Team Zero00 (gtd)

Graph:

Task 1: (Dominos 2)

Given a set of dominos that are knocked down by hand, your task is to determine the total number of dominos that fall.

Input:

here 3 inputs, n, m, l;

n → number of dominos (nodes)

m→ number of edges (connected to each other) /// undirected.

l → dominos knocked down by hand.

Output:

number of dominos that fall.

code:

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
#define endl "\n"
```

```
#define test   \
    int tc;    \
    cin >> tc; \
    while (tc--)

vector<int> adj[123456];
bool vis[123456] = {0};
set<int> fall;

void dfs(int source)
{
    vis[source] = 1;
    fall.insert(source);

    for (auto e : adj[source])
    {
        if (vis[e] == 0)
        {
            dfs(e);
        }
    }
    return;
}

int main()
{

    test
    {
        int n, m, l; // n->nodes,m->edges , l is those nodes which is knocked by hand to fall.
        cin >> n >> m >> l;

        // thing about the dominos , they are connected in order to fall.

        for (int i = 0; i <= n; i++)
        {
            vis[i]= 0;
            adj[i].clear();
        }

        for (int i = 0; i < m; i++)
        {
            int u, v;
            cin >> u >> v;
            adj[u].push_back(v); // directed.
        }

        for (int i = 0; i < l; i++)
        {
            int x;
            cin >> x;
            if (vis[x] == 0)
            {
                dfs(x);
            }
        }

        cout << fall.size() << endl;
        fall.clear();

    }
```

```
    return 0;
}
```

Task 2(Rumor-Cf):

Statement:

A quest in which Vova must spread a rumor to all characters in a settlement named Overcity, with each character having a cost for spreading the rumor. The goal is to determine the minimum amount of gold Vova needs to spend to complete the quest.

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

vector<int> adj[123456];
bool vis[123456] = {false};
vector<int> friends;
void dfs(int source)
{
    vis[source] = 1;
    friends.push_back(source);
    for(auto e: adj[source]){
        if(vis[e]==0){
            dfs(e);
        }
    }
}
int main()
{
    int n, m; // n->nodes,m->edges
    cin >> n >> m;
    ll cost[n + 1]={}; // path cost.
    for (int i = 1; i <= n; i++)
    {
        cin >> cost[i];
    }
    for (int i = 0; i < m; i++)
    {
        int u, v;
        cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    ll ans = 0;
    for (int i = 1; i <= n; i++)
    {
        if (vis[i] == 0)
        {
            friends.clear();
            dfs(i);
            ll mini = INT_MAX;
            for (auto e : friends)
            {
                mini = min(mini, cost[e]);
```

```
            }
            ans += mini;
            mini=INT_MAX;
            friends.clear();
        }
    }
    cout << ans << endl;

    return 0;
}
```

Task 3: (Count the Number of Complete Components)

Intuition

In a connected graph or sub-graph(component), `|E|=v*(v-1)/2`

Approach

Use DFS in the same way as to find connected components. Additionally, during each DFS pass for a component check if in that component sum of all the edges taken twice is equal to n*(n-1) or not i.e., `|E|*2==v*(v-1)`

Complexity

- Time complexity: `O(n+E`)
- Space complexity: `O(n+E)`

```
class Solution
{
public:
    void dfs(vector<vector<int>> &adj, int i, vector<int> &vis, int &node, int &ce)
    {
        vis[i] = 1;
        node++;
        ce += adj[i].size(); // undirect graph howay overlap korbe, jeta hobe din sheshe 2 gun.
        for (auto e : adj[i])
        {
            if (!vis[e])
            {
                dfs(adj, e, vis, node, ce);
            }
        }
        return;
    }
    int countCompleteComponents(int n, vector<vector<int>> &edges)
    {
        vector<vector<int>> adj(n, vector<int>());
        for (auto edge : edges)
        {
            int u = edge[0], v = edge[1];
            adj[u].push_back(v);
            adj[v].push_back(u);
        }
        vector<int> vis(n, 0);
        int ccc = 0; // Complete connected Components
        for (int i = 0; i < n; i++)
        {
```

```
            if (!vis[i])
            {
                int node = 0, ce = 0; // connected edge;
                dfs(adj, i, vis, node, ce);
                //  ce=(node*node-1)/2;
                // 2 bar kore jog hocche, modified dfs function e.
                if (ce == (node * (node - 1)))
                {
                    ccc++;
                }
            }
        }
        return ccc;
    }
};
```

DP:

UVA weeding shopping:

Problem statement:

Given a money M (<= 200) and a list of garments C (<= 20). Each garment has K (<= 20) models. You want to buy one model for each garment, and you want to spend your money as much as possible.

Explanation:

This is a typical Dynamic Programming problem. Let **dp(m,c)** be the maximum money you can spend when you have **m** money, and you want to buy one model for each garment from garment 0 to garment **c-1**. If it's not possible to buy one model for each garment then the value of **dp(m,c)** is **-2** (or any other special value).

Then **dp(m,c)** can be computed recursively:

```
dp(m,c) =        -2,   if m is negative
                  c,   if c is zero
      max(spending),   otherwise
```

Where **spending** is the maximum value of **dp(m-ci, c-1) + ci** for all the prices of model **ci** of garment **c-1** and **dp(m-ci, c-1)** is not -2.

