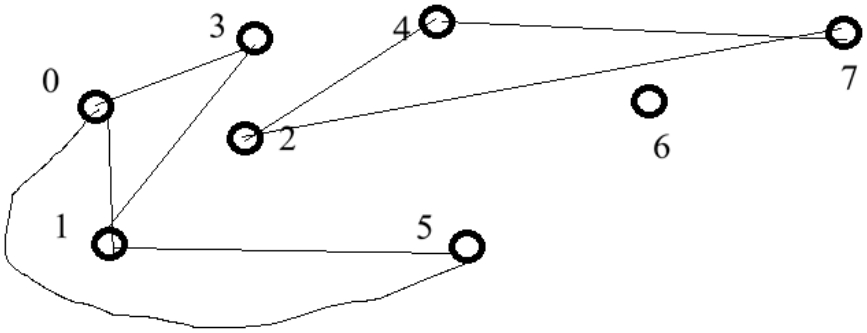
Depth First Search (DFS)

Depth First Search (DFS) is a classic graph algorithm that allows the exploration of a graph. A graph is a collection of locations (vertices), and connections (edges) between those locations. The goal of a DFS is to mark each location that is reachable from a given starting point. An undirected graph is one where each edge can be traveled (traversed) in either direction.

If we run a DFS from a particular vertex in a graph, then the set of vertices that the DFS reaches (can be reached by traversing a sequence of connected edges) is defined as a connected component of a graph. Any undirected graph can be decomposed into a set of connected components. For example, consider the following graph pictured below:



In this graph, the vertices labeled 0, 1, 3 and 5 are part of one connected component, the vertices labeled 2, 4 and 7 are part of a second connected component, and 6, but itself, is part of the third connected component that comprise the graph.

Graph Storage

The easiest way to store a graph is to store it as a vector of vectors. The first index into the vector of vectors represents a vertex. For each vertex we store a vector which has a list of each of the vertices to which it's connected. For the graph above, here is how it would be stored:

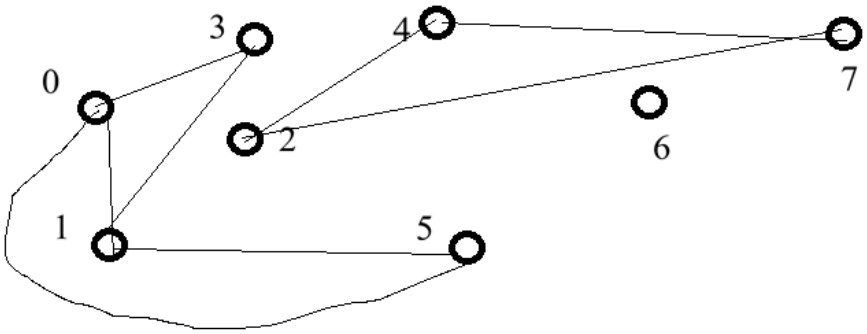
0 → 1, 3, 5 (this is a vector)
1 → 0, 3, 5
2 → 4, 7
3 → 0, 1
4 → 2, 7
5 → 0, 1
6 →
7 → 2, 4

Usually, in competitive programming problems, we create the vector of vectors to be size, n, where n is the number of vertices in the graph, and then for each undirected edge (u, v) listed, we add v to u's list and u to v's list.

DFS Overview, Trace and Pseudocode

A depth first search explores a network connected by roads (in computer science this is called a graph) in a recursive fashion, starting at some source location and then recursively exploring all unexplored neighbors. The use of recursion forces the algorithm to mark separate nodes (depth) on a single path, until it gets stuck. Then recursive calls pop off the stack until there's a vertex from which there exists an unexplored vertex to travel to, and then the recursion goes there.

Consider a DFS that starts at vertex 0 on the graph on the previous page, redrawn here for ease:



DFS(0) will mark that it's visited 0 and then travel to an unvisited neighbor. Let's say it goes to 1, so DFS(0) calls DFS(1). DFS(1) marks 1 and then skips going to 0 since it's visited, but does go to 3, so it calls DFS(3). DFS(3) marks 3 and at this point the call stack looks like this:

DFS(3)
DFS(1)
DFS(0)

When DFS(3) looks at its neighbors, 0 and 1, both are already marked, so it gets stuck, which is to say that the call completes, leaving the call stack to look like this:

DFS(1)
DFS(0)

In the DFS(1) call, it hasn't yet tried visiting 5, which is unvisited, so it goes there next calling DFS(5). At this point, Each of the calls DFS(5), DFS(1) and then DFS(0) have no new vertices to explore so the original call to DFS(0) completes, marking vertices 0, 1, 3 and 5 as being part of a single component. One would have to try multiple DFS calls until each vertex in a graph was marked to count up all of the components of the graph.

