DU_INCEPTION

-TANZIR

Table of Contents:

2D BIT:

```
// 1 based indexing
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;
}
// sum of the values enclosed by the rectangle x1,y1 and x2,y2 where x1,y1 is the
lower corner.
int query_rectangle(int x1, int y1, int x2, int y2)
{
    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:

```
/// 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));
}
```

2SAT:

```
/*
    1 based indexing
    2-SAT
    Finds if 2-SAT is possible and also prints the solution if possible.
    Solution to the problem LOJ - Forming the council
    Problem Description:
    In a city there are n voters, and m people formed the Govt. council. The
council members are numbered from 1 to
    m. Now everyone is complaining that the council is biased. So, they made a
plan. The plan is that the voters are
```

```
    given a chance to vote again to form the new council. A vote will be like ±i
±j. '+' means the voter wants that member
    to be in the council, '-' means the voter doesn't want the member to be in the
council. For example, there are 4 voters,
    they voted like
    +1 -3    the voter wants member 1 to be kept in the council or member 3 to be
thrown out
    +2 +3  the voter wants member 2 to be kept in the council or member 3 to be
kept in the council
    -1 -2     the voter wants member 1 to be thrown out or member 2 to be thrown
out
    -4 +1    the voter wants member 4 to be thrown out or member 1 to be kept in
the council
    A voter will be satisfied if at least one of his wishes becomes true. Now your
task is to form the council such that all
    the voters are happy.

    Input:
    Input starts with an integer T (≤ 20), denoting the number of test cases.
    Each case starts with a line containing two integers n (1 ≤ n ≤ 20000) and m (1
≤ m ≤ 8000). Each of the next n lines contains a vote in the form ±i ±j (1 ≤ i, j ≤
m).
    Output:
    For each case, print the case number and 'Yes' if a solution exists, or 'No' if
there is no solution. Then if the result is yes, print another line containing the
number of members in the council followed by the members in ascending order. And
print a single space between two numbers. There can be many solutions. Any valid
one will do.
*/
```

```
#define MAX         16010
int m, n, cnt;
/// total nodes = n
/// i is i, -i is n+i
int actual(int a)
{
    if(a < 0)
        return n+(-a);
    else
        return a;
}
/// sqnc holds sequence of the nodes obtained from the reversed graph
/// E holds the edges
/// component[i] holds the nodes in i'th component
/// scc holds the compressed graph description with each component as one node
```

```
vector<int> E[MAX+10], sqnc, REV[MAX+10], component[MAX+10], scc[MAX+10];
/// TIME holds every node of the components with their time after topologically
sorting them
/// xx means time, yy means the node.
/// The starting node has the largest time.
vector <pii> TIME;
/// state holds the final state of the nodes after satisfying everyone
bool state[MAX+10];
int vis[MAX+10], TM[MAX+10];
/// lead means the current leader under whom everyone who will go in the component
/// leader[i] signifies the leader of node i in the component
int  lead, leader[MAX+10];
/// Runs DFS on the reversed graph
void dfsRev(int node)
{
    if(vis[node])
        return;
    int i;
    vis[node]=1;

    FRL(i,0,REV[node].size())
        dfsRev(REV[node][i]);

    sqnc.pb(node);
    return;
}
/// dfs to create the components and the compressed graph
void dfs(int node)
{
    if(vis[node]==-1)
        return;
    if(vis[node])
    {
        if(leader[node]!=lead )
            scc[lead].pb(leader[node]);
        return;
    }
    int i;
    leader[node]=lead;
    component[lead].pb(node);
    vis[node]=-1;
    FRL(i,0,E[node].size())
        dfs(E[node][i]);