The total complexity is O(M*C*max(K)) which is around 200*20*20 = 80,000 operations per test case.

code:

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

```
int money, numGarments, numModels, price[25][25]; // price[garment_id (<= 20)][model (<= 20)]
int memo[210][25];          // dp table memo[money (<= 200)][garment_id (<= 20)]

int shop(int remainingMoney, int garment_id)
{
    if (remainingMoney < 0)
        return -1 * INT_MAX; // fail, return a large negative number (1B)

    if (garment_id == numGarments)  // we have bought the last garment
        return money - remainingMoney; // done, return this value

    if (memo[remainingMoney][garment_id] != -1)  // if this state has been visited before
        return memo[remainingMoney][garment_id]; // simply return it

    int maxSpentMoney = -1 * INT_MAX;

    for (int model = 1; model <= price[garment_id][0]; model++) // try all possible models
        maxSpentMoney = max(maxSpentMoney, shop(remainingMoney - price[garment_id][model], garment_id + 1));

    return memo[remainingMoney][garment_id] = maxSpentMoney; // assign maxSpentMoney to dp table + return it!
}

#define test   \
    int tc;    \
    cin >> tc; \
    while (tc--)

int main()
{
    test
    {
        cin >> money >> numGarments;

        for (int i = 0; i < numGarments; i++)
        {
            cin >> numModels;

            price[i][0] = numModels;

            for (int j = 1; j <= numModels; j++)
                cin >> price[i][j];
        }
        memset(memo, -1, sizeof memo); // initialize DP memo table

        ll maxSpentMoney = shop(money, 0); // start the top-down DP

        if (maxSpentMoney < 0)
            cout << "no solution" << endl;
        else
            cout << maxSpentMoney << endl;
    }
    return 0;
}
```

Leetcode:
Longest Common Subsequence

Explanation:

Basic LCS.

Code: (recursion + memorization may cause TLE)

```
class Solution {
public:
    int longestCommonSubsequence(string text1, string text2) {
        int n=text1.length();
        int m=text2.length();
        int dp[n+1][m+1];
        for(int i=0;i<=n;i++){
            for(int j=0;j<=m;j++){
                if(i==0 || j==0){
                    dp[i][j]=0;
                }

                else if(text1[i-1]==text2[j-1]){
                    dp[i][j]=1+dp[i-1][j-1];
                }

                else{dp[i][j]=max(dp[i-1][j],dp[i][j-1]);}
            }
        }
        return dp[n][m];


    }
};
```

Tree

UVA Binary Search tree:

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

void postOrder(int pre[], int n, int minval, int maxval, int &preIndex)
{

    if (preIndex == n)
        return;

    if (pre[preIndex] < minval || pre[preIndex] > maxval)
    {
        return;
    }

    int val = pre[preIndex];
    preIndex++;
    postOrder(pre, n, minval, val, preIndex);
    postOrder(pre, n, val, maxval, preIndex);
    cout << val << endl;
}

int main()
```

```
{
    int pre[123456];

    int val;
    scanf("%d", &val);

    int n=0;
    pre[n] = val;
    while (scanf("%d", &val) == 1)
    {

        n++;
        pre[n] = val;
    }



    int preIndex = 0;

    postOrder(pre, n+1, INT_MIN, INT_MAX, preIndex);

    return 0;
}
```

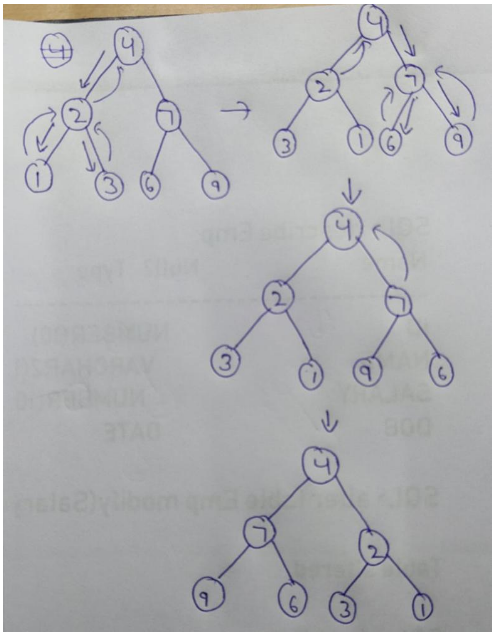
Leetcode (Invert Binary Tree):

Intuition

In this question we have to **Invert the binary tree**.

So we use **Post Order Treversal** in which first we go in **Left subtree** and then in **Right subtree** then we return back to **Parent node**.

When we come back to the parent node we **swap** it's **Left subtree** and **Right subtree**.



```
class Solution {
public:
    TreeNode* invertTree(TreeNode* root) {
        // Base Case
        if(root==NULL)
            return NULL;
        invertTree(root->left); //Call the left substree
        invertTree(root->right); //Call the right substree
        // Swap the nodes
        TreeNode* temp = root->left;
        root->left = root->right;
        root->right = temp;
        return root; // Return the root
    }
};
```

CODEFORCES: kori nai, but tomra parba.