In order to get the code to work, you just have to pass in a boolean vector (or integer vector) into the DFS call. The value at index i of this vector can either store whether or not that vertex has been visited (if boolean) or which component number that vertex is a part of.

The high-level sketch of the DFS function looks like this:

```
DFS(Graph g, int vertex, vector used) {

        1. Mark used[vector] = true.
        2. For each neighbor next of vertex
                if not used[next]
                        DFS(g, next, used)
}
```

Traditional DFS Code

In this section we'll look at traditional DFS code on a graph. The input format is as follows:

First line contains two space separated integers, ***n***, the number of vertices in the graph and ***e***, the number of edges in the graph, respectively.

The following ***e*** lines each contain two distinct positive integers, ***u*** and ***v***, in between 1 and ***n***, inclusive, indicating that vertices ***u*** and ***v*** are connected by an edge. Each of these pairs are distinct.

The program will read in the graph and then run a BFS from every vertex. For each BFS, the corresponding distance array will be printed out.

Here is the code:

```
// Arup Guha
// Arup Guha
// 6/2/2026
// Illustration of how to read in a graph as a list of adjacencly lists and run
DFS on it in C++

using namespace std;

#include <iostream>
#include <vector>
#include <queue>

int numV;
vector< vector<int> > graph;

void dfs(int v, vector<int>& compNum, vector<int>& items, int ID);
void print(vector<int>& v);

int main() {

    int numE, v1, v2;
    cin >> numV >> numE;
    graph.resize(numV);

    // Read in the edges - add both ways.
    for (int i=0; i<numE; i++) {
        cin >> v1 >> v2;
```

```
        graph[v1-1].push_back(v2-1);
        graph[v2-1].push_back(v1-1);
    }

    // compNum[i] will store which connected component vertex i is in.
    int numComp = 0;
    vector<int> compNum(numV, -1);

    vector<vector<int>> compList;

    // Run a BFS from each vertex.
    for (int i=0; i<numV; i++) {

        // We've visited this before.
        if (compNum[i] != -1) continue;

        // Store all items in this component here.
        vector<int> items;
        dfs(i, compNum, items, numComp);

        // Update for the next component.
        numComp++;

        // Add to component list.
        compList.push_back(items);
    }

    // Print the compNum vector.
    cout << "compNum vector = ";
    print(compNum);

    // Print each component.
    for (int i=0; i<numComp; i++) {
        cout << "component # " << i << " ";
        print(compList[i]);
    }

    return 0;
}

void dfs(int v, vector<int>& compNum, vector<int>& items, int ID) {

    // Mark which component v is in.
    compNum[v] = ID;
    items.push_back(v);

    // Go through neighbors.
    for (int next: graph[v]) {

        // Been visited.
        if (compNum[next] != -1) continue;

        // Recursively mark everything unvisited connected to next.
        dfs(next, compNum, items, ID);
    }
}
```

```
// Prints out the contents of a vector.
void print(vector<int>& v) {
    for (int x: v)
        cout << x << " ";
    cout << endl;
}
```

A few notes about the code:

1. Most competitive programming problems label vertices 1 to n, but in code we label them 0 to n-1, so right when we read in the vertices, we subtract 1.

2. Edges are undirected in most problems, which means that whenever an edge is given between u and v, you have to add both the edge u→v and v→u. This is because v is next to u and u is next to v. One of the most common errors beginning programmers make is to forget adding undirected edges both ways in their graph.

3. If you don't like the iterator syntax of finding all the neighbors of a vertex, you can use a regular for loop with an index.

Now, let's dig into the details of the dfs itself. Recall that the graph is stored as a global variable, so it's not in the parameter list:

```
void dfs(int v, vector<int>& compNum, vector<int>& items, int ID) {
    // Mark which component v is in.
    compNum[v] = ID;
    items.push_back(v);

    // Go through neighbors.
    for (int next: graph[v]) {

        // Been visited.
        if (compNum[next] != -1) continue;

        // Recursively mark everything unvisited connected to next.
        dfs(next, compNum, items, ID);
    }
}
```

The parameters are: v, the vertex the DFS is being run on, compNum, the vector storing the component number of each vertex, items, the vector storing each vertex for this component, and ID, the identification number for this component (the one with vertex v).

First, we update the component number for vertex v and add it to the items list.

Then, we want to visit all unvisited neighbors. In code, you just have to loop through all neighbors and skip over the visited ones, which is what the if statement in the loop does. Then, whenever you get to an unvisited neighbor, just run a DFS from it. The DFS call can happen at most once from each vertex, so its run time is O(V+E), where V is the number of vertices and E is the number of edges in the graph.

Kattis Problem: Sky Islands

Here is a link to the problem: https://open.kattis.com/problems/skyislands

In this problem, it's fairly clear that the problem is asking if the graph given as input is a connected graph (so one connected component total). This means we can read in the graph and then run one single DFS from any vertex of our choosing (usually we pick 0), and after the DFS completes, we'll just check if every vertex is visited or not.

Since we don't have to keep track of various components, the solution that is included below is more simple than the previous DFS code shown. The recursive DFS function just takes in the graph, a visited array and the vertex upon which the DFS is called. Here's that function in isolation first:

```
// Runs a DFS from loc in the graph g, marking visited locations in visited.
void go(vector<vector<int>>& g, vector<bool>& visited, int loc) {

    // Mark it.
    visited[loc] = true;

    for (auto x=g[loc].begin(); x != g[loc].end(); x++) {

        // Been here before.
        if (visited[*x]) continue;

        // We can go to this new neighbor, *x...
        go(g, visited, *x);
    }
}
```

The rest of the code (main) just reads in the input and forms the graph. Here is that code:

```
int main() {

    // Get # vertices, edges.
    int n, numE;
    cin >> n >> numE;

    // Set up an empty graph.
    vector<vector<int>> g(n);

    // Read in each edge.
    for (int i=0; i<numE; i++) {

        // Do 0 based.
        int u, v;
        scanf("%d%d", &u, &v);
        u--;
        v--;

        // Mark both ways.
        g[u].push_back(v);
        g[v].push_back(u);
    }
```

```
    // Will mark where we've visited.
    vector<bool> visited(n, false);

    // Run our DFS!
    go(g, visited, 0);

    // If anyone's not visited, change the result to false.
    bool res = true;
    for (int i=0; i<n; i++)
        if (!visited[i])
            res = false;

    // Ta da!
    if (res)    printf("YES\n");
    else        printf("NO\n");

    return 0;
}
```

First, note that we can quickly initialize a vector of n empty vectors like so:

```
vector<vector<int>> g(n);
```

Then it's fairly easy to add each edge as needed in the edge reading loop.

Finally, after we complete our DFS call, we just have to loop through the visited vector one last time to see if any vertex is unvisited.

Kattis Problem: Where's My Internet?

In this problem, there is a graph given and a specific vertex, vertex #1 is connected to the internet. In order for other houses to connect to the internet, they must be directly or indirectly connected to house #1 via cables. The input graph given has the list of cables between houses. The goal of the problem is to determine each house without internet. Here is the complete description of problem:

https://open.kattis.com/problems/wheresmyinternet

Notice that we should be able to use our Sky Islands code almost exactly, but at the end, we'll have to output either "Connected" or a list of all of the vertices that aren't connected. Let's look at the old code that looks at the visited array in the previous problem and look to modify it:

```
// If anyone's not visited, change the result to false.
bool res = true;
for (int i=0; i<n; i++)
    if (!visited[i])
        res = false;

// Ta da!
if (res)    printf("YES\n");
else        printf("NO\n");
```

While we loop through the visited array, we might as well add each unvisited item to a new vector:

```
// If anyone's not visited, change the result to false, store unvisited.
vector<int> unvisited;
bool res = true;
for (int i=0; i<n; i++) {
    if (!visited[i]) {
        res = false;
        unvisited.push_back(i+1);
    }
}
```

Since the problem requests 1-based output, we have to add 1 to each of our 0-based vertex numbers. Then for output, instead of outputting no, we must output each item in our unvisited list:

```
// Ta da!
if (res)
    printf("Connected\n");
else {
    for (int i=0; i<unvisited.size(); i++)
        cout << unvisited[i] << endl;
}
```

Use of DX/DY Arrays

Some DFS problems in programming competitions are on two dimensional grids. There's lots of flavor text, but most of these problems have some sort of movement rules, a starting position and some sort of goal position (or positions). A few problems of this nature are an exact DFS without an frills, but most of them require some observations that necessitate minor modifications to a regular DFS. Regardless, almost all of these problems delineate movement rules in a grid, and this grid can be thought of as a graph, where each grid square is a vertex and each edge is between two grid cells that can be directly traveled between. If we are at some location (x, y) in the grid, we can reach some list of positions dictated by directions of movement. Perhaps, we can move up, down, left or right, which means that from (x, y), we can, in one move, get to (x-1, y), (x, y-1), (x, y+1) and (x+1, y). In other problems, we can move diagonally as well, meaning that we can move to eight possible locations. These are the two most common specifications for movement, but others exist; you just have to read the problem carefully!

No matter, what the specification is, in each case, always create two parallel vectors that store the possible offset of movements in X and Y as follows:

```
const vector<int> DX = {-1,0,0,1};
const vector<int> DY = {0,-1,1,0};
```

In this example, we encode the movement for up, left, right and down respectively, assuming that x is up and down and y is left and right.

If our current position is (curx, cury), here is how we iterate through all possible next locations:

```
for (int i=0; i<DX.size(); i++) {
    int nextx = curx + DX[i];
    int nexty = cury + DY[i];

    // Skip if (nextx, nexty) is out of bounds, previously visited
    // or illegal.

    // If new and valid, add to queue.
}
```

Kattis Problem: Counting Stars

This our graph is given in a grid, where adjacent squares on the grid that are the '-' create a star and we have to count the number of stars in the grid. In our solution, we'll never formally store the graph (it's implicitly stored in the grid), but we'll run our DFS by traveling to adjacent star squares by only looking at where the potential edges could be (up, down, left or right). Many people often call a DFS run on a grid a floodfill, since this was the original term for the Microsoft Paint feature that fills an enclose region with a color. (The original floodfill just changes the color of every pixel in a connected region that is bound by some border.)

Here is a link to the problem:

https://open.kattis.com/problems/countingstars

We can loop through each square on the grid. If it's a star square ('-'), then we must find and mark the whole star that is connected to this point and add 1 to a total count. We use the DFS/floodfill to mark these squares. In the solution presented here, the '*' character will be used to fill stars. This way, once a star is seen, all of its grid squares will no longer appear to be new stars. Thus, over the course of the algorithm, this grid will change so that all regions with '-' will change to be marked with '*'. A different common technique is to not change the grid, but keep a separate 2D vector of boolean to mark which grid squares have been previously visited.

Here is the solution, pay special attention to the floodfill function fillStar:

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

// Stars are connected in 4 directions.
const vector<int> DR = {-1,0,0,1};
const vector<int> DC = {0,-1,1,0};
const char FILL = '*';
const char STAR = '-';

// Global grid
int r, c;
vector<string> grid;

// Function prototypes.
bool inbounds(int myr, int myc);
int solve();
void fillStar(int myr, int myc);

int main() {

    int cnt = 1;
    while (cin >> r) {
        cin >> c;
        grid.clear();
        for (int i=0; i<r; i++) {
            string tmp;
            cin >> tmp;
            grid.push_back(tmp);
        }
```

```
        // Solve it!
        cout << "Case " << cnt++ << ": " << solve() << endl;
    }

    return 0;
}

// Returns true iff (myr,myc) is in the grid.
bool inbounds(int myr, int myc) {
    return myr>=0 && myr<r && myc>=0 && myc<c;
}

// Solves the given input case.
int solve() {
    int res = 0;

    // Go to each square.
    for (int i=0; i<r; i++) {
        for (int j=0; j<c; j++) {

            // If necessary, fill this star.
            if (grid[i][j] == STAR) {
                res++;
                fillStar(i, j);
            }
        }
    }

    // Ta da!
    return res;
}

// Fill the star at (myr, myc).
void fillStar(int myr, int myc) {

    // Fill this square.
    grid[myr][myc] = FILL;

    // Try all directions.
    for (int i=0; i<DR.size(); i++) {

        // Skip stuff out of bounds and previously filled.
        if (!inbounds(myr+DR[i],myc+DC[i])) continue;
        if (grid[myr+DR[i]][myc+DC[i]] != STAR) continue;

        // Recursively fill this star.
        fillStar(myr+DR[i],myc+DC[i]);
    }
}
```

Notice that the fill function itself is quite short. The structure is generally similar in most applications. First fill/mark the square. Next, try all adjacent squares. Skip ones that are out of bounds, previously visited, or invalid. Then, if a neighbor is previously unvisited and valid, recursively call the fill function on that square.