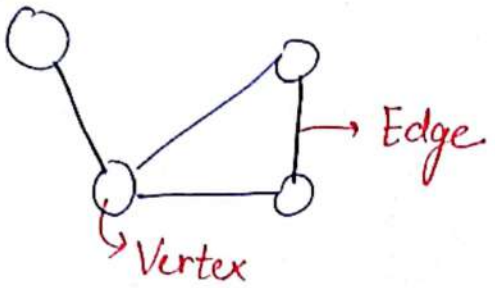
* **Graphs :-** (Interview Centric) Pre-req. [Recursion + Queues BFS & DFS]

collection of nodes known as vertices and edges.



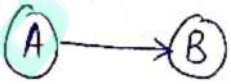
Applications :-

* Cost estimation (while travelling to select path).
* Google Maps (shortest distance + traffic).

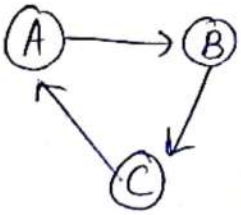
Terminologies :- [Graphs are easy but longer implementation]

* **Undirected :-** When the edge has no directivity i.e. it is bidirectional

* **Directed :-** When all the edges in a graph have directivity

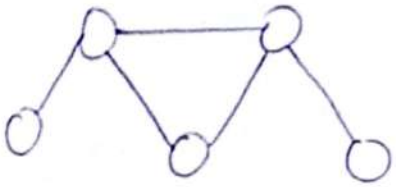


* **Cycle :-** During traversal in a directed graph, if you reach a particular node once again.

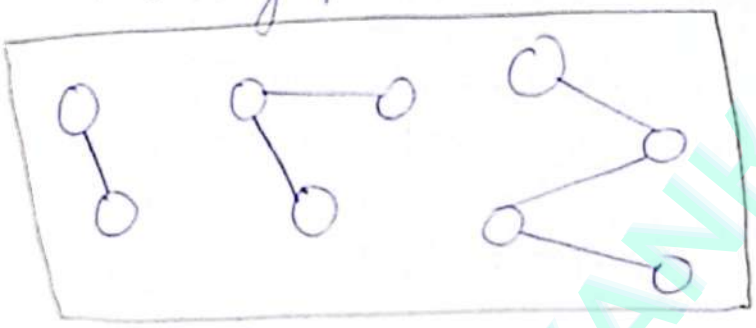


* **Cyclic Graph :-** When the graph has cycle
* **Acyclic Graph :-** When the graph does not have a cycle.

* **Connected component :-** When in a graph, all the nodes are connected to some or the other node.

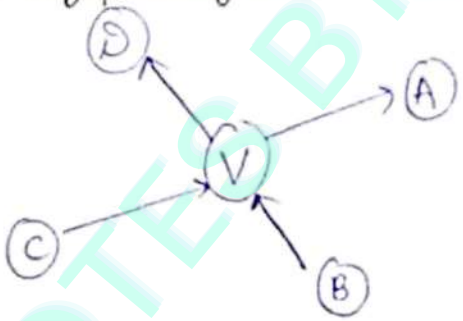


* **Disconnected graph :-** It has multiple connected components but in whole, the graph is disconnected.



→ If in any undirected graph, I start from one node & can reach all the other nodes, it is connected graph.

* **Incoming / Outgoing edges :-** only possible for directed graph



In = {B, C}
Out = {A, D}

* **Outdegree :-** No. of outgoing edges.

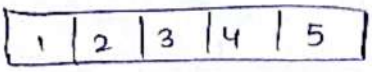
* **Indegree :-** No. of incoming edges.

Representation :-

(I) class graph {

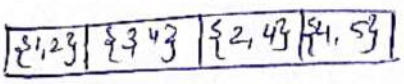
Time - $O(n^2)$
Space - $O(E)$

vector for vertices



<int>

vector for edges



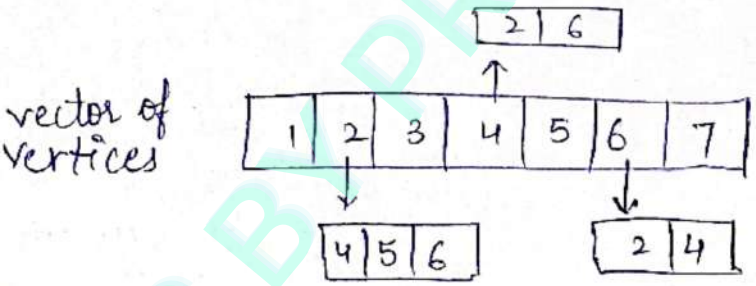
<source, edge>
pair<int, int>

};

Disadvantage :- Time Complexity to search in vector of ~~vert~~ edges is $O(n^2)$ [in case of complete graph]

(II) Adjacency List :- (vector of vectors).

Time - $O(n)$
Space - $O(E)$



Directly jump to index of vertex, and then search for its edges. So, time complexity is $O(1) + O(n)$

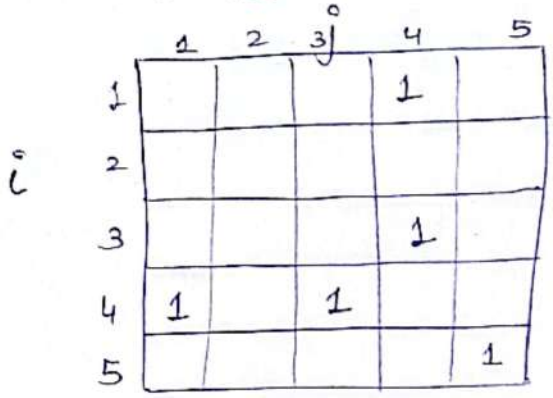
$= O(n)$

(III) **Adjacency Matrix:-** Make a 2-D matrix.

Go to $(i, j)^{th}$ cell and mark true

For undirected graph, mark (j, i) as well.

→ The matrix is diagonally symmetric



i \ j	1	2	3	4	5
1				1	
2					
3				1	
4	1		1		
5					1

Time – O(1)
Space – $O(n^2)$

Time Complexity: $O(1)$

Space Complexity: $O(n^2)$

where n = no. of vertices.

★★ When you have large no. of queries or (n → no. of vertices) $n <= 1000$, use AM
↳ $n^2 = 10^6$ 1MB.

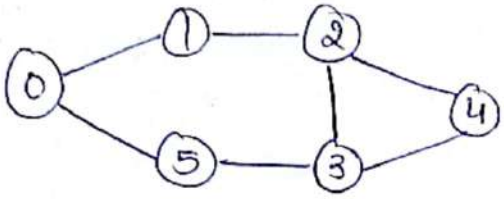
★★ moderate no. of queries or $n \sim 10^5, 10^6$, use A.L.

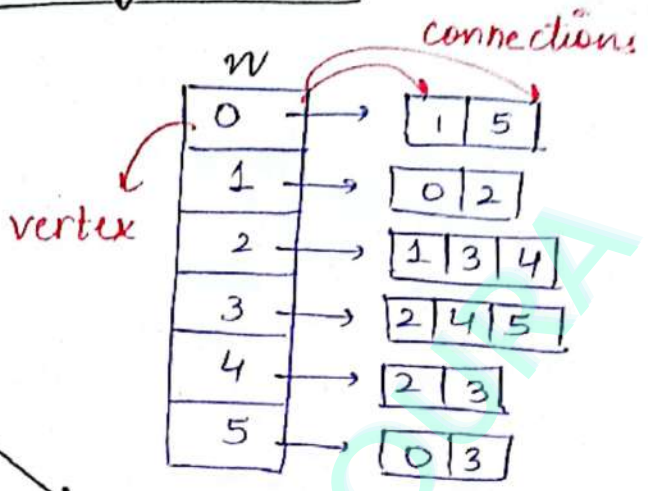
If we take larger n, memory used would be around 10^8 or 10^9, our system might crash b coz static programs do not use huge memory

→ Adjacency list can be optimised from $O(n)$ to $O(\log n)$ if instead using vector<int>, we use set<int>

Implementation using Adjacency List :-

vector of vectors





input :- (Undirected)

n → no. of vertices
e → no. of edges.

- e times f (first)
- e times s (second)

Input :-

```
6 7        n, e
0 1
1 2
2 3
3 4        f, s (7 times)
2 4
3 5
5 0
```

[considering 0 based indexing]

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

void print(vector<vector<int>> &adj){
    for(int i=0; i<adj.size(); i++){
        cout<< i <<" : ";
        for(int j=0; j< adj[i].size(); j++){
            cout << adj[i][j] << " ";
        }
        cout << endl;
    }
}

int main(){
    int n, e;
    cin >> n >> e;
    vector< vector<int> > adj(n);
}
```

initialising with size n.

vector storing vector<int> datatype.

```
    while (e--){
        int f, s;
        cin>> f >> s;
        adj [f]. push_back(s);
        adj [s] . push_back(f);
    }
    print (adj);
}
```

// undirected graph

→ We can use list STL to implement adjacency list. It behaves like normal linked list.

```
list<int> *adj;    adj = new list<int> [n];
```

→ Implemented ahead.

→ Another way :- array of vectors :-

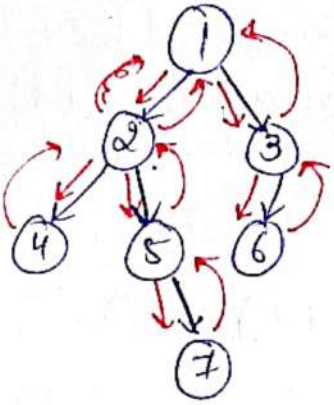
```
vector <int> adj [n];
```

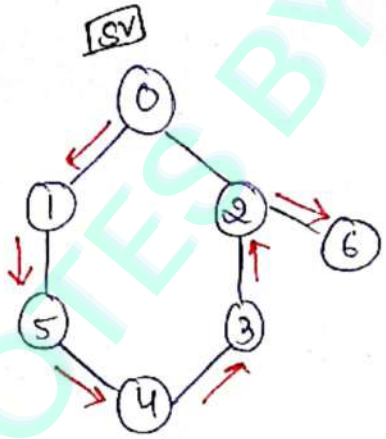
Implementation using Adjacency Matrix :-

```
#include <bits/stdc++.h>
using namespace std;
void print (int grid [][100], int n) {
    for (int i=0; i<n; i++) {
        cout << i << " : ";
        for (int j=0; j<n; j++) {
            if (grid[i][j] == 1) cout<< j << " ";
        }
        cout << endl;
    }
}
int main(){
    int n, e;
    cin>> n>> e;
```

```
int grid[100][100];
for(int i=0; i<n; i++){
    for(int j=0; j<n; j++){
        grid[i][j] = 0; // Initialising all to zero.
    }
}
while(e--){
    int f, s;
    cin>>f>>s;
    grid[f][s] = 1;
    grid[s][f] = 1; // Undirected graph
}
print(grid, n);
}
```

Depth First Search (DFS):





0 1 5 4 3 2 6

Don't visit a node again, if once it is visited.

Range Based for Loop :-

integers inside that container (iterator)

```
for(int c : adj[i])
```

c → inside

```
for(pair<int,int> x : arr)
```

↳ To ~~void~~ avoid writing so long, directly use auto

```
for(auto x : arr)
```

↳ directly fetches the kind of datatype

Implementation of graphs using list STL :-

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

list<int> *adj; // dynamically & globally allocated array of lists.

void print(int n){
    for(int i=0; i<n; i++){
        cout << i << ": ";
        for(auto j : adj[i]){
            cout << j << " ";
        }
        cout << endl;
    }
}
int main(){
    int n, e;
    cin >> n >> e;
```

```
    adj = new list<int>[n];
    while (e--){
        int f, s;
        cin >> f >> s;

        adj[f].push_back(s);
        adj[s].push_back(f);
    }
    print(n);
}
```

Continuing to DFS:-

```
vector<int> adj[100]; // global array of vectors
void dfs(int sv, vector<int> &vis){
    cout<< sv << " ";
    vis[sv] = true;

    for(int i=0; i< adj[sv].size(); i++){
        int neighbor= adj[sv][i];
        if (vis[neighbor] == false)  dfs(neighbor, vis);
    }
}
int main(){
    int n, e;
    cin >> n >> e;

    while (e--){
        int f, s;
        cin >> f >> s;

        adj[f].push_back(s);
        adj[s].push_back(f);
    }
    vector<int> vs(n,0);   // all values initialised to zero
    dfs(0, vis);           // size
}
```

OUTPUT:- (same input)

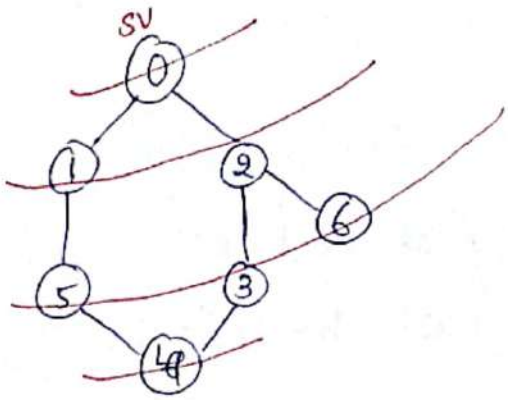
0 1 5 4 3 2 6

Breadth First Search (BFS):

FCFS

0 1 2 5 3 6 4

will use Queue.



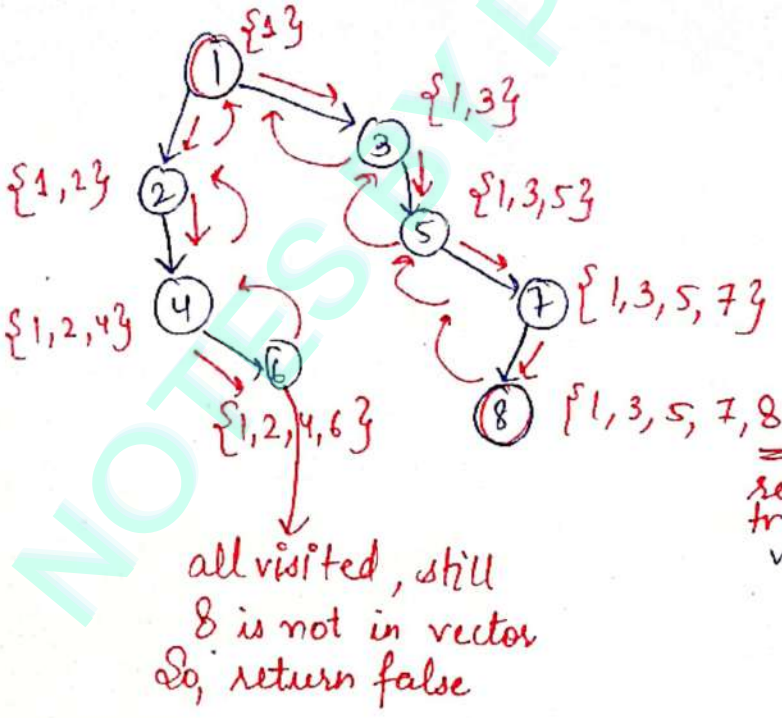
0 — 1, 2
1 — 0, 5
2 — , 0, 3, 6
3 — 2, 4
4 — 3, 5
5 — 1, 4
6 — 2

```
void bfs(int sv, int n){
    vector<int> vis(n,0);
    queue<int> q;
    q.push(sv);
    vis[sv] = 1;

    while(!q.empty()){
        int f = q.front();
        q.pop();
        cout << f << " ";
        for(int i=0; i<adj[f].size(); i++){
            int neigbor = adj[f][i];
            if(vis[neighbor]==0){
                vis[neighbor] = 1;
                q.push(neighbor);
            }
        }
    }
}
```

Ques Source and destination are given. Tell whether the source & destination has a path or not.

→
```
bool path (int s, int d, vector<int> & vis){
    if (s==d) return true;
    vis[s] = true;
    for(int i=0; i<adj[s].size(); i++){
        int neighbor = adj[s][i];
        if ( vis[neighbor] == false) {
            if (path( neighbor, d, vis) == true) return true;
        }
    }
    return false;
}
```

Get path (using DFS) :- s = 1, d = 8



return true and vector

```
vector<int> v;
v.push_back(s v);
if (s v == d)
    push_back(d);
return vector;
```

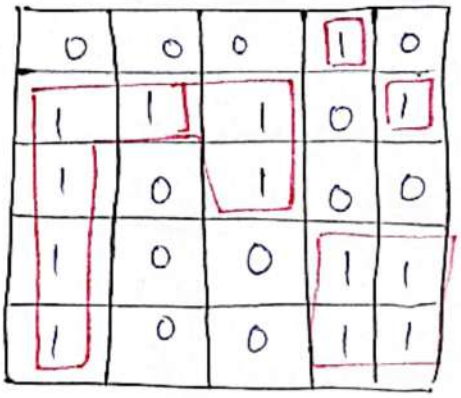
For same ques, if we do by BFS, we need map to store the parent.

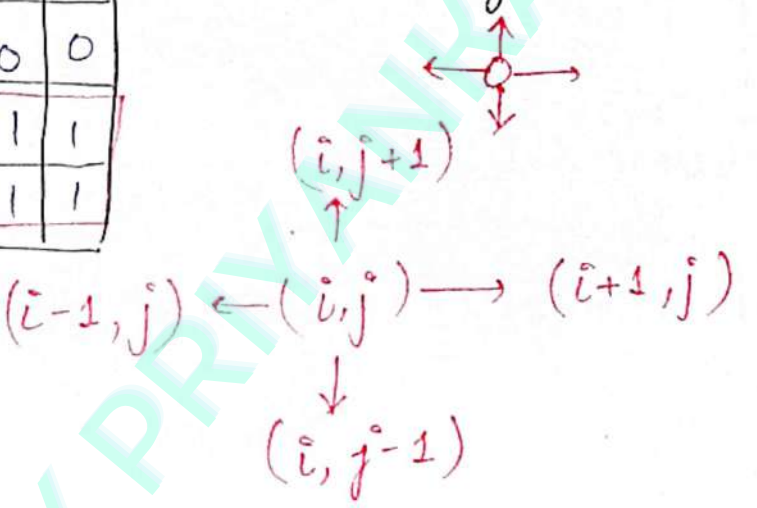
Ques Is connected?

Start DFS from any node
After finishing DFS, check vis array.
If any entry in vis is false, the graph is not connected.

Ques Largest piece.

A matrix is given, find the largest cake.
(1 is a cake)
0 is empty space.
I can't go diagonally.



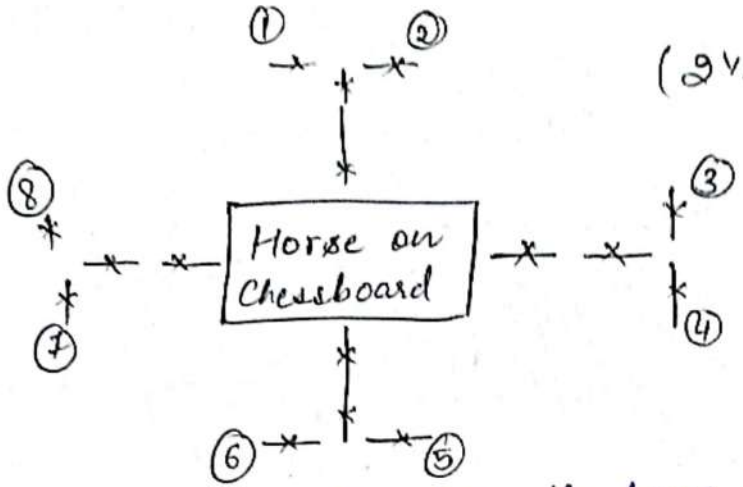


Intuition:-

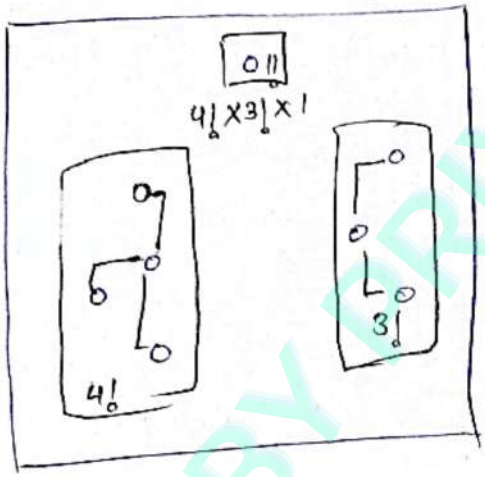
Scan through the matrix, whenever you find a 1, apply dfs and return the max^n of the dfs covered.

And after dfs, make the 1's 0 so that they won't be involved to evaluate dfs again.

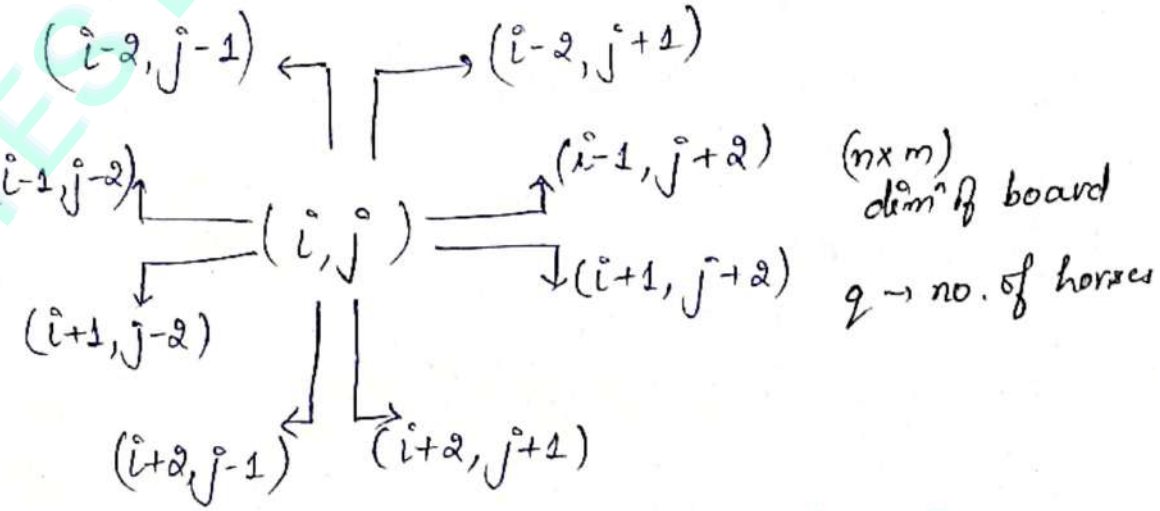
Ques Connected Horses



Total → 9 positions [8 where the horse can go & 1 where it is present] of horses given. Provide all possible configurations of the horses. — A horse will change its position with another reachable horse only, it won't go in empty space.



DFS would be used to calculate reachable horses.

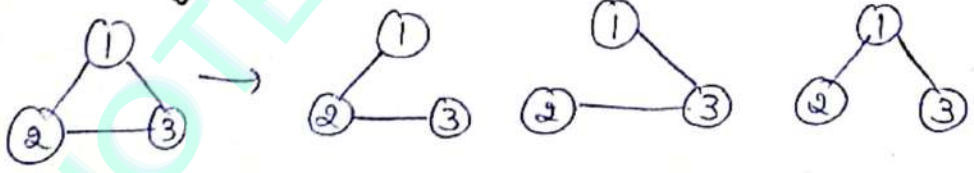


To store horses, we will use adjacency list/matrix.

```
#include<bits/stdc++.h>
using namespace std;
#define int long long // instead of changing int to long long at every place, you can define at top.
                      // ⇒ Remember to make main() as signed.

int n, m, q;
int MOD = 1e9 +7; // 1000000007 or 10^9 +7
int board[1001][1001]; // adjacency matrix
int dir[8][2] = {{-2,-1}, {-2,1}, {2,1}, {2,-1}, {1,2}, {1,-2},
                 {-1,-2}, {-1,2}};

void dfs(int i, int j, int &cnt){ // reference variable
    cnt++; // bcoz we found a horse
    board[i][j] = -1; // to prevent double count

    for(int d=0; d<8; d++){ // going in 8 directions of that horse
        int newX = i + dir[d][0]; // element of 2D array
        int newY = j + dir[d][1];
                   (x,y)
        if(newX>=0 && newY>=0 && newX<n && newY<m &&
           board[newX][newY]==1){ // bound check
                                  // Horse present.
            dfs(newX, newY, cnt);
        }
    }
}

int fact(int n){
    if(n==1) return 1;
    return (n*fact(n-1)) % MOD;
}

signed main(){
    int t;
    cin >> t;
    while(t--){
```

```
cin>> n>> m >> q;        // dimensions of board. no. of horses
memset(board, -1, sizeof(board)); // to clear out memory bcoz after every test case, some places might be occupied. (can only take 0,1,-1).
while(q--) { // taking input of position of all horses.
    int i, j;
    cin>> i>> j;
    i--; j--; // bcoz our table starts from (0,0)
    board[i][j] = 1; // marking those positions as 1, where horse is present
}
int ans = 1; // atleast one configuration is there
for(int i=0; i<n; i++) {
    for(int j=0; j<m; j++) {
        if(board[i][j] == 1) {
            int cnt = 0; // no. of horses reachable from (i,j)
            dfs(i,j,cnt);
            ans = (ans * fact(cnt)) % MOD;
        }
    }
}
cout << ans << endl;
}
}
```

Spanning Tree:- Configuration which covers all vertices with min^{m} no. of edges $(V-1)$.



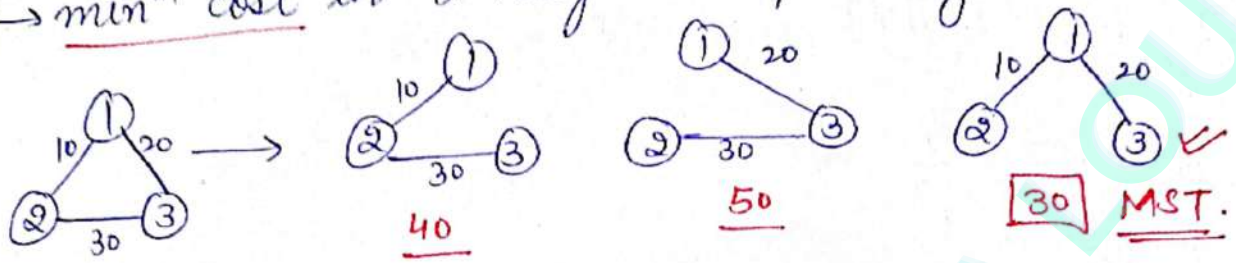
Properties:- * ST is a subset of the graph.

* ST can't be disconnected.
* Graph should not be disconnected.
* In an ST, if no. of vertices is V, then no. of edges is $V-1$
* We can never have cycle in ST.

* A connected and undirected graph has atleast 1 ST.
* ST is minimally connected to cover all the vertices such that if I remove 1 edge, it would become disconnected.

Minimum Spanning Trees (MST):-

↳ min^m cost in a weighted spanning tree.



Disjoint Sets:- two or more sets with nothing in common.

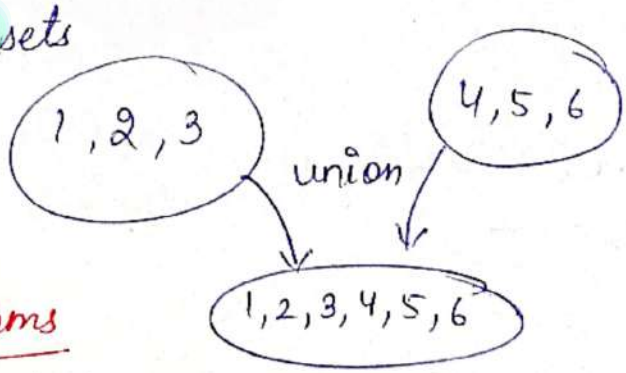
$S_1 = \{1, 2, 3\}$ $S_2 = \{4, 5\}$

$S_1 \cap S_2 = \phi$

Use cases of disjoint sets:-

(1) If we have an element and keep track of which set it belongs to. **find** (7) → {1, 2, 3, 7} → {4, 5, 6}

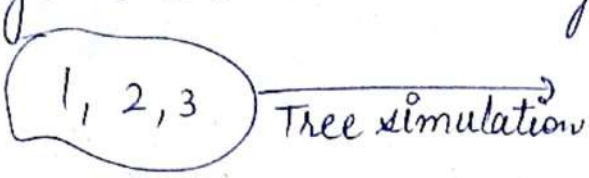
(2) **Merge** 2 or more sets

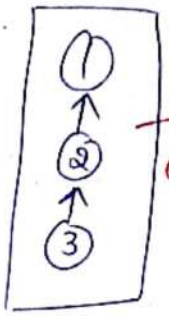


Union and find algorithms

Implementation of disjoint sets :-

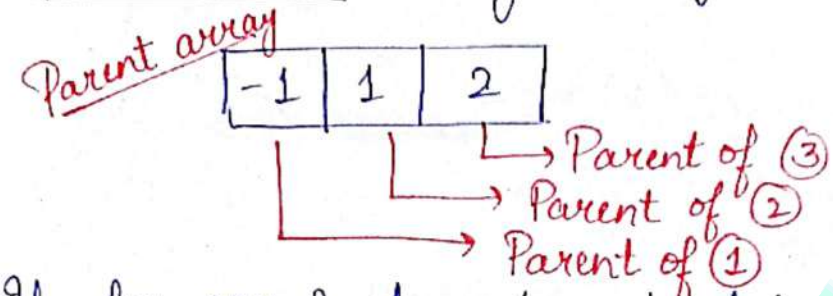
Disjoint set uses chaining :- Parent-child relationship.



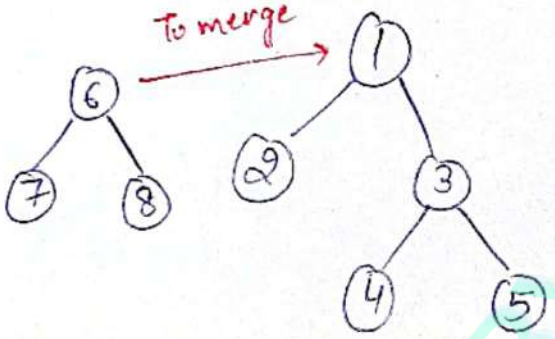


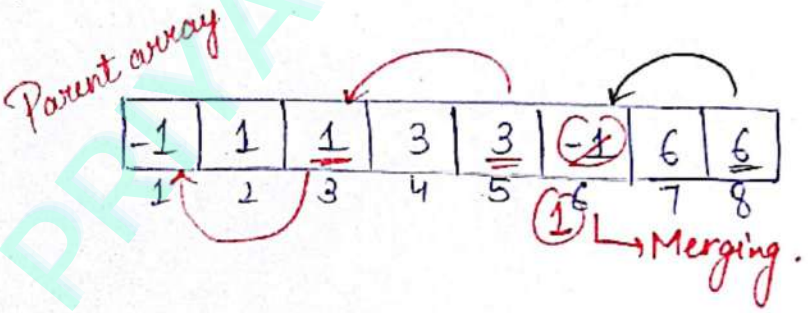
They all have a common absolute root (1)

Representation :- using arrays



If for any 2 elements, absolute root is same, they belong to the same set.





↑ If you want to merge both, make any one of them parent of other.

Absolute root of (5)?

(5) → (1)

(8) → (6)

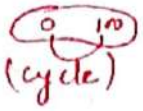
Whether (8) and (5) belong to same set?

↳ No, they don't have same absolute root.

What is the use of union & find algo (disjoint sets)

↳ In cycle detection (of undirected graph).

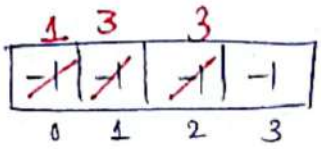
Cycle Detection :-



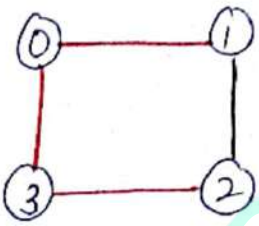
Edges
(0,1)
(0,3)
(2,3)
(1,2)
L → R

Check whether they form cycle or not.

Parent



(Initially, nothing is connected).



(0,1) aR(0) = -1 aR(1) = -1

I'll make 1 as parent of 0
R will be the parent of left.

(0,3) aR(0) = 1 aR(3) = -1] Both are in different components, cycle won't be formed.

I'll make 3 as parent of aR of 0.

(2,3) aR(2) = -1 aR(3) = -1

I'll make 3 as parent of 2

(1,2) aR(1) = 3 aR(2) = 3] Both belong to same component, if we join them, it will result in a cycle formation

Time Complexity :- $O(E \times V)$

process all edges

If the tree was skewed.

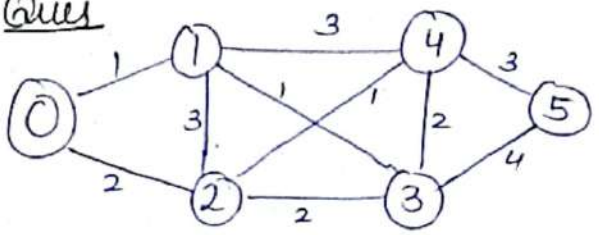
Direct Q's on Kruskal, Prim's & Djikstra algorithms.

Kruskal's Algorithm:- Greedy algorithm to find MST.

{Edge-based}

↳ can form forest in between (i.e. graph might be disconnected in between).

Ques



Steps:-

I) Sort all the edges in non-decreasing order of their weights.

II) Pick the smallest edge in terms of weight.

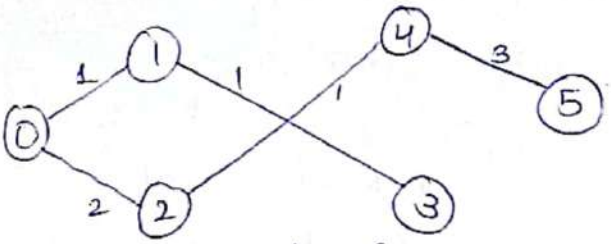
III) Check if this new edge forms a cycle or not.
- ↳ if not → include the edge in MST.
- ↳ else → discard.

IV) Repeat upto (V-1) edges in MST.

Ques Sorted	Edge List		Weight
~~count=0~~ →	0	1	1 ✓
~~1~~ →	1	3	1 ✓
~~2~~ →	2	4	1 ✓
~~3~~ →	0	2	2 ✓
4 →	2	3	2 ×
4 →	3	4	2 ×
4 →	1	2	3 ×
4 →	1	4	3 ×
~~4~~ →	4	5	3 ✓
5	3	5	4

↳ both have same aR

5 → V-1 edges



Cost = 8

Parent

0	1	2	3	4	5
1	3	4	~~1~~	~~1~~	-1

(4, 5)

T.C. of Kruskal:-

$$O(E \log E + EV)$$

- $E \log E$ → sorting edge list
- EV → cycle detection; it can be reduced to $E \log V$ using unionByRank algo

```
#include <bits/stdc++.h>
using namespace std;
// Edge class
class Edge{
    public:
    int s,d,w;
};
// Custom comparator
bool compare (edge a, edge b){
    return a.w < b.w;
}
int getParent ( int * parent, int p) { // to get absolute root
    if (parent[p] == p) return p; // base case
    return getParent( parent, parent[p]);
}
vector<edge> helper ( edge * arr, int * parent, int n ) {
    vector <edge> ans;
    int cnt=0, i=0; // i -> which edge to be considered
    while (cnt<n-1){ // until n-1 edges are not created
        edge currEdge = arr[i];
        // finding abblute root.
        int srcRoot = getParent (parent, currEdge.s);
        int desRoot = getParent (parent, currEdge.d);
        // Different root: different set
        if(srcRoot != desRoot){
            parent[srcRoot] = desRoot;
            ans.push_back (currEdge);
            cnt++;
        }
        i++;
    }
    return ans;
}
```

```
int main(){
    int n, e;
    cin >> n >> e;

    edge * arr = new edge[e];
    for(int i=0; i<e; i++){
        cin >> arr[i].s >> arr[i].d >> arr[i].w;
    }

    sort ( arr, arr+e , compare);
                  └→ end of arr        └→ custom comparator

    int * parent = new int[n];
    for (int i=0; i<n; i++)  parent[i] = i; // Instead of -1, taking
                                               parent initially as i.

    vector<edge> MST = helper(arr, parent, n);

    for (int i=0; i< MST.size(); i++) {
        cout << min(MST[i].s, MST[i].d) << " " << max(MST[i].s
             MST[i].d) << " " << MST[i].w << endl;
    }
}
```

Prim's Algorithm :- used to find MST. {greedy approach}
{vertex based}
└→ forest is never generated in between

Steps :-

I) Assign ∞ wt to all the nodes except source.

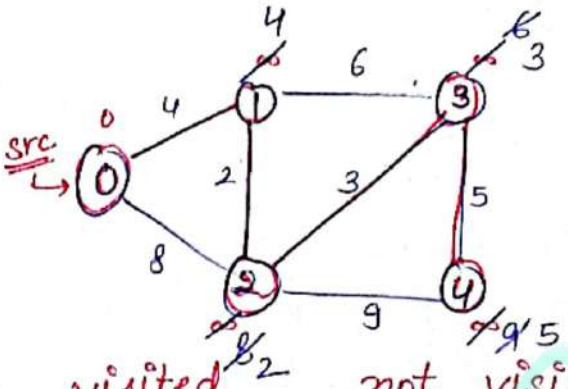
II) Select node with min^m wt.

III) Include this node in set → mark visited.

IV) And follow same for all of its adjacent nodes.

Ques

vertex	Parent	Weight
src 0	-1	0
1	~~0~~	~~∞~~ 4
2	~~0~~ ~~1~~	~~∞~~ ~~8~~ 2
3	~~2~~	~~∞~~ ~~6~~ 3
4	~~3~~	~~∞~~ ~~9~~ 5



visited	not visited
{ }	{0, 1, 2, 3, 4}
{0}	{1, 2, 3, 4}
{0, 1}	{2, 3, 4}
{0, 1, 2}	{3, 4}
{0, 1, 2, 3}	{4}
{0, 1, 2, 3, 4}	{ }

Then, I'll check all adjacents of {0} & will relive weights

min in non-visited

{We will use adjacency matrix}.

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

int findMinVertex(int * weights, bool * visited, int n) {
    int minVertex = -1;
    for(int i=0; i<n; i++) {
        if (visited[i] == false && (minVertex == -1 || weights[i] <
                                    weights[minVertex])) {
            minVertex = i;
        }
    }
    return minVertex;
}

void prims(int ** edges, int V) {
    int * weights = new int[V];
    int * parent = new int[V];
    bool * visited = new bool[V];
```

size of both arrays.

not visited yet

```
for (int i=0; i<V; i++){
    visited[i] = false;          //initially marking all as unvisited
    weights[i] = INT_MAX;        and their weight as ∞
}
0th element as source.
weights[0] = -1; //minm weight for source
parent[0] = 0;

for (int i=0; i<V; i++){
    int minVertex = findMinVertex(weights, visited, V);
    visited[minVertex] = true; //masking it visited
    for (int j=0; j<V; j++){
        if(visited[j] == false && edges[minVertex][j] != 0){
            if(edges[minVertex][j] < weights[j]){
                weights[j] = edges[minVertex][j];
            } parent[j] = minVertex;
        }
    }
}
for (int i=1; i<V; i++){
    cout << min(parent[i], i) << " " << max(parent[i], i) << " "
         << weights[i] << endl;
}
}
int main(){
    int V, E; temp*, vtemp*;
    cin >> V >> E;
    int ** edges = new int*[V]; // an array storing int * datatype
    for (int i=0; i<V; i++){
        edges[i] = new int[V];
        for (int j=0; j<V; j++){
            edges[i][j] = 0; // initially marking all entries in adj. matrix
        }                    as zero.
    }
    for (int i=0; i<E; i++){
        int s, f, w;
```

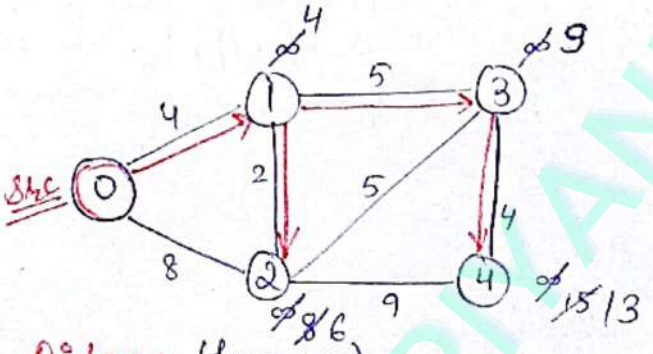
an edge exists b/w minVertex & j.

updating weight and parent.

```
    cin >> s >> f >> w;
    edges[s][f] = w;   // updating weights in adjacency matrix.
    edges[f][s] = w;
}
prims(edges, V);
}
```

Dijkstra Algorithm:- algorithm to find single source shortest path.

↳ similar to Prim's, there we assigned weights to nodes, here we will assign distance from the source.



Vertex	Distance (from src) (initially all ∞ except src→src)
0	0 ✓
1	~~∞~~ 4 ✓
2	~~∞~~ ~~8~~ 6 ✓
3	~~∞~~ ~~9~~ ✓
4	~~∞~~ ~~15~~ 13 ✓

visited	unvisited
{ }	{0, 1, 2, 3, 4}
{0} ← min distance	{1, 2, 3, 4}
{0, 1}	{2, 3, 4}
{0, 1, 2}	{3, 4}
{0, 1, 2, 3}	{4}
{0, 1, 2, 3, 4}	{ }

and then explore its adjacents and perform relaxation.

Steps:-

I) Find min^m distance vertex.

II) Mark it visited.

III) Explore all the unvisited vertices & update distance.

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

int getMinimumVertex(int * distance, bool * visited, int n){
    int minVertex = -1; // means we haven't selected any vertex.

    for(int i=0; i<n; i++){
        if(visited[i] == false && ( minVertex = -1 || distance[i] <
                                                    distance[minVertex])){
            minVertex = i;
        }
    }
    return minVertex;
}
// To find single source shortest path
void djikstra( int ** edges, int n){
    bool * visited = new bool[n];
    int * distance = new int[n]; // distance from source.

    for(int i=0; i<n; i++){
        visited [i] = false;
        distance [i] = INT_MAX;
    }
    // src : 0
    distance [0] = 0;
    // O(n^2)
    for(int i=0; i<n-1; i++){
        int minVertex = getMinimumVertex( distance, visited, n);
        visited [minVertex] = true;

        for(int j=0; j<n; j++){
            if (edges[minVertex][j] != 0 && visited [j] == false){
                if(distance [j] > distance[minVertex]+ edges [minVertex][j]){
                    distance [j] = distance [minVertex] + edges [minVertex][j];
                }
            }
        }
    }
```

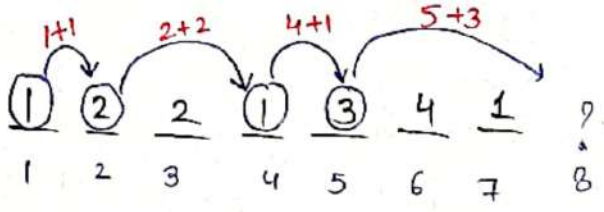
// intialising at starting.

```
    for(int i=0; i<n; i++){
        cout << i << " " << distance[i] << endl;
    }
}
int main(){
    int n, e;
    cin >> n >> e;
    // Adjacency Matrix
    int** edges = new int*[n];
    for(int i=0; i<n; i++){
        edges[i] = new int[n];
        for(int j=0; j<n; j++){
            edges[i][j] = 0; // no edge
        }
    }
    for(int i=0; i<e; i++){
        int f, s, w;
        cin >> f >> s >> w;
        edges[f][s] = w;
        edges[s][f] = w;
    }
    djikstra(edges, n);
}
```

Ques From i → can go to position i + arr[i]

t is the index you have to reach

(n-1) indexes are given



If t lies in the path, return yes.

But if t stays behind (but not in path eg. index 3) return no.

Return no → if t is not in the range as well

```cpp
#include <iostream>
using namespace std;
int main(){
    int n, t;
    cin >> n >> t;

    int * arr = new int[n-1];
    for(int i=1; i<n; i++) cin >> arr[i];

    int i = 1;
    bool reached = false;
    while( i<n && i<=t){
        if(i==t){//reached the index
            reached = true;
            break;
        }
        i += i + arr[i];
    }
    if(reached) cout<<"YES!"<<endl;
    else cout<<"NO!"<<endl;
}
```

OUTPUT:-

```
8 8
1 2 2 1 3 4 1
NO!
8 2
1 2 2 1 3 4 1
YES!
```

Topological Sort:- very useful in activities which have specific orders.

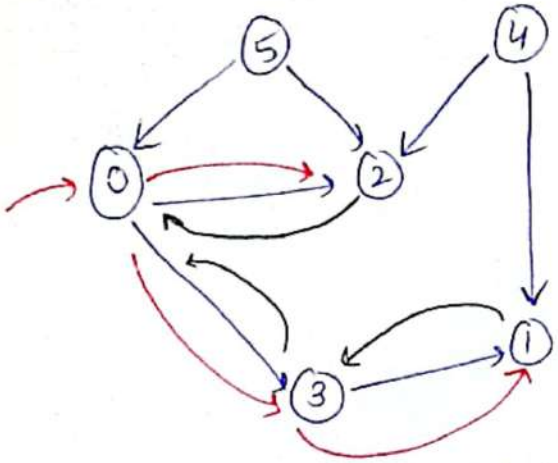
for example. ① Breakfast → ② Lunch → ③ Dinner.

Topological sort is only possible for directed acyclic graph (DAG)

DFS + stack would be used.

stack

2	1	3	0	4	5



visited

T	T	T	T	T	T
F	F	F	F	F	F
0	1	2	3	4	5

Initially, visited is false.

I'll start with first node (0), mark it true and go in its DFS.

From (2), I can't go anywhere so, push (2) in stack & backtrack.

Then, from (0), I can go to (3) and then to (1)

Then backtrack to (3), then to (0).

Now I'll check is there anything which is not visited. Yes

I'll go to (4). I can't go to (1) and (2) as they are already visited.

Then I'll go to (5).

Reverse order of stack is topological sort:

5 4 0 3 1 2

```
#include<bits/stdc++.h>
list<int> * adj;

// Graph coloring Method
bool cycle(int v, vector<int> & vis){
    if(vis[v] == 1) return true; // Not yet processed
    if (vis[v] == 2) return false; // Completely processed

    vis[v] = 1;
```

```
        for(int c: adj[v]){
            if(!vis[c]){
                if(cycle(c,vis)) return true;
            }
        }
        vis[v] = 2;
        return false;
    }
// Cycle Detection
bool checkCycles(int n){
    vector<int> vis(n,0);
    for(int i=0; i<n; i++){
        if(!vis[i]){
            if(cycle(i,vis)) return true;
        }
    }
    return false;
}
void dfs(int v, stack<int> &s, vector<bool> & vis){
    if(vis[v]) return;
    vis[v]=1;
    for(int c: adj[v]){
        if(!vis[c]){
            dfs(c,s,vis);
        }
    }
    s.push(v);
}
vector<int> ts(vector<vector<int>>& edges, int v, int e){
    adj = new list<int>[v];
    if(edges.size()==0){
        vector<int> ans;
        for(int i=0; i<v; i++) ans.push_back(i);
        return ans;
    }
    // Creating graph {u,v}  u→v
    for(int i=0; i<e; i++){
        int u = edges[i][0], v = edges[i][1];
        if(u!=v) adj[u].push_back(v);
    }
    if(checkcycles(v)) return {};
    stack<int>s;
    vector<bool> vis(v,false);
    vector<int> toposort;
    for(int i=0; i<v; i++){
        if(!vis[i])
            dfs(i,s,vis);
    }
    while(!s.empty()){
        int t=s.top();
        s.pop();
        toposort.push_back(t);
    }
    delete [] adj;
    return toposort;
}
```

Cycle Detection — Directed Graph 2

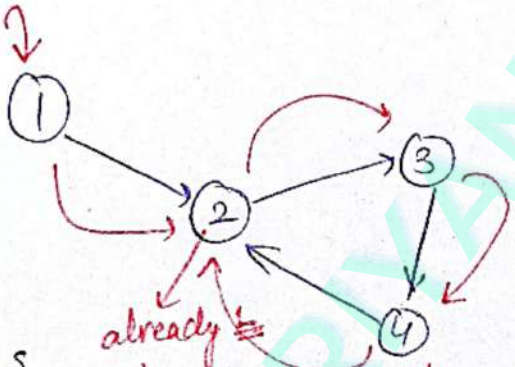
Graph coloring Method

T.C. $O(V+E)$

Color 0 → not visited.
Color 1 → visited but not processed.
Color 2 → visited & processed.

If we visit a (1) marked node again, then cycle exists.

If we visit a (2) marked node again, cycle does not exist.



color array

1	1	1	1
∅	∅	∅	∅
1	2	3	4

Algo

```
dfs (int v){
  color[v] = 1;
  for(neighbor: adj[v]){
    if(color(neighor) == 0){
      if(dfs(neighbor)) return true; false;
    }
    if(color(neighbor) == 1) return true; → cycle exists
  }
  color[v] = 2;
  return false;
}
```

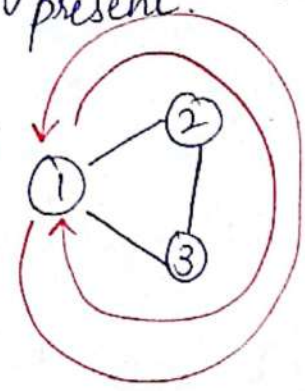
Algo

```
for(v=1; v<=4; v++)
  if(color[v] == 0 &&
     dfs(v))
  {
    return true;
  }
```

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

list <int> adj [1005];
vector <int> color;
int n, e;
bool dfs( int v){
    color[v] = 1; // visited and processing

    for(int u : adj[v]){ //checking for every neighbor u of v.
        if (color[u] == 0){
            if (dfs(u) == true) return true; // cycle found in neighbor
        }
        else if {color[u] == 1) return true; // we are in cycle of u.
    }
    // means no cycle found
    color[v] = 2;
    return false;
}

bool findCycle (){
    color.assign (n,0);

    for(int v=0; v<n; v++){
        if ( color[v]== 0 && dfs(v) == true){
            return true;
        }
    }
    return false;
}
int main(){
    cin >> n >> e;
    for(int i=0; i<e ; i++){
        int f, s;
        cin >> f >> s;
        adj[f].push_back(s); //  f ----> s
    }
    if (findCycle()) cout<< "Cycle present!";
    else cout << "Acyclic!";
}
```

Cycle Detection - Undirected Graph

If I visit any node again in undirected graph, cycle is present.

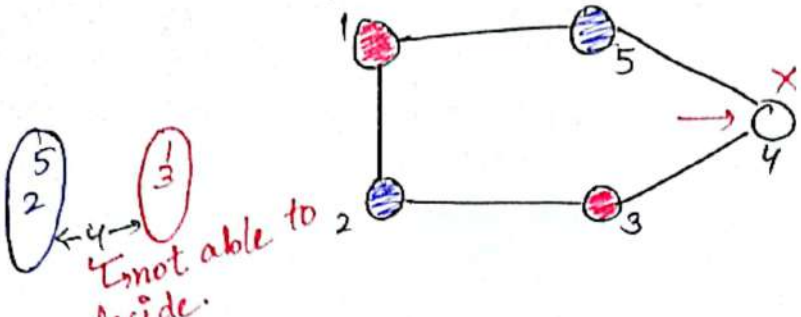


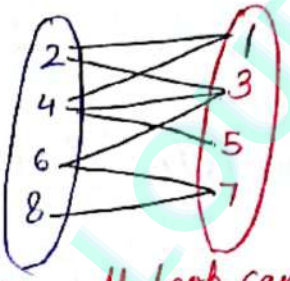
Color ① and ② are of same meaning in undirected graph.

```
bool dfs (int v){
    color[v] = 1; // visited

    for(int u : adj[v] ){ // adj[v] contains all neighbors of v.
        if (color[u] ==0){ // edge neighbor not visited
            if(dfs(u) == true) return true; // cycle found in neighbor
        }
        else return true; // color[u] = 1 i.e. already visited, so cycle present
    }
    return false;
}
```

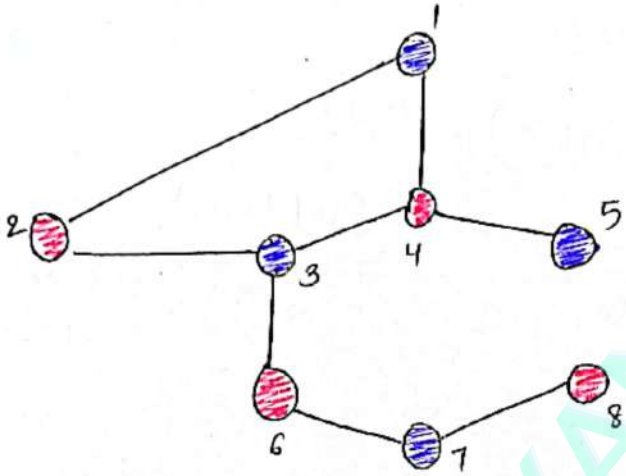
// Rest all code would be same as directed Graph

Ques Color this graph with red & blue such that no two adjacents have same color.





No self loop can be formed in any set
✓ possible



odd lengthed cycle ⟶ never bipartite.

even lengthed cycle ⟶ can be colored.

Bipartite Graph:-

Bipartite graph is a graph in which its vertices can be divided into two disjoint sets U & V such that every vertex/edge in set U has edge in set V.

Approach I :-

-1 → no color
0 → red
1 → blue

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

list<int> adj[10];
vector<int> color;
vector<bool> vis;
int n, e;
bool bipartite = true;

void usingColor (int v, int c){ // dfc
    if (color[v] != -1 && color[v] != c){
        bipartite = false;
        return;
    }

    color[v] = c;
    if (vis[v] == true) return;
    vis[v] = true;

    for (int u: adj[v]){ // now check for all neighbors
        usingColor(u, 1-c);
    }
}

int main(){
    cin >> n >> e;
    color.assign (n, -1);
    vis. assign (n, false);
    bipartite = true;

    for(int i=0; i<e; i++){
        int f, s;
        cin >> f >> s;
        adj[f]. push_back(s);
        adj[s]. push_back(f);
```

vertex
color
the vertex already has different color

```
    for(int i=0; i<n; i++){
        if(vis[i]==false){
            fill(i, 0); // filling color 0 in beginning
        }
    }
    if (bipartite) cout<<"Can be 2-colored!";
    else cout<<"Not possible!";
}
```

Approach II :→ (Using sets concept)

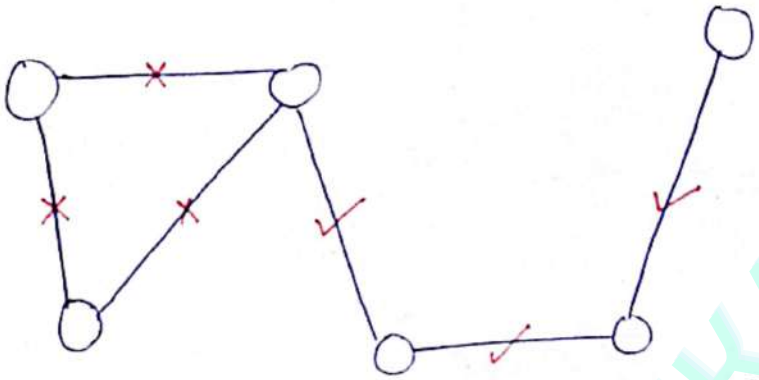
```
void solve(){
    vector<int> sets(n, -1);

    queue<int> q;
    q.push(0); // first vertex [in our graph its 0]
    sets[0] = 0;

    while(!q.empty()){
        int v = q.front();
        q.pop();
        for(int u : adj[v]){ // BFS
            if(sets[u] != -1 && sets[u]==sets[v]){
                bipartite = false;
                break;
            }
            else if(sets[u]==-1){
                sets[u] = 1 - sets[v];
                q.push(u);
            }
        }
    }
}
```

self-loop in a set (same color).

to avoid ∞ loop

Bridges and Articulation points :-

Bridges are those edges in graph which upon removal, increase connected components. (i.e. makes the graph disconnected).



Application :- Map of cities connected with roads is given, find all important roads, which when removed cause disappearance of a path b/w some pair of cities.

Algo :- For every edge :- [adjacency list].

1. Remove the edge
2. perform DFS, count connected components.
   if cc = 1, not a bridge
   if cc > 1, its a bridge.
3. Put back the edge.

$O(E(V+E))$

Articulation points :- same concept as bridges but these are vertices. and delete its associated edges.