    vis[node]=1;
    return;
}
```

```
/// topologically sorts the components
int topo_sort(int node)
{
    if(vis[node]!=-1)
        return TM[node];
    int i,ret=0;
    for(i=0;i<scc[node].size();i++)
        ret=max(ret,topo_sort(scc[node][i]));
    TIME.pb({ret,node});
    vis[node] = 1;
    TM[node] = ret;
    return ret+1;
}
void SCC()
{
    /// find out the order in the reversed graph
    int i;
    sqnc.clear();
    mem(vis,0);
    FRE(i,1,2*n)
    {
        if(!vis[i])
            dfsRev(i);
    }
    /// run dfs from the end of sqnc
    mem(vis,0);
    lead=0;
    for(i=(2*n)-1;i>=0;i--)
    {
        if(!vis[sqnc[i]])
        {
            lead++;
            dfs(sqnc[i]);
        }
    }
}
bool two_Sat()
{
    int i, u, v, j;
    SCC();
    /// if leader of i and -i is the same, then they are in the same component
    /// 2-sat is impossible, return 0
    FRE(i,1,n)
    {
        if(leader[i]==leader[i+n])
            return 0;
    }
    /// topologically sort the components
    mem(vis,-1);
    TIME.clear();
    FRE(i,1,lead)
    {
        if(vis[i] == -1)
            topo_sort(i);
    }
    /// 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.
    sort(all(TIME));
    mem(state, 0);
    FRL(i,0,TIME.size())
    {
        v = TIME[i].yy;
        FRL(j,0,component[v].size())
        {
            u = component[v][j];
            if(( u>n && !state[u-n]) || ( u<=n && !state[u+n]))
            {
                if(u <= n)
                    cnt++;
                state[u]=1;
            }
        }
    }
    return 1;
}
void add(int u, int v)
{
    u = actual(u);
    v = actual(v);
    E[u].pb(v);
    REV[v].pb(u);
}
void _or(int u, int v)
{
    add(-u, v);
    add(-v, u);
}
int main()
{
    int cs, t, i, j, u, v;

    sf(t);
    FRE(cs,1,t)
    {
        cnt = 0;
        FRE(i,1,MAX)
        {
            E[i].clear();
            REV[i].clear();
            scc[i].clear();
            component[i].clear();
        }
        sff(m,n);

        FRE(i,1,m)
        {
            sff(u, v);
            _or(u,v);
        }
        printf("Case %d: ",cs);
        if(!two_Sat())
            printf("No\n");
        else
        {
            printf("Yes\n");
            printf("%d",cnt);
            FRE(i,1,n)
            {
                if(state[i])
                    printf(" %d",i);
            }
            printf("\n");
        }
    }
}
/*
    For building graphs in 2-SAT
    1.      1 x y means that either x or y should be present in the meeting.
    2.      2 x y means that if x is present, then no condition on y, but if x is
absent y should be absent
    3.      3 x y means that either x or y must be absent.
    4.      4 x y means that either x or y must be present but not both.
*/
    void buildGraph(int m)
    {
        int i, j, u, v;
        FRE(i,1,m)
        {
            sf(j);
            sff(u,v);
            if(j == 1)
            {
                E[n+u].pb(v);
                E[n+v].pb(u);
                REV[v].pb(n+u);
                REV[u].pb(n+v);
            }
            else if ( j == 2)
            {
                E[n+u].pb(n+v);
                E[v].pb(u);
                REV[n+v].pb(n+u);
                REV[u].pb(v);
            }
            else if( j == 3)
            {
                E[u].pb(n+v);
                E[v].pb(n+u);
                REV[n+v].pb(u);
                REV[n+u].pb(v);
            }
            else
            {
                E[u].pb(n+v);
                E[v].pb(n+u);
                E[n+u].pb(v);
                E[n+v].pb(u);
                REV[n+v].pb(u);
                REV[n+u].pb(v);
                REV[v].pb(n+u);
                REV[u].pb(n+v);
            }
        }
    }
```

Adaptive Simpson:

```
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, i), EPS);
}
```

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);
    }
}
```

Bellman Ford:

```
/// 1 based indexing
/// MAX is the highest number of nodes
/// returns true if there exists a negative cycle
/// otherwise returns false and calculates shortest path from src to all nodes
/// O(N*E)

/// famous bellman ford recurrence:
/// dist[v][k] = min(dist[v][k], dist[u][k-1] + cost of edge from u to v)
/// dist[v][k] is the minimum distance from source to v using exactly k edges.
/// where k is the number of edges used
/// so bellman ford can give shortest path from src to any node using (at most/ at
least ) k edges
/// O(K*E) for a particular source
/// O(V*K*E) for APSP
/// O(V^3*log2(K)) if matrix expo is used in place of bellman ford

vector<pii> E[MAX+10];
const int inf = 1.5e9;
int dist[MAX+10];
bool bellman_ford(int src, int n)
{
    for(int i = 1; i<=n; i++)
        dist[i] = inf;
    dist[src] = 0;
    for(int i = 1; i<=n; i++)
    {
        for(int u = 1; u<=n; u++)
        {
            for(int j = 0; j<E[u].size(); j++)
            {
                int v = E[u][j].xx;
                int c = E[u][j].yy;
                if(dist[v] > dist[u]+c)
                {
                    dist[v] = dist[u]+c;
                    if(i == n)
                        return true;
                }
            }
        }
    }
    return false;
}
```

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();
            }
        }

    }
}
```

BIT:

```
// 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));
}
```

Bridge:

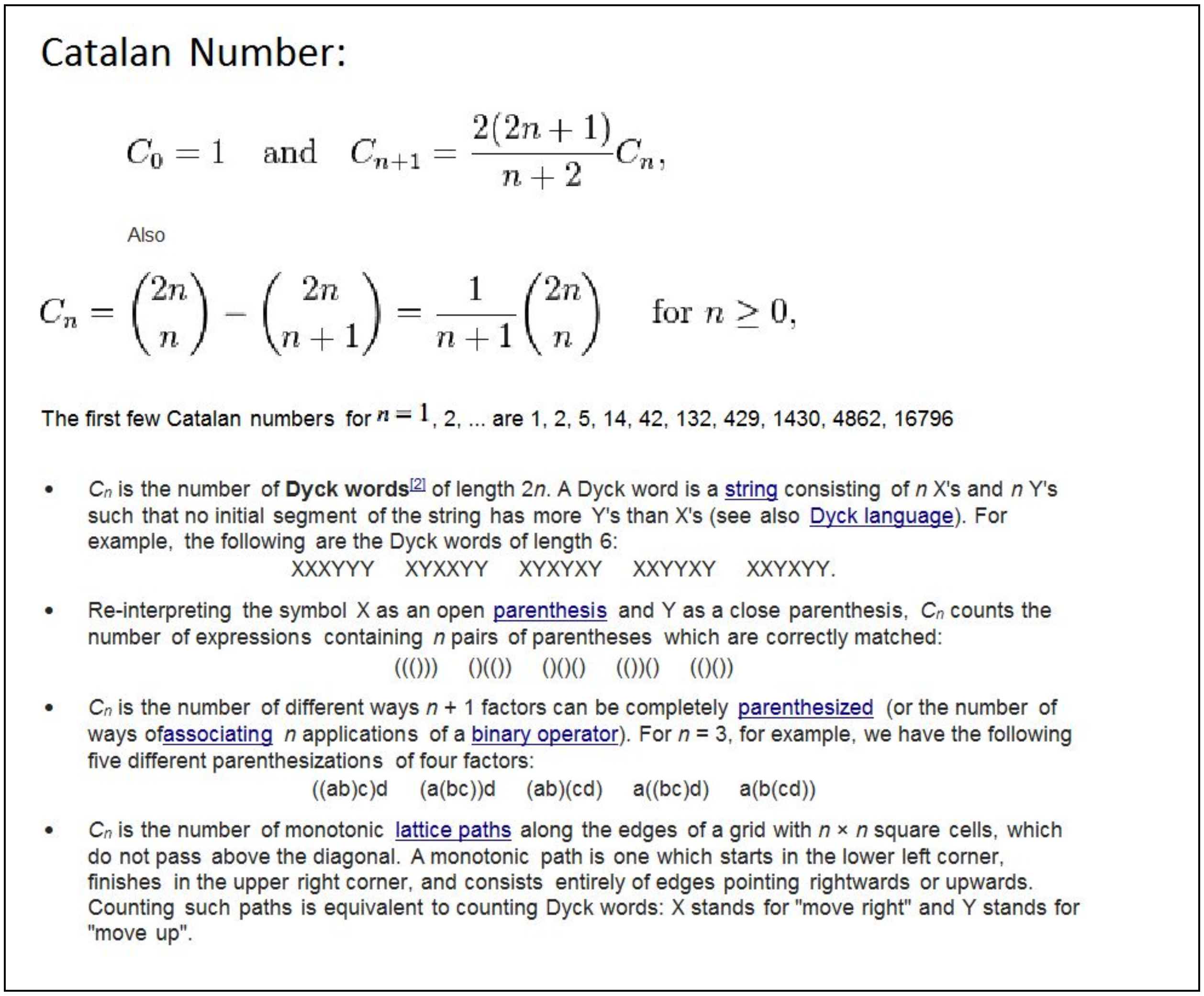
```
/// Bridge
/// n is the number of nodes
/// bridge will contain the bridge edges
/// 1 based indexing
vector<int> E[MAX+10];
vector <pii> bridge;
bool vis[MAX+10];
int parent[MAX+10], disc[MAX+10], low[MAX+10], n;
void dfs(int node)
{
    vis[node] = 1;
    if(parent[node] == -1)
        disc[node] = 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 trees
/// 1 based indexing
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 = 0; 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_Notes:



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,

Centroid Decomposition:

```
/*
    Centroid Decomposition tree.
    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.
*/
vector<int> E[MAX+10], tree[MAX+10];
int Root, taken[MAX+10];
pii mx[MAX+10];
int dfs1(int nd, int par)
{
    int ret = 1;
    mx[nd] = {0, nd};
    for(int i = 0; i<E[nd].size(); i++)
    {
        int v = E[nd][i];
        if(par == v || taken[v])
            continue;
        int nw = dfs1(v, nd);
        mx[nd] = max(mx[nd], (pii){nw, v});
        ret += nw;
    }
    return ret;
}
int dfs2(int nd, int n)
{
    if(mx[nd].xx <= n/2)
        return nd;
    return dfs2(mx[nd].yy, n);
}
int get_centroid(int nd)
{
    /// n = number of nodes in current tree
    int n = dfs1(nd, 0);
    return dfs2(nd, n);
}
void build(int cur_root)
{
    for(int i = 0; i<E[cur_root].size(); i++)
    {
        int v= E[cur_root][i];
        if(taken[v])
            continue;
        int nw = get_centroid(v);
        tree[cur_root].pb(nw);
        taken[nw] = 1;
        build(nw);
    }
}
void centroid_decomposition()
{
    /// Find Root
    Root = get_centroid(1);
    taken[Root] = 1;
    build(Root);
}
```

Circle Circle Intersection Points:

```
/// p1 and p2 are the points where the circles intersect.
/// Returns 0 if no intersection, inf if the circles are same, 1 if one, 2 if two
intersections

int circle_circle_intersection(circle A, circle B, point *p1, point *p2)
{
    double x0, y0, r0, x1, y1, r1;
    double xi, yi, xi_prime,yi_prime;
    x0 = A.center.x;
    y0 = A.center.y;
    r0 = A.r;

    x1 = B.center.x;
    y1 = B.center.y;
    r1 = B.r;

    if((fabs(x0 -x1) < eps && fabs(y0 - y1) < eps && fabs(r0 - r1) < eps) )
        return inf;

    double a, dx, dy, d, h, rx, ry;
    double x2, y2;

    dx = x1 - x0;
    dy = y1 - y0;

    d = hypot(dx,dy);
    if (d > (r0 + r1))  return 0;

    if (d < fabs(r0 - r1))  return 0;

    a = ((r0*r0) - (r1*r1) + (d*d)) / (2.0 * d) ;
    x2 = x0 + (dx * a/d);
    y2 = y0 + (dy * a/d);
    h = sqrt((r0*r0) - (a*a));
    rx = -dy * (h/d);
    ry = dx * (h/d);

    p1->x = x2 + rx;
    p2->x = x2 - rx;
    p1->y = y2 + ry;
    p2->y = y2 - ry;

    if(fabs(p1->x - p2->x) < eps && fabs(p1->y - p2->y) < eps)
        return 1;
    return 2;
}
```

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;
}
```

Convex Hull:

```
//Convex-Hull (O(nlogn))
LL triArea2(pii a, pii b, pii c) // includes sign.. +ve -> ccw, -ve -> cw;
{
    LL ret = 0;
    ret += (LL) a.xx*b.yy + (LL) b.xx*c.yy + (LL) c.xx*a.yy - (LL) a.xx*c.yy - (LL)
c.xx*b.yy - (LL) b.xx*a.yy;
    return ret;
}
bool ccw(pii a, pii b, pii c) {return triArea2(a,b,c) >= 0;}
bool cw(pii a, pii b, pii c)  {return triArea2(a,b,c) <= 0;}

void Convex_hull(vector<pii> &P, vector<pii> &hull) // Returns the points in acw
order using minimum number of points
{
    hull.clear();
    vector< pair <int, int> > up, dwn;
    pair <int, int> L, R;

    int i, len = P.size();
    sort(P.begin(), P.end());

    up.push_back(P[0]), dwn.push_back(P.back());
    L = P[0]; R = P.back();

    for(i = 1; i <= len-1; i++) {
        if(i != len-1 && ccw(L, P[i], R)) continue;
        else while(up.size() >= 2 && ccw(up[up.size()-2], up.back(), P[i]))
up.pop_back();
        up.push_back(P[i]);
    }

    for(i = len - 2; i >= 0; i--) {
        if(i && cw(L, P[i], R)) continue;
        else while(dwn.size() >= 2 && ccw(dwn[dwn.size()-2], dwn.back(), P[i]))
dwn.pop_back();
        dwn.push_back(P[i]);
    }

    up.pop_back(); dwn.pop_back();
    hull = up;
    for(i = 0; i < dwn.size();i++) hull.push_back(dwn[i]);

    reverse(hull.begin(), hull.end());
}
```

Convex Hull(Angular Sort):

```
Convex Hull (Graham Scan) O(nlogn):
// compare Function for qsort in convex hull
point Firstpoint;
int cmp(const void *a,const void *b) {
      double x,y;
      point aa,bb;
      aa = *(point *)a;
      bb = *(point *)b;
      x = isleft( Firstpoint, aa, bb );
      if( x > eps ) return -1;
      else if( x < -eps ) return 1;
      x = sq_Distance( Firstpoint, aa );
      y = sq_Distance( Firstpoint, bb );
      if( x + eps < y ) return -1;
      return 1;
}
// 'P' contains all the points, 'C' contains the convex hull
// 'nP' = total points of 'P', 'nC' = total points of 'C'
void ConvexHull( point P[], point C[], int &nP, int &nC ) {
      int i, j, pos = 0; // Remove duplicate points if necesary
      for( i = 1; i < nP; i++ )
            if( P[i].y < P[pos].y || ( eq( P[i].y, P[pos].y ) && P[i].x > P[pos].x
+ eps ) )
```

```
                    pos = i;
        swap( P[pos], P[0] );
        Firstpoint = P[0];
        qsort( P + 1, nP - 1, sizeof( point ), cmp );
        C[0] = P[0]; C[1] = P[1];
        i = 2, j = 1;
        while( i < nP ) {
                if( isleft( C[j-1], C[j], P[i] ) > -eps ) C[++j] = P[i++];
                else j--;
        }
        nC = j + 1;
}
```

Convex Hull Trick DP Optimization:

```
struct line {
    LL m, c;
    double intersect(const line p) const {
        LL a = p.c-c, b = m-p.m;
        return (long double)a / (long double)b;
//        LL g = __gcd(a, b);
//        return { a/g, b/g };
    }
    inline LL getY(LL x) {
        return m*x + c;
    }
};
namespace cht {
    line arr[MAX];
    int key;
    bool minFlag;

    void init(bool f) {
        key = 0;
        minFlag = f;
    }

    inline bool check(line a, line b, line c) {
        auto ac = a.intersect(c);
        auto ab = a.intersect(b);
        return (ac > ab);
//        return (1.0 * ac.uu * ab.vv > 1.0 * ac.vv * ab.uu);
    }

    inline void add(line l) {
        if(key == 0) arr[++key] = l;
        else if(arr[key].m == l.m) {
            if(minFlag) {
                if(arr[key].c < l.c) return;
                else arr[key] = l;
            }
            else {
                if(arr[key].c > l.c) return;
                else arr[key] = l;
            }
        }


        else {
            while(key >= 2) {
                if(!check(arr[key-1], arr[key], l)) key--;
                else break;
            }
            arr[++key] = l;
        }
    }
    inline bool onSegment(int idx, LL x) {
        if(idx != 1)  {
            auto p = arr[idx].intersect(arr[idx-1]);
            if(p > x) return false;
//            if(1.0 * p.uu > 1.0 * p.vv*x) return false;
        }
        if(idx != key) {
            auto p = arr[idx].intersect(arr[idx+1]);
            if(p < x) return false;
//            if(1.0 * p.uu < 1.0 * p.vv*x) return false;
        }
        return true;
    }
}

LL solve() {
    LL ret = LLONG_MAX;
    cht::init(true);
    cht::add({ height[n-1], 0 });
    int last = cht::key;
    for(int idx=n-2; idx>=0; idx--) {
        // querying
        last = min(last, cht::key);
        while(last <= cht::key && !cht::onSegment(last, cost[idx])) last++;
        ret = cht::arr[last].getY(cost[idx]);
        // inserting
        cht::add({ height[idx], ret });
    }
    return ret;
}
```

Derangement:

```cpp
// The number of ways to permute n balls so that none of them have its initial
position
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

```cpp
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 Notes:

```
  1.Min Priority queue
    priority_queue<int, vector<int>, greater <int> > q;

  2.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
```

Dinitz MaxFlow:

```
/// 1 based indexing
/// MAXE = twice the number of edges
/// for directed edge, cap[nEdge] = 0
const int MAXN = ?, MAXE = ?; // MAXE = twice the number of edges
int src, snk, nNode, nEdge;
int Q[MAXN], fin[MAXN], pro[MAXN], dist[MAXN];
int flow[MAXE], cap[MAXE], nxt[MAXE], to[MAXE];
const int inf = ?;
inline void init(int _src, int _snk, int _n)
{
    src = _src, snk = _snk, nNode = _n, nEdge = 0;
    mem(fin, -1);
}
inline void add(int u, int v, int _cap)
```

```
{
    to[nEdge] = v, cap[nEdge] = _cap, flow[nEdge] = 0, nxt[nEdge] = fin[u], fin[u]
= nEdge++;
    to[nEdge] = u, cap[nEdge] = _cap, flow[nEdge] = 0, nxt[nEdge] = fin[v], fin[v]
= nEdge++; // for directed cap[nEdge]=0 here
}
bool bfs()
{
    int st, en, i, u, v;
    mem(dist, -1);
    dist[src] = st = en = 0;
    Q[en++] = src;
    while(st < en)
    {
        u = Q[st++];
        for(i=fin[u]; i>=0; i=nxt[i])
        {
            v = to[i];
            if(flow[i] < cap[i] && dist[v]==-1)
            {
                dist[v] = dist[u]+1;
                if(v == snk)
                    return true;
                Q[en++] = v;
            }
        }
    }
    return dist[snk]!=-1;
}
int dfs(int u, int fl)
{
    if(u==snk) return fl;
    for(int &e=pro[u], v, df; e>=0; e=nxt[e])
    {
        v = to[e];
        if(flow[e] < cap[e] && dist[v]==dist[u]+1)
        {
            df = dfs(v, min(cap[e]-flow[e], fl));
            if(df>0)
            {
                flow[e] += df;
                flow[e^1] -= df;
                return df;
            }
        }
    }
    return 0;
}
int dinitz()   // 1-based indexing
{
    int ret = 0;
    int df;
    while(bfs())
    {
        for(int i=1; i<=nNode; i++) pro[i] = fin[i];
        while(true)
        {
            df = dfs(src, inf);
            if(df) ret += df;
            else break;
        }
    }
    return ret;
}
```

Directed Minimum Spanning Tree:

```
/*
*     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 );
}
```

Divide & Conquer DP Optimization:

```
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];
}
```

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]);
    }
}
```

<u>**Euclidean Formulas:**</u>

```
///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 Construction:

```
/*
    Euler path = path that visits all the nodes 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;
}
```

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 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')
    {
        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);
}
```

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--]);
}
```

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;
}
```

Harmonic Numbers:

```
/// 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;
}
```

Heavy Light Decomposition:

```
/// 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
#define MAXN ?
int chainNo = 0, chainHead[MAXN+10], chainSize[MAXN+10], chainId[MAXN+10],
idxInBaseArray[MAXN+10],
    baseArray[MAXN+10], cnt = 0, subtreeSize[MAXN+10];

/// HLD start
void HLD(int node, int par)
{
    cnt++;
    idxInBaseArray[node] = cnt;
    baseArray[cnt] = val[node];
    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);
    initBIT(n);
    for(int i = 1; i<=n; i++)
        updateBIT(i, baseArray[i]);
}
/// HLD End
inline int solve(int u, int v)
{
    /// v is below u, level[v] > level[u];
    int ret = 0;
    while(true)
    {
        if(chainId[u] == chainId[v])
        {
            int L = idxInBaseArray[u];
            int R = idxInBaseArray[v];
            ret += range_query(L,R);
            return ret;
        }
        int head = chainHead[chainId[v]];
        int L = idxInBaseArray[head];
        int R = idxInBaseArray[v];
        ret += range_query(L,R);
        v = parent[head];
    }
}
inline int query(int u, int v)
{
    int lca = findLCA(u,v);
    int ret = 0;
    return solve(lca,u) + solve(lca, v) - baseArray[idxInBaseArray[lca]];
}
inline void update(int a, int v)
{
    int idx = idxInBaseArray[a];
    int diff = v - baseArray[idx];
    baseArray[idx] = v;
    updateBIT(idx,diff);
}
```

IDA*:

```
/*
    IDAStar to solve 15 puzzle
    Finds the lexicographical minimum steps to reach solution.
    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()
{
    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;
}
```

Josephus Recurrence:

```
//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;
}
```

Knuth Morris Pratt:

```
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;
}
```

Knuth Optimization

```
/// 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;
}
```

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;
}
```

Kuhn's Bipartite Matching:

```
/// Kuhn's BPM
/// O(V*E)
/// n is the number of nodes on the left side
/// 1 based indexing
int A[MAX+10], B[MAX+10], Left[MAX+10], Right[MAX+10], n, m;
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;
}
```

Longest Increasing Subsequence (Nlogn):

```
/// 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:

```
/// Run a dfs and fill up level and parent arrays first.
/// 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);
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];
}
```

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 Important Notes:

```
 1. 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.

 2. 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

 3. 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.

 4. 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.

 5. 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.

 6. 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.

 7. 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.

  8.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.
```

Mincost Maxflow (SPFA):

```
/// 0 based indexing
namespace mcmf{
    const int MAXN = ?, inf = ?;
    int src, snk, n, 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, n = _n;
        edges.clear();
        for(int i = 0; i<n; 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 = 0; i<n; 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> MCMF()
    {
        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};
    }
}
```

Minimum Enclosing Sphere:

```
//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));
}
```

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;
    }
}
```

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 Rectangle Intersection:

```
struct rectangle{
    point LB, RT;
    rectangle(){};
    rectangle(point a, point b)
    {
        LB = a;
        RT = b;
    }
    int area()
    {
        return (RT.x - LB.x) * (RT.y - LB.y);
    }
};
int rect_rect_intersection(rectangle Q, rectangle W)
{
    point LB, RT;
    LB.x = max(Q.LB.x,W.LB.x);
    LB.y = max(Q.LB.y,W.LB.y);
    RT.x = min(Q.RT.x,W.RT.x);
    RT.y = min(Q.RT.y,W.RT.y);

    if(LB.x >= RT.x || LB.y >= RT.y)
        return 0;
    return (RT.x - LB.x) * (RT.y - LB.y);
}
```

Rectangle Union:

```
struct point{
    LL x, y1, y2;
    bool st;
}A[MAX*2+10];
```

```
struct node{
    LL on, val;
}tree[MAX*32+10];
map<int,int> M;
vector<int> v;
LL ulta[MAX*4+10];
void update(int nd, int st, int ed, int i, int j, int v)
{
    if(st == ed)
        return;
    if(st > j || ed < i)
        return;
    int mid = (st+ed+1)/2, lft = 2*nd, rght = lft+1;
    if(i <= st && j >= ed)
    {
        tree[nd].val+=v;
        if(tree[nd].val == 0)
            tree[nd].on = tree[lft].on + tree[rght].on;
        else
            tree[nd].on = (ulta[ed]-ulta[st]);
        return;
    }
    if(ed -st == 1) return;
    update(lft, st, mid, i, j, v);
    update(rght, mid, ed, i, j, v);
    if(!tree[nd].val)
        tree[nd].on = tree[lft].on + tree[rght].on;
}
LL query()
{
    return tree[1].on;
}
bool operator < (point a, point b)
{
    return a.x < b.x;
}
int main()
{
    int cs, t, i, j, k, n;
    LL x1, y1, x2, y2;
    sf(t);
    FRE(cs,1,t)
    {
        mem(tree,0);
        M.clear();
        v.clear();
        sf(n);
        int mx = 1;
        FRE(i,1,n)
        {
            sll(x1,y1);
            sll(x2,y2);
            v.pb(x1);
            v.pb(x2);
            v.pb(y1);
            v.pb(y2);
            A[i*2-1].st = 1;
            A[i*2-1].x = x1;
            A[i*2-1].y1 = y1;
            A[i*2-1].y2 = y2;

            A[i*2].st = 0;
            A[i*2].x = x2;
            A[i*2].y1 = y1;
            A[i*2].y2 = y2;
        }
        sort(all(v));
        un(v);
        for(i = 0; i<v.sz; i++)
            M[v[i]] = i+1, ulta[i+1] = v[i];
        sort(A+1,A+2*n+1);
        LL nw = 0;
        LL total = 0;
        mx = v.sz;
        for(i = 1; i<=2*n; i++)
        {
            total += (A[i].x - nw) * (query());
            nw = A[i].x;
            if(A[i].st == 1)
                update(1,1,mx,M[A[i].y1],M[A[i].y2],1);
            else
                update(1,1,mx,M[A[i].y1],M[A[i].y2],-1);
        }
        pf("Case %d: %lld\n",cs,total);
    }
    return 0;
}
```

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]);
}
```

Russian Peasant Multiplication with NFSSDQ's Addition:

```
Check Rabin MIller
```

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;
}
```

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;
}
```

Sterling Numbers of the first and the second kind:

```
/// 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;
}
```

SPOJ Plane Hopping:

```
/*
    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("");
    }

}
```

SPOJ 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);
    }
}
```

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;
}
```

Timetracker:

```
int main()
{
    clock_t start, ed;
    start = clock();
    /// code
    /// code end
    ed = clock();
    double time_elapsed = (double)(ed - start) / CLOCKS_PER_SEC;
    fprintf(stderr, "%.3f s\n", time_elapsed);
    return 0;
}
```

Tetrahedron AKA Pyramid Formulas:

```
/*
Some tetrahedron formulas
*/
/// Code: Zobayer Vai
inline double volume(double u, double v, double w, double U, double V, double W) {
    double u1,v1,w1;
    u1 = v * v + w * w - U * U;
    v1 = w * w + u * u - V * V;
    w1 = u * u + v * v - W * W;
    return sqrt(4.0*u*u*v*v*w*w - u*u*u1*u1 - v*v*v1*v1 - w*w*w1*w1 + u1*v1*w1) / 
12.0;
}

inline double surface(double a, double b, double c) {
    return sqrt((a + b + c) * (-a + b + c) * (a - b + c) * (a + b - c)) / 4.0;
}

inline double insphere(double WX, double WY, double WZ, double XY, double XZ, 
double YZ) {
    double sur, rad;
    sur = surface(WX, WY, XY) + surface(WX, XZ, WZ) + surface(WY, YZ, WZ) + 
surface(XY, XZ, YZ);
    rad = volume(WX, WY, WZ, YZ, XZ, XY) * 3.0 / sur;
    return rad;
}
```

Trie (Dynamic):

```
/*
    Dynamic memory allocation.
    Will need O(total number of characters in all the input strings) memory at 
most.
    Find returns true if a string has been previously inserted into the trie.
*/
```

```
struct node{
    bool end_mark;
    node *nxt[26];
    node()
    {
        end_mark = false;
        for(int i = 0; i<26; i++)
            nxt[i] = NULL;
    }
}*root;
void del(node *cur)
{
    for(int i = 0; i<26; i++)
    {
        if(cur->nxt[i] != NULL)
            del(cur->nxt[i]);
    }
    delete(cur);
}
void init()
{
    if(root != NULL)
        del(root);
    root = new node();
}
void add(string s)
{
    node *current = root;
    for(int i = 0; i<s.size(); i++)
    {
        int nw = s[i] - 'a';
        if(current->next[nw] == NULL)
            current->next[nw] = new node();
        current = current->next[nw];
    }
    current->end_mark = true;
}
bool Find(string s)
{
    node *current = root;
    for(int i = 0; i<s.size(); i++)
    {
        int nw = s[i] - 'a';
        if(current->next[nw] == NULL)
            return 0;
        current = current->next[nw];
    }
    return current->end_mark;
}
```

Trie (Static):

```
/*
    Static memory allocation.
    Will need O(total number of characters in all the input strings * 26) memory at
most.
    Find returns true if a string has been previously inserted into the trie.
*/
#define MAX     total number of characters in all the input strings ?
int total;
struct node{
    bool end_mark;
    int nxt[26];
}trie[MAX+10];
void init()
{
    total = 0;
    mem(trie,0);
}

void add(char s[])
{
    int len = strlen(s);
    int cur = 0;
    for(int i = 0; i<len; i++)
    {
        int nw = s[i] - 'a';
        if(trie[cur].nxt[nw] == 0)
            trie[cur].nxt[nw] = ++total;
        cur = trie[cur].nxt[nw];
    }
    trie[cur].end_mark = true;
}

bool Find(char s[])
{
    int len = strlen(s);
    int cur = 0;
    for(int i = 0; i<len; i++)
    {
        int nw = s[i] - 'a';
        if(trie[cur].nxt[nw] == 0)
            return false;
        cur = trie[cur].nxt[nw];
    }
    return trie[cur].end_mark;
}
```

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--;
            }
        }
    }
}
```

Template:

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

#define FRE(i,a,b)  for(i = a; i <= b; i++)
#define FRL(i,a,b)  for(i = a; i < b; i++)
#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 << #x " = " << (x) << '\n'
#define DBG          cerr << "Hi" << '\n'
#define pb           push_back
#define PI           acos(-1.00)
#define xx           first
#define yy           second
#define eps          1e-9

typedef long long int LL;
typedef double db;
typedef pair<int,int> pii;
typedef pair<LL,LL> pll;

int main()
{
    //freopen("in.txt","r",stdin);
    //freopen("out.txt","w",stdout);
    int i, j, cs, t;
    return 0;
}
```