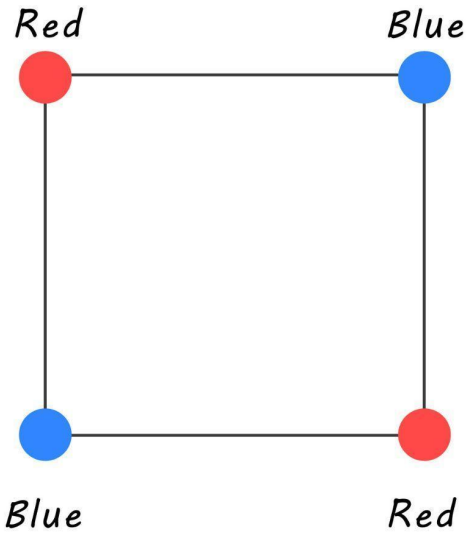
Graph III

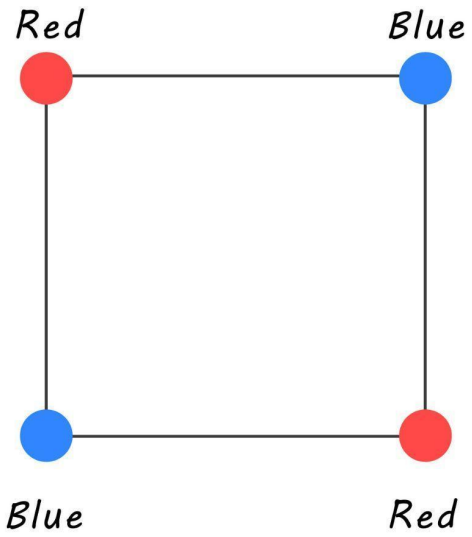
Bipartite Graph

A bipartite graph is a graph whose vertices can be divided into two disjoint sets so that every edge connects two vertices from different sets (i.e. there are no edges which connect vertices from the same set). These sets are usually called sides.

Let's take an example to understand the concept better



In the above graph, the nodes are marked with either Red Color or Blue Color. Now, these colors can be represented as Sets. So according to the definition of Bipartite graphs there should be no edge between two red vertices or two blue vertices. So the above graph is clearly not Bipartite.
Now, let's take a look at another example



The above graph has no edge between the vertices of the same color (set), therefore, it is a bipartite graph.

Given below is the algorithm to check for bipartiteness of a graph:

1. Use a col[] array which stores 0 or 1 for every node which denotes opposite colors.
2. Call the function DFS from any node.

3. If the node u has not been visited previously, then assign !col[v] to col[u] and call DFS again to visit nodes connected to u.
4. If at any point, color[u] is equal to col[v], then the node is not bipartite.
5. Modify the DFS function such that it returns a boolean value at the end.

Let's take a look at the code for this algorithm:

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

const int N = 1e5;
vector<int> adj[N+1];
int color[N+1];
bool vis[N+1];

//returns if the connected component contains invalid edge
bool dfs(int node , int col){
	vis[node] = true;
	color[node] = col;

	for(int v : adj[node]){
		if(vis[v] == false){
			bool result = dfs(v , col ^ 1);

			if(result == true) return true;
		}
		else{
			if(color[v] == col) return true;
		}
	}

	return false;
}


int main(){

	int n , m;
	int a , b;
```

```
        cin>>n>>m;

        for(int i=0;i<m;i++){
                cin>>a>>b;
                adj[a].push_back(b);
                adj[b].push_back(a);
        }

        //flag : whether graph contains invalid edge or not
        bool flag = false;

        for(int i=1;i<=n;i++)
                if(vis[i] == false){
                        bool result = dfs(i , 0);

                        if(result == true){
                                flag = true;
                                break;
                        }
                }

        if(flag) cout<<"Graph is not bi-partite";
        else    cout<<"Graph is bi-partite";

}
```

Bridges in a Graph

Problem Statement:

Given an undirected graph of V vertices and E edges. Your task is to find all the bridges in the given undirected graph. A bridge in any graph is defined as an edge which, when removed, makes the graph disconnected (or more precisely, increases the number of connected components in the graph).

Approach:

The idea of this approach is to do DFS traversal of the given graph.

- In a DFS tree, an edge (u, v) (u is the parent of v in a DFS tree) is a bridge if and only if
    - There does not exist any other alternative to reach v from u.
    - Or none of the vertices v and its descendants in the DFS traversal tree has a back-edge to vertex u or any of its ancestors. The reason is that each back edge gives us a cycle, and no edge that is a member of a cycle can be a bridge.
- Now to check this we will be maintaining two arrays
    - in[] - this stores the time when the vertex is first visited.
    - low[] - for every vertex u it stores the discovery time of the earliest visited vertex to which the vertex u or any of its descendants which have a back edge. Thus, low[u] is the minimum of three values,
        - in[u]: the discovery time of vertex v.

- in[p]: where p is an ancestor of u in the dfs tree and have an edge with p, i.e there is a back edge between p and u.
- low[v]: where there is a tree edge between u and v. This means that v is a vertex directly connected to u with an edge and v is not the parent node of u in the dfs tree.

- Initially, we will initialize low[] to -1. We will also initialize one variable timer = 0 to keep track of current discovery time.
- Now, there is a back edge from v or one of its descendants to ancestors of u (u is the parent of v) if and only if low[v]≤ tin[u]. This is due to the fact that if there is a back edge then the value of low[v] must be lesser than tin[u]. If low[v]=tin[u] then the back edge comes directly from v to u. Thus, the edge (u,v) in the DFS tree is a bridge if and only if low[v]>tin[u].

Code:

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

const int N = 1e5;

vector<int> adj[N+1];
int in[N+1] , low[N+1];
int timer;
bool vis[N+1];

void dfs(int node , int par){
      in[node] = low[node] = timer++;
      vis[node] = true;
```

```
      for(int v : adj[node]){
            if(v == par) continue;

            //forward edge node->v
            if(vis[v] == false){
                  dfs(v , node);
                  if(low[v] > in[node]) cout<<"Edge "<<node<<" -> "<<v<<" is a
bridge\n";
                  low[node] = min(low[node] , low[v]);
            }
            //backEdge
            else{
                  low[node] = min(low[node] , in[v]);
            }
      }

}

int main(){

       #ifndef ONLINE_JUDGE
       freopen("input.txt" , "r" , stdin);
       freopen("output.txt" , "w" , stdout);
       #endif

       int n , m;
       int a , b;

       cin>>n>>m;

       for(int i=1;i<=m;i++){
            cin>>a>>b;
            adj[a].push_back(b);
            adj[b].push_back(a);
       }

       dfs(1 , -1);


}
```

Time Complexity:

O(E * (V + E)), where V is the number of vertices and E is the number of edges in the graph.

In the worst case for each edge present in graph O(E), we will traverse the whole graph O(V+E), Hence the overall complexity will be O(E * (V + E)).

Space Complexity:

O(V), where V is the number of vertices in the graph.

In the worst case, extra space is required to store visited vertices O(V) and also an extra space is used by recursion function call stack O(V).

Articulation Points

Introduction:

A vertex in an undirected connected graph is an articulation point (or cut vertex) if removing it (and edges through it) disconnects the graph. Articulation points are also used to represent vulnerabilities in a connected network – single points whose failure would split the network into 2 or more components. They are useful for designing reliable networks.

For a disconnected undirected graph, an articulation point is a vertex removing which increases number of connected components.

Problem Statement:

You are given an undirected unweighted graph and you are supposed to find all articulation in the graph.

Approach:

The idea is to use DFS (Depth First Search). In DFS, we follow vertices in a tree form called DFS tree. In DFS tree, a vertex u is the parent of another vertex v, if v is discovered by u (obviously v is adjacent of u in the graph). In the DFS tree, a vertex u is an articulation point if one of the following two conditions is true.

1. u is root of DFS tree and it has at least two children.
2. u is not the root of DFS tree and it has a child v such that no vertex in the subtree rooted with v has a back edge to one of the ancestors (in the DFS tree) of u.

We do DFS traversal of given graph with additional code to find out Articulation Points (APs). In DFS traversal, we maintain a parent[] array where parent[u] stores parent of vertex u. Among the above-mentioned two cases, the first case is simple to detect. For every vertex, count children. If currently visited vertex u is root (parent[u] is NIL) and has more than two children, print it.

The second case is trickier. We maintain an array disc[] to store discovery time of vertices. For every node u, we need to find out the earliest visited vertex (the vertex

with minimum discovery time) that can be reached from subtree rooted with u. So we maintain an additional array low[] which is defined as follows.

```
low[u] = min(disc[u], disc[w])
where w is an ancestor of u and there is a back edge from
some descendant of u to w.
```

Code:

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

const int N = 1e5;
vector<int> adj[N+1];
bool isArticulationPoint[N+1];
bool vis[N+1];
int in[N+1] , low[N+1];
int timer;

void dfs(int node , int par){
      vis[node] = true;
      in[node] = low[node] = timer++;
      int child = 0;

      for(int v : adj[node]){
            if(v == par) continue;

            if(vis[v]){
                  low[node] = min(low[node] , in[v]);
            }
            else{
                  dfs(v , node) , child++;
                  if(par != -1 && low[v] >= in[node])
isArticulationPoint[node] = true;
                  low[node] = min(low[node] , low[v]);
            }
      }

      if(par == -1 && child > 1) isArticulationPoint[node] = true;
```

```
}

int main(){
	#ifndef ONLINE_JUDGE
		freopen("input.txt" , "r" , stdin);
		freopen("output.txt" , "w" , stdout);
	#endif

	int n , m;
	int a , b;

	cin>>n>>m;
	for(int i=1;i<=m;i++){
		cin>>a>>b;
		adj[a].push_back(b);
		adj[b].push_back(a);
	}

	dfs(1 , -1);

	for(int i=1;i<=n;i++)
		if(isArticulationPoint[i]) cout<<"Node "<<i<<" is articulation
point\n";
}
```

Time Complexity:

