance, what if there are several possible choices satisfying all the ministers, or even worse, what if it is impossible to satisfy all the ministers? And even if the ministers' opinions lead to a unique choice, how is that choice found? Your job is to write a program to help the ministers with some of these issues. Given the ministers' votes, the program must find out whether all the ministers can be satisfied, and if so, determine the decision on those bills for which, given the constraints, there is only one possible choice.

Input
Input consists of multiple test cases. Each test case starts with integers B (1 ≤ B ≤ 100), which is the number of distinct bills to vote on, and M (1 ≤ M ≤ 500), which is the number of ministers. The next M lines give the votes of the ministers. Each such line starts with an integer 1 ≤ k ≤ 4, indicating the number of bills that the minister has voted on, followed by the k votes. Each vote is of the format < bill >< vote >, where < bill > is an integer between 1 and B identifying the bill that is voted on, and < vote > is either 'y' or 'n', indicating that the minister's opinion is "yes" or "no." No minister votes on the same bill more than once. The last test case is followed by a line containing two zeros.

Output
For each test case, print the test case number (starting with 1) followed by the result of the process. If it is impossible to satisfy all ministers, the result should be 'impossible'. Otherwise, the result should be a string of length B, where the i-th character is 'y', 'n', or '?', depending on whether the decision on the i-th bill should be "yes," whether it should be "no," or whether the given votes do not determine the decision on this bill.

Solution:

```
#define MAX         ?

/*
    Assign true or false values to n variables to satisfy in order to satisfy
    a system of constraints on pairs of variables.
    E.g: (x1 or !x2) and (x2 or x3) and (!x3 or !x3)
    x1 = true
    x2 = true
    x3 = false
    is a solution to make the above formula true.

    MAX must be equal to the maximum number of variables.
    n passed in init() is the number of variables.
    O(V+E)
    !a is represented as -a.
    example xor:
    |a|b|
    |0|0| x  or(a,b)
    |0|1|
    |1|0|
    |1|1| x  or(-a,-b)
    do OR of negation of values of variables for each undesired situation to
make it impossible.
*/

struct twoSat{
    int n;
    vector<int> E[MAX*2+10], V, Rev[MAX*2+10], sortedNodes;
    bool state[MAX*2+10], vis[MAX*2+10];
    int compId[MAX*2+10];

    void init(int _n)
    {
        n = _n;
        for(int i = 0; i<=2*n; i++)
            E[i].clear(), Rev[i].clear();
        V.clear();
        sortedNodes.clear();
        mem(state, 0);
    }

    inline int actual(int a)
    {
        if(a < 0)
            return n - a; /// minus e minus e plus.
        else
            return a;
    }

    inline int neg(int a)
```

```
{
    if(a > n)
        return a-n;
    else
        return n+a;
}

void dfs(int node)
{
    vis[node] = true;
    for(auto v: E[node])
    {
        if(!vis[v])
            dfs(v);
    }
    V.push_back(node);
}

void dfsRev(int node, int id)
{
    sortedNodes.push_back(node);
    vis[node] = true;
    compId[node] = id;
    for(auto v: Rev[node])
    {
        if(!vis[v])
            dfsRev(v, id);
    }
}

void buildSCC()
{
    int i;
    V.clear();
    mem(vis,0);
    for(int i = 1; i<=2*n; i++)
    {
        if(!vis[i])
            dfs(i);
    }
    mem(vis,0);
    reverse(all(V));
    int cnt = 0;
```

```
        for(auto u: V)
        {
            if(!vis[u])
                cnt++,dfsRev(u, cnt);
        }
    }

    bool topologicalOrder(int a, int b)
    {
        return compId[a] < compId[b];
    }
    bool satisfy()
    {
        buildSCC();
        /// if leader of i and -i is the same, then they are in the same
component
        /// 2-sat is impossible, return 0
        for(int i = 1; i<=n; i++)
        {
            if(compId[i]==compId[i+n])
                return 0;
        }
        /// topologically sort the components

        /// start from the back end of topologically sorted order and try to
give everyone true state in that component
        /// if someone's opposite has true state, then let him have false
state.
        for(int i = (int)sortedNodes.size()-1; i>=0; i--)
        {
            int u = sortedNodes[i];
            if( state[neg(u)] == 0)
                state[u]=1;
        }
        return 1;
    }

    void addEdge(int u, int v)
    {
        u = actual(u);
        v = actual(v);
        E[u].pb(v);
        Rev[v].pb(u);
    }
    void addOr(int u, int v)
    {
        addEdge(-u, v);
        addEdge(-v, u);
    }

    void addXor(int u, int v)
    {
        addOr(u,v);
        addOr(-u,-v);
    }

    void forceTrue(int u)
    {
        addEdge(-u, u);
    }
    void forceFalse(int u)
    {
        addEdge(u,-u);
    }

    void addOriginalImplication(int u, int v)
    {
        addOr(-u,v);
    }
}solver, solver2;
int n;
void copySolver()
{
    solver2.init(n);
    for(int i = 1; i<=2*n; i++)
        solver2.E[i] = solver.E[i], solver2.Rev[i] = solver.Rev[i];
}

inline int check(int a)
{
    copySolver();
    solver2.forceFalse(a);
    int f = solver2.satisfy();
    copySolver();
    solver2.forceTrue(a);
    int t = solver2.satisfy();
    if(f && t)
        return 1;
    else if(f)
        return 0;
    else
        return 2;
}
int main()
{
    /*
        //Check for testing
    solver.init(3);
    solver.addOr(1, -2);
    solver.addOr(2, 3);
    solver.forceFalse(3);
    assert(solver.satisfy());
    for(int i = 1; i<=3; i++)
        DD(i, solver.state[i]);

    */
    int cs = 0, m, k;
    while(sff(n,m) && (n+m))
    {
        solver.init(n);
        for(int i = 1; i<=m; i++)
        {
            sf(k);
            vector<int>tmp;
            int u;
            char s[5];
            for(int j = 1; j<=k; j++)
            {
                sf(u);
                scanf("%s",s);
                if(s[0] == 'n')
                    u *= -1;
                tmp.pb(u);
            }
            if(k <= 2)
            {
                for(auto u: tmp)
                    solver.forceTrue(u);
            }
            else
            {
                for(auto u: tmp)
                {
                    for(auto v: tmp)
                    {
                        if(u == v)
                            continue;
                        solver.addOriginalImplication(-u,v);
                    }
                }
            }
        }
        printf("Case %d: ",++cs);
        if(!solver.satisfy())
            puts("impossible");
        else
        {
            for(int i = 1; i<=n; i++)
            {
                int res = check(i);
                if(res == 2)
                    printf("y");
                else if(res == 1)
                    printf("?");
                else
                    printf("n");
            }
            puts("");
        }
    }
    return 0;
}
```

CODES:

Grid Compression:

Problem:

Girdland is a square shaped country of size 2000000000 × 2000000000 units. We will identify any point in this land using Cartesian coordinate system. Using this system the coordinate of the center of Gridland is (0, 0), the coordinate of the lower-left corner is (-1000000000, -1000000000) and upper-right corner is (1000000000, 1000000000). So the x-axis is a horizontal line which divides the land into two equal rectangles and y-axis is a vertical line which divides the land into two equal rectangles. There are roads in the country which goes through the grid lines only i.e. along $x = -1000000000, -999999999, \ldots, -1, 0, 1, \ldots, 999999999, 1000000000$ and $y = -1000000000, -999999999, \ldots, -1, 0, 1, \ldots, 999999999, 1000000000$ So all roads are either a horizontal line or a vertical line having integer distance from axis.

Travelers like this Gridland because of the simplicity of its road network. Each of the roads is so straight and axis parallel.

From any position (a, b), in unit time any traveler can move to

i. $(a + 1, b)$

ii. $(a - 1, b)$

iii. $(a, b + 1)$

iv. $(a, b - 1)$

A traveler cannot move outside the Gridland. She also cannot move to any position which is occupied by a monster. There may be some monsters in Gridland. There are three types of monsters

i. Point monster

ii. Line monster

iii. Rectangle monster

A point monster can occupy only a single point, a line monster can occupy a straight line, and a rectangle monster can occupy a rectangular region.

The position of the point monster can be specified by a pair of integers (u, v), which indicates that the monster occupies the coordinate position (u, v).

The position of a line monster can be specified by two pair of integers (u_1, v_1) and (u_2, v_2). (u_1, v_1) is the one end of the line monster and (u_2, v_2) is the other end of the monster. It is guaranteed that this line will always be axis parallel. The monster occupies the whole line region inclusively.

The position of rectangle monster can be specified by two pairs of integers (u_1, v_1) and (u_2, v_2). (u_1, v_1) is the coordinate of the lower left corner of monster and (u_2, v_2) is the upper right corner of the monster. It is guaranteed that the edges of monster will always be axis parallel. The monster occupies the whole rectangular region inclusively.

Initially a traveler is in a position of coordinate $(sourceX, sourceY)$ in Gridland. She needs to reach the position of coordinate $(destinationX, destinationY)$. You have to calculate the minimum unit time required for her to reach the destination. If it is not possible to reach the destination, you have to output '`Impossible`' (quotes for clarity).

Input

Input starts with an integer T (≤ 100), denoting the number of test cases.

Each test case starts with four integer $sourceX$, $sourceY$, $destinationX$, $destinationY$ ($-1000000000 \leq sourceX, sourceY, destinationX, destinationY \leq 1000000000$). Next line contains three integers M, N and Q, the number of point monsters, the number of line monsters and the number of rectangle monsters. Total number of monsters can be at most 100, i.e. ($0 \leq M + N + Q \leq 100$). Each of the next M lines describe a point monster by two integers u and v ($-1000000000 \leq u, v \leq 1000000000$). Each of the next N lines will describes a line monster by four integers u_1, v_1, u_2 and v_2 ($-1000000000 \leq u_1, v_1, u_2, v_2 \leq 1000000000$). Each of the next Q lines will describes a rectangle monster by four integers u_1, v_1, u_2 and v_2 ($-1000000000 \leq u_1, v_1, u_2, v_2 \leq 1000000000$). Note that one monster can overlap with another. You can safely assume that source and destination is not occupied by any monster.

Output

For each test case, output a single line in the format '`Case` C: N', where C will be replaced by the case number and N will be replaced by the shortest path distance from source to destination. If it is not possible to reach source to destination replace 'N' by '`Impossible`' without quotes.

Solution:

```
struct monster{
    LL x1, y1, x2, y2;
}M[110];
LL total;
const LL inf = 1000000000;
map<int,int>X,Y;
LL ultaX[610], ultaY[610];
inline void add(LL x, LL y)
{
    X[x] = X[x-1] = X[x+1] = 1;
    Y[y] = Y[y-1] = Y[y+1] = 1;
}
LL xCnt, yCnt;
void compress()
{
    LL cnt = 0;
    for(auto &m: X)
    {
        if(m.xx < -inf || m.xx > inf)
            continue;
        ultaX[++cnt] = m.xx;
        m.yy = cnt;
    }
    xCnt = cnt;
    cnt = 0;
    for(auto &m: Y)
    {
        if(m.xx < -inf || m.xx > inf)
            continue;
        ultaY[++cnt] = m.xx;
        m.yy = cnt;
    }
    yCnt = cnt;
}
LL csum[610][610];
void block()
{
    mem(csum, 0);
    for(LL i = 0; i<total; i++)
    {
        if(M[i].x1 > M[i].x2)
            swap(M[i].x1, M[i].x2);
        if(M[i].y1 > M[i].y2)
            swap(M[i].y1, M[i].y2);
        for(LL r = X[M[i].x1]; r<=X[M[i].x2]; r++)
        {
            csum[r][Y[M[i].y1]]++;
            csum[r][Y[M[i].y2]+1]--;
        }
    }
    for(LL i = 1; i<=xCnt; i++)
    {
        for(LL j = 1; j<=yCnt; j++)
            csum[i][j] += csum[i][j-1];
    }
}

pii source, dest;

struct data{
    LL x, y;
    LL cost;
};

bool operator < (data a, data b)
{
    return a.cost > b.cost;
}

priority_queue<data>PQ;
LL dist[610][610];

bool ok(LL x, LL y)
{
    if(1 <= x && x <= xCnt && 1 <= y && y <= yCnt)
        return true;
    else
        return false;
}

LL getDist(LL x1, LL y1, LL x2, LL y2)
{
    return llabs(ultaX[x2] - ultaX[x1]) + llabs(ultaY[y2] - ultaY[y1]);
}

LL dijkstra()
{
    while(!PQ.empty())
        PQ.pop();
    mem(dist, 127);
    source = {X[source.xx], Y[source.yy]};
    dist[source.xx][source.yy] = 0;
    PQ.push({source.xx, source.yy, 0});
    dest = {X[dest.xx], Y[dest.yy]};
    while(!PQ.empty())
    {
        data u = PQ.top();
        PQ.pop();
        if(u.x == dest.xx && u.y == dest.yy)
            return u.cost;
        for(LL i = 0; i<4; i++)
        {
            LL r = u.x, c = u.y;
            r += fx[i];
            c += fy[i];
            if(ok(r,c) && csum[r][c] == 0)
            {
                LL curCost = getDist(u.x,u.y,r,c);
                if(dist[r][c] > dist[u.x][u.y] + curCost)
                {
                    dist[r][c] = dist[u.x][u.y] + curCost;
                    PQ.push({r, c, dist[r][c]});
                }
            }
        }
    }
    return -1;
}
int main()
{
    LL t, a, b, c, x1, y1, x2, y2;
    sf(t);
    for(LL cs = 1; cs<=t; cs++)
    {
        X.clear(), Y.clear();
        total = 0;
        sff(source.xx,source.yy);
        sff(dest.xx,dest.yy);
        add(source.xx,source.yy);
        add(dest.xx,dest.yy);
        sfff(a,b,c);
        for(LL i = 1; i<=a; i++)
        {
            sff(x1,y1);
            add(x1,y1);
            M[total++] = {x1,y1,x1,y1};
        }
        for(LL i = 1; i<=b; i++)
        {
            sff(x1,y1);
            sff(x2,y2);
            add(x1,y1);
            add(x2,y2);
            M[total++] = {x1,y1,x2,y2};
        }
        for(LL i = 1; i<=c; i++)
        {
            sff(x1,y1);
            sff(x2,y2);
            add(x1,y1);
            add(x2,y2);
            M[total++] = {x1,y1,x2,y2};
        }
        compress();
        block();
        LL res = dijkstra();
        printf("Case %lld: ",cs);
        if(res == -1)
            puts("Impossible");
        else
            printf("%lld\n",res);
    }
    return 0;
}
```

Plane Hopping (Spoj) :

```
/*
    Spoj Plane Hopping
    Find all pair shortest paths where at least k edges are used in every
path
    Use bellman ford equation(O(V*E*K)) with matrix expo(O(V^3*log2(K)))
    Modify Matrix multiplication with the bellman ford recurrence.
    Do V more multiplications to make sure you have got the best path.
*/

#include<bits/stdc++.h>
using namespace std;

#define D(x)    cerr << #x " = " << x << endl
#define mem(arr, x)     memset(arr, x, sizeof(arr))
typedef long long int LL;
struct mat{
    int r, c;
    LL arr[55][55];
    void print()
    {
        cout << endl << "Printing Matrix " << endl;
        for(int i = 0; i<r; i++)
        {
            for(int j = 0; j<c; j++)
                cout << arr[i][j] << " " ;
            cout << endl;
        }
    }
}M, A, B, id;
const LL inf = 1e19;
mat mul(mat a, mat b)
{
    mat ret;
    ret.r = a.r, ret.c = b.c;
    for(int i = 0; i<a.r; i++)
    {
        for(int j = 0; j<b.c; j++)
        {
            ret.arr[i][j] = inf;
            for(int k = 0; k<a.c; k++)
            {
                //ret.arr[i][j] += (a.arr[i][k] * b.arr[k][j])%MOD;
                ret.arr[i][j] = min(ret.arr[i][j], a.arr[i][k] +
b.arr[k][j]);
               // ret.arr[i][j] %= MOD;
            }
        }
    }
    return ret;
}
int n;
void init()
{
    id.r = id.c = n;
    for(int i = 0; i<id.r; i++)
    {
        for(int j = 0; j<id.c; j++)
            id.arr[i][j] = A.arr[i][j];
    }
    M.r = M.c = (n);

    A.r = n, A.c = n;
}

mat mat_exp(mat M, LL n)
```

```
{
    mat ret = id;
    while(n)
    {
        if(n&1) ret = mul(ret, M);
        n>>=1;
        M = mul(M,M);
    }
    return ret;
}
LL ans[55][55];
int main()
{
    int i, j, k, cs, t;
    scanf("%d",&t);
    for(cs = 1; cs<=t; cs++)
    {
        scanf("%d %d",&k,&n);
        for(i = 0; i<n; i++)
        {
            for(j = 0; j<n; j++)
                scanf("%d",&A.arr[i][j]), M.arr[i][j] = A.arr[i][j];
        }
        init();
        M = mat_exp(M, k-1);
        for(i = 0; i<n; i++)
        {
            for(j = 0; j<n; j++)
            {
                ans[i][j] = M.arr[i][j];
            }
        }

        for(k = 1; k<=n; k++)
        {
            M = mul(M, A);
            for(i = 0; i<n; i++)
            {
                for(j = 0; j<n; j++)
                {
                    ans[i][j] = min(ans[i][j], M.arr[i][j]);
                }
            }
        }
        printf("Region #%d:\n",cs);
        for(i = 0; i<n; i++)
        {
            for(j = 0; j<n; j++)
            {
                if(j)
                    printf(" ");
                printf("%lld",ans[i][j]);
            }
            puts("");
        }
        puts("");
    }

}
```

Powpow:

```
/*
    our task is to find a^(exp^(b)),
    a:Provided in input,10^5=>a>=0
    b:Provided in Input,10^5=>b>=0
    exp=(nC0)^2 + (nC1)^2 +(nC2)^2+..........+(nCn)^2,
    n:Provided in the input, 10^5=>n>=0
*/
int main()
{
    int i, k, cs, t;
    LL a, b, n;
    LL exp, MOD2, ans, j, MOD3;
    MOD2 = (MOD-1)/2;
    MOD3 = MOD-1;
    fact[0] = 1;
    FRE(i,1,200010)
        fact[i] = (fact[i-1] * i) % MOD2;
    sf(t);
    FRE(cs,1,t)
    {
        scanf("%lld %lld %lld",&a,&b,&n);
        /// find (a^(exp^b)) % MOD
        /// where exp = C(2n,n);
        /// C(a,b) means combination of b things out of a things.
        /// p = 1000000007 which is a prime.
        /// From Fermat's little theorem, (a^(p-1)) % p = 1 for any prime p.
        /// let's assume m = exp^b;
        /// m = c*(p-1) + (m%(p-1));
        /// j = (m%(p-1));
        /// so m = c*(p-1) + j;
        /// a^m = (a^(p-1))^c * a^j;
        /// (a^(p-1))^c = 1;(Fermat)
        /// so (a^m) % p = (a^j) % p;

        /// now for j, j = (exp^b)%(p-1);
        /// let MOD2 = (p-1)/2; [which is 500000003, a prime number]
        /// exp = C(2n,n) = 2*C(2n-1,n-1);
        /// we know, (2*a) % (2*b) = (2*(a%b))%(2*b);
        /// so j = (2*C(2n-1,n-1))^b % (p-1)
        /// j = ((2*(C(2n-1,n-1)% MOD2))^b)%(p-1);
        /// then find (a^j) % p;
        ans = fact[2*n-1];
        ans = (ans * ip((fact[n-1]*fact[n])%MOD2, MOD2-2, MOD2))%MOD2;
        ans = (2 * ans) % MOD3;
        ans = ip(ans,b,MOD3);
        ans = ip(a,ans,MOD);
        pf("%lld\n",ans);
    }
    return 0;
}
```

Centroid Decomposition Brute Force Approach:

Problem (CF 766E):

Mahmoud and Ehab live in a country with n<=1e5 cities numbered from 1 to n and connected by n - 1 undirected roads. It's guaranteed that you can reach any city from any other using these roads. Each city has a number ai attached to it.

We define the distance from city x to city y as the xor of numbers attached to the cities on the path from x to y (including both x and y). In other words if values attached to the cities on the path from x to y form an array p of length l then the distance between them is , where is bitwise xor operation.

Mahmoud and Ehab want to choose two cities and make a journey from one to another. The index of the start city is always less than or equal to the index of the finish city (they may start and finish in the same city and in this case the distance equals the number attached to that city). They can't determine the two cities so they try every city as a start and every city with greater index as a finish. They want to know the total distance between all pairs of cities.

Solution:

```
vector<int> E[MAX+10], tree[MAX+10];
int val[MAX+10], taken[MAX+10], dist[MAXLG+5][MAX+10], myLevel[MAX+10],
par[MAX+10];
pii mx[MAX+10];

int dfs1(int node, int par)
{
    int ret = 1;
    mx[node] = {0, node};
    for(auto v: E[node])
    {
        if(par == v || taken[v])
            continue;
        int nw = dfs1(v, node);
        mx[node] = max(mx[node], (pii){nw, v});
        ret += nw;
    }
    return ret;
}
int dfs2(int node, int nNode)
{
    if(mx[node].xx <= nNode/2)
        return node;
    return dfs2(mx[node].yy, nNode);
}
int getCentroid(int node)
{
    /// nNode = number of nodes in current tree
    int nNode = dfs1(node, 0);
    return dfs2(node, nNode);
```

```
}

void setDist(int node, int level, int d, int par)
{
    d ^= val[node];
    dist[level][node] = d;
    for(auto v: E[node])
    {
        if(taken[v] || v == par)
            continue;
        setDist(v, level, d, node);
    }
}

void build(int root, int L)
{
    setDist(root, L, 0, -1);
    myLevel[root] = L;
    for(auto v: E[root])
    {
        if(taken[v])
            continue;
        int nw = getCentroid(v);

        tree[root].pb(nw);
        par[nw] = root;
        taken[nw] = 1;
        build(nw, L+1);
    }
}
void centroidDecomposition(int &Root)
{
    mem(taken, 0);
    Root = getCentroid(1);
    taken[Root] = 1;
    build(Root, 1);
}

int getLCA(int u, int v)
{
    if(myLevel[u] < myLevel[v])
        swap(u,v);
    while(myLevel[u] != myLevel[v])
```

```
        u = par[u];
    while(u != v)
        u = par[u], v = par[v];
    return u;
}

int queryDist(int u, int v)
{
    int lca = getLCA(u,v);
    return dist[myLevel[lca]][u] + dist[myLevel[lca]][v];
}

LL ans, p2[25];
int cnt[23][2];

void query(int node, int P)
{
    int d = dist[myLevel[P]][node];
    d ^= val[P];
    for(int i = 0; i<=21; i++)
        ans += cnt[i][!checkBit(d,i)] * (p2[i]);
    for(auto v: tree[node])
        query(v, P);
}

void update(int node, int P)
{
    int d = dist[myLevel[P]][node];
    for(int i = 0; i<=21; i++)
        cnt[i][checkBit(d, i)]++;
    for(auto v: tree[node])
        update(v, P);
}

void solve(int node)
{
    mem(cnt, 0);
    for(int i = 0; i<=21; i++)
        cnt[i][checkBit(val[node], i)]++;
    for(auto v: tree[node])
    {
        query(v, node);
        update(v, node);
    }
    ans += val[node];
    for(auto v: tree[node])
        solve(v);
}

int main()
{
    p2[0] = 1;
    for(int i = 1; i<=20; i++)
        p2[i] = p2[i-1] * 2;
    int n;
    sf(n);
    for(int i = 1; i<=n; i++)
        sf(val[i]);
    int u, v;
    for(int i = 1; i<n; i++)
    {
        sff(u, v);
        E[u].pb(v);
        E[v].pb(u);
    }
    int Root;
    centroidDecomposition(Root);
    solve(Root);
    cout << ans << endl;
    return 0;
}
```

DSU on Tree:

Problem(CF 570D):
Roman planted a tree consisting of n vertices. Each vertex contains a lowercase English letter(26 in total). Vertex 1 is the root of the tree, each of the n - 1 remaining vertices has a parent in the tree. Vertex is connected with its parent by an edge. The parent of vertex i is vertex pi, the parent index is always less than the index of the vertex (i.e., pi < i).
The depth of the vertex is the number of nodes on the path from the root to v along the edges. In particular, the depth of the root is equal to 1.
We say that vertex u is in the subtree of vertex v, if we can get from u to v, moving from the vertex to the parent. In particular, vertex v is in its subtree.

Roma gives you m queries, the i-th of which consists of two numbers vi, hi. Let's consider the vertices in the subtree vi located at depth hi. Determine whether you can use the letters written at these vertices to make a string that is a palindrome. The letters that are written in the vertexes, can be rearranged in any order to make a palindrome, but all letters should be used.

Solution:

```
#define MAX         500000
int n, q, subtree[MAX+10];
vector<int>E[MAX+10];
int getSize(int node)
{
    subtree[node] = 1;
    for(auto v: E[node])
        subtree[node] += getSize(v);
    return subtree[node];
}
vector<pii>query[MAX+10];
struct data{
    bitset<26>parity;
    int odd;
}level[MAX+10];

char str[MAX+10];
int sltn[MAX+10];
void add(int node, int big, int depth)
{
    int my = str[node] - 'a';
    if(level[depth].parity[my])
        level[depth].parity[my] = 0;
    else
        level[depth].parity[my] = 1;
    if(level[depth].parity[my] == 1)
        level[depth].odd++;
    else
        level[depth].odd--;
    for(auto v: E[node])
    {
        if(v != big)
            add(v, -1, depth+1);
    }
}
void dfs(int node, bool keep, int depth)
```

```
{
    int bigChild = -1;
    for(auto v: E[node])
    {
        if(bigChild == -1 || subtree[bigChild] < subtree[v])
            bigChild = v;
    }
    for(auto v: E[node])
    {
        if(v != bigChild)
            dfs(v, 0, depth+1);
    }
    if(bigChild != -1)
        dfs(bigChild, 1, depth+1);
    add(node, bigChild, depth);
    for(auto q: query[node])
    {
        int h = q.xx, idx = q.yy;
        if(level[h].odd <= 1)
            sltn[idx] = true;
    }
    if(!keep)
        add(node, -1, depth);
}
int main()
{
    sff(n,q);
    int p;
    for(int i = 2; i<=n; i++)
    {
        sf(p);
        E[p].pb(i);
    }
    getSize(1);
    int v, h;
    scanf("%s",str+1);
    for(int i = 1; i<=q; i++)
    {
        sff(v,h);
        query[v].pb({h, i});
    }
    dfs(1,1,1);
    for(int i = 1; i<=q; i++)
    {
        if(sltn[i])
            puts("Yes");
        else
            puts("No");
    }
    return 0;
}
```

Fusion Principle (LOJ Game of CS):

Green Hackenbush with multiple edges between two nodes. Number of edges between two nodes stored in val[i][j]. n <= 1000, val[i][j] <= 1e9

```
vector<int>E[MAX+10], G[MAX+10];
int n, val[1010][1010];
void dfs(int node, int par)
{
    for(int i=  0; i<E[node].size(); i++)
    {
        int v = E[node][i];
        if(v != par)
        {
            dfs(v, node);
            if(val[node][v] != 1)
            {
                if(val[node][v] & 1)
                    G[node].pb(++n);
                for(int j = 0; j<G[v].size(); j++)
                {
                    int vv = G[v][j];
                    G[node].pb(vv);
                }
            }
            else
            {
                G[node].pb(v);
            }
        }
    }
}

int green(int node, int par = -1)
```

```
{
    int ret = 0;
    for(int i = 0; i<G[node].size(); i++)
    {
        int v = G[node][i];
        if(v != par)
            ret ^= (green(v, node)+1);
    }
    return ret;
}

int solve()
{
    dfs(1,-1);
    return green(1);
}
```

Link Cut Tree with Propagation:

Problem (IPSC 2009, Problem L: Let there be Rainbows):
Range query, range update on a tree. In a tree with n<=1e5 nodes, initially no edge is colored. There are 7 colors only. A query of (u,v,c) means that you have to color every edge between u and v who do not have the color c, with color c. If an edge is colored with a different color, it loses its previous color and gets repainted with current color c in the process. Finally report for each color, how many litres of that color did you need in total. It takes one litre to color any edge. Total query 1e5.

Solution:

```
struct Node
{
    int sz, label, lazy, myColor; /* size, label, flip for evert operation */
/// keep additional information here as you need
    LL sum[8];
    Node *p, *pp, *l, *r; /* parent, path-parent, left, right pointers in
splay tree*/
    Node()
    {
        p = pp = l = r = 0; /// initialize some of the custom variables here
if needed. ( like mx)
        lazy = myColor = 0;
        mem(sum, 0);
        sum[0] = 1;
    }
};

void normalize(Node *x)
{
    if(x->lazy)
    {
        if(x->l)
        {
            mem(x->l->sum, 0);
            x->l->lazy = x->lazy;
            x->l->myColor = x->lazy;
            x->l->sum[x->lazy]+=x->l->sz;
        }
        if(x->r)
        {
            mem(x->r->sum, 0);
            x->r->lazy = x->lazy;
            x->r->myColor = x->lazy;
            x->r->sum[x->lazy]+=x->r->sz;
        }
        x->lazy = 0;
    }
}



void update(Node *x) /// change here as you need.
{
    x->sz = 1;
    mem(x->sum, 0);
    x->sum[x->myColor] = 1;
    if(x->l)
    {
        x->sz += x->l->sz;
        for(int i = 0; i<=7; i++)
            x->sum[i] += x->l->sum[i];
    }
    if(x->r)
    {
        x->sz += x->r->sz;
        for(int i = 0; i<=7; i++)
            x->sum[i] += x->r->sum[i];
    }
}

void rotr(Node *x)
{
    Node *y, *z;
    y = x->p, z = y->p;
    normalize(y);
    normalize(x);
    if((y->l = x->r)) y->l->p = y;
    x->r = y, y->p = x;
    if((x->p = z))
    {
        if(y == z->l) z->l = x;
        else z->r = x;
    }
    x->pp = y->pp;
    y->pp = 0;
    update(y);
}

void rotl(Node *x)
{
    Node *y, *z;
    y = x->p, z = y->p;
    normalize(y);
    normalize(x);
    if((y->r = x->l)) y->r->p = y;
    x->l = y, y->p = x;
    if((x->p = z))
    {
        if(y == z->l) z->l = x;
        else z->r = x;
    }
    x->pp = y->pp;
    y->pp = 0;
    update(y);
}

void splay(Node *x)
{
    Node *y, *z;
    while(x->p)
    {
        y = x->p;
        if(y->p == 0)
        {
            if(x == y->l) rotr(x);
            else rotl(x);
        }
        else
        {
            z = y->p;
            if(y == z->l)
            {
                if(x == y->l) rotr(y), rotr(x);
                else rotl(x), rotr(x);
            }
            else
            {
                if(x == y->r) rotl(y), rotl(x);
                else rotr(x), rotl(x);
            }
        }
    }
    normalize(x);
    update(x);
}

Node *access(Node *x)
{
    splay(x);
    if(x->r)
    {
        x->r->pp = x;
        x->r->p = 0;
        x->r = 0;
        update(x);
    }

    Node *last = x;
    while(x->pp)
    {
        Node *y = x->pp;
        last = y;
        splay(y);
        if(y->r)
        {
            y->r->pp = y;
            y->r->p = 0;
        }
        y->r = x;
        x->p = y;
        x->pp = 0;
        update(y);
        splay(x);
    }
    return last;
}

//void makeRoot(Node *x)
//{
//    access(x);
//    x->flip ^= 1;
//    swap(x->l, x->r);
//}

int depth(Node *x)
{
    access(x);
    return x->sz - 1;
}

Node *root(Node *x)
{
    access(x);
    while(x->l) x = x->l;
    splay(x);
    return x;
}

void cut(Node *x)
{
    access(x);
    x->l->p = 0;
    x->l = 0;
    update(x);
}
```

```
void link(Node *x, Node *y)
{
//    makeRoot(x);
    access(x);
    access(y);
    x->l = y;
    y->p = x;
    update(x);
}

Node *lca(Node *x, Node *y)
{
    access(x);
    return access(y);
}

Node *getNext(Node *x, Node *y)
{
    access(y);
    splay(x);
    Node *ret = x->r;
    while(ret->l) ret = ret->l;
    return ret;
}

void up(Node *x, int c)
{
    x->myColor = c;
    x->lazy = c;
    mem(x->sum, 0);
    x->sum[c] = 1;
}



class LinkCut
{
public:
    Node *x;

    LinkCut(int n)
    {
        x = new Node[n+5];
        for(int i = 1; i <= n; i++)
        {
            x[i].label = i;
            update(&x[i]); /// initialize your custom variables here.
        }
    }

    virtual ~LinkCut()
    {
        delete[] x;
    }

    /// ***The link function makes u a child of v. It must be the case that u
is a root of a tree.
    ///    Also u and v must be in different trees.
    void link(int u, int v)
    {
        ::link(&x[u], &x[v]);
    }

    void cut(int u)
    {
        if(u == 1)return;
        ::cut(&x[u]);
    }

    int root(int u)
    {
        return ::root(&x[u])->label;
    }

    int depth(int u)
    {
        return ::depth(&x[u]);
    }

    int lca(int u, int v)
    {
        return ::lca(&x[u], &x[v])->label;
    }
    void access(int u)
    {
        ::access(&x[u]);
    }

    int getNext(int u, int v)
    {
        return ::getNext(&x[u],&x[v])->label;
    }

    void up(int v, int c)
    {
        ::up(&x[v], c);
    }

}*tree;

#define MAX     1000000
int parent[MAX+10], n;
vector<int>E[MAX+10];
int level[MAX+10];
void dfs(int node, int par, int depth)
{
    parent[node] = par;
    level[node] = depth;
    for(auto v: E[node])
    {
        if(v != par)
            dfs(v, node, depth+1);
    }
}

LL sltn[10];

LL go(int u, int v, int c, int &nxt)
{
    nxt = tree->getNext(u, v);
    tree->cut(nxt);
    tree->access(v);
    int total = level[v] - level[u];
    int ret = total - tree->x[v].sum[c];
    tree->link(nxt, u);
    return ret;
}
```

```
void update(int u, int v, int c)
{
    if(u == v)
        assert(false);
    int lca = tree->lca(u,v);
    LL ret;
    if(lca == v)
        swap(u,v);
    if(lca == u)
    {
        int nxt;
        ret = go(lca, v, c, nxt);
        tree->cut(nxt);
        tree->access(v);
        tree->up(v, c);
        tree->link(nxt, lca);
    }
    else
    {
        int nxt;
        ret = go(lca, u, c, nxt);
        tree->cut(nxt);
        tree->access(u);
        tree->up(u, c);
        tree->link(nxt, lca);

        ret += go(lca, v, c, nxt);
        tree->cut(nxt);
        tree->access(v);
        tree->up(v, c);
        tree->link(nxt, lca);
    }
    sltn[c] += ret;
}

char s[20];
string str;
map<string,int>color;
string bla[] = {"red", "orange", "yellow", "green", "blue", "indigo",
"violet"};

int main()
{
    for(int i = 1; i<=7; i++)
        color[bla[i-1]] = i;
    int t, u, v, q;
    sf(t);
    for(int cs= 1;cs<=t;cs++)
    {
        mem(sltn, 0);
        sf(n);
        for(int i = 1; i<n; i++)
        {
            sff(u,v);
            u++,v++;
            E[u].pb(v);
            E[v].pb(u);
        }
        tree = new LinkCut(n);
        dfs(1,-1,0);
        for(int i = 2; i<=n; i++)
            tree->link(i, parent[i]);
        sf(q);
        while(q--)
        {
            sff(u,v);
            u++,v++;
            scanf("%s",s);
            str = s;
            update(u,v,color[str]);
        }
        for(int i = 1; i<=7; i++)
            printf("%s %lld\n",bla[i-1].c_str(), sltn[i]);
        for(int i = 1; i<=n; i++)
            E[i].clear();
    }
    return 0;
}
```

Square Root Decomposition with buckets:

Problem (SPOJ SEGSQRSS):
Everything 1e5. The sum of squares over a range with range updates of 2 types:
1) increment in a range

2) set all numbers the same in a range.
Query:
Sum of squares over a range.

Solution:
```
#define MAX     100000
LL A[MAX+10];
LL bucketSize = sqrt(MAX);
struct bucketData{
    int st, ed;
    LL setLazy, addLazy, sqSum, sum;
    bool isSet;
}B[320];
void init(int n)
{
    int cnt = 0;
    B[0].st = B[0].addLazy = B[0].sqSum = B[0].sum = B[0].isSet = 0;
    int b = 0;
    for(int i = 0; i<n; i++, cnt++)
    {
        if(cnt == bucketSize)
        {
            B[b].ed = i-1;
            b++;
            B[b].st = i;
            B[b].addLazy = B[b].sqSum = B[b].sum = B[b].isSet = 0;
            cnt = 0;
        }
        B[b].sqSum += A[i] * A[i];
        B[b].sum += A[i];
    }
    B[b].ed = n-1;
}

void propagate(int b)
{
    for(int i = B[b].st; i<=B[b].ed; i++)
    {
        if(B[b].isSet)
            A[i] = B[b].setLazy;
        A[i] += B[b].addLazy;
    }
    B[b].isSet = 0;
    B[b].addLazy = 0;
}

void partialSetUp(int st, int ed, int b, LL x)
{
    propagate(b);
    for(int i = st; i<=ed; i++)
    {
        B[b].sqSum -= (A[i]*A[i]);
        B[b].sum -= A[i];
        A[i] = x;
        B[b].sqSum += (A[i]*A[i]);
        B[b].sum += A[i];
    }
}
void setUpdate(int st, int ed, LL x)
{
    int L = st/bucketSize, R = ed/bucketSize;
    int lim = min(ed, B[L].ed);
    partialSetUp(st, lim, L, x);
    if(L==R)
        return;

    for(int i = L+1; i<R; i++)
    {
        B[i].isSet = 1;
        B[i].addLazy = 0;
        B[i].setLazy = x;
        B[i].sqSum = (B[i].ed - B[i].st + 1) * x * x;
        B[i].sum = (B[i].ed - B[i].st + 1) * x ;
    }
    partialSetUp(B[R].st, ed, R, x);
}

void partialAddUp(int st, int ed, int b, LL x)
{
    propagate(b);
    for(int i = st; i<=ed; i++)
    {
        B[b].sqSum -= (A[i] * A[i]);
        B[b].sum -= A[i];
        A[i] += x;
        B[b].sqSum += (A[i] * A[i]);
        B[b].sum += A[i];
    }
}

void addUpdate(int st, int ed, LL x)
{
    int L = st/bucketSize, R = ed/bucketSize;
    int lim = min(ed, B[L].ed);
    partialAddUp(st, lim, L, x);
    if(L==R)
        return;

    for(int i = L+1; i<R; i++)
    {
        B[i].addLazy += x;
        B[i].sqSum += (B[i].ed - B[i].st + 1) * x * x + B[i].sum * 2 * x;
        B[i].sum += (B[i].ed - B[i].st + 1) * x ;
    }
    partialAddUp(B[R].st, ed, R, x);
}

LL query(int st, int ed)
{
    int L = st/bucketSize, R = ed/bucketSize;
    int lim = min(ed, B[L].ed);
    LL ret = 0;
    propagate(L);
    for(int i = st; i<=lim; i++)
        ret += (A[i] * A[i]);
    if(L == R)
        return ret;
    for(int i = L+1; i<R; i++)
        ret += B[i].sqSum;
    propagate(R);
    for(int i = B[R].st; i<=ed; i++)
        ret += (A[i] * A[i]);
    return ret;
}

int main()
{
    int n, t, q;
    sf(t);
    for(int cs = 1; cs<=t; cs++)
    {
        sff(n, q);
        for(int i = 0; i<n; i++)
            sl(A[i]);
        init(n);
        int type, st, ed, x;
        printf("Case %d:\n",cs);
        while(q--)
        {
            sfff(type, st, ed);
            st--, ed--;
            if(type == 0)
            {
                sf(x);
                setUpdate(st,ed,x);
            }
            else if(type == 1)
            {
                sf(x);
                addUpdate(st,ed,x);
            }
            else
                printf("%lld\n",query(st,ed));
        }
    }
    return 0;
}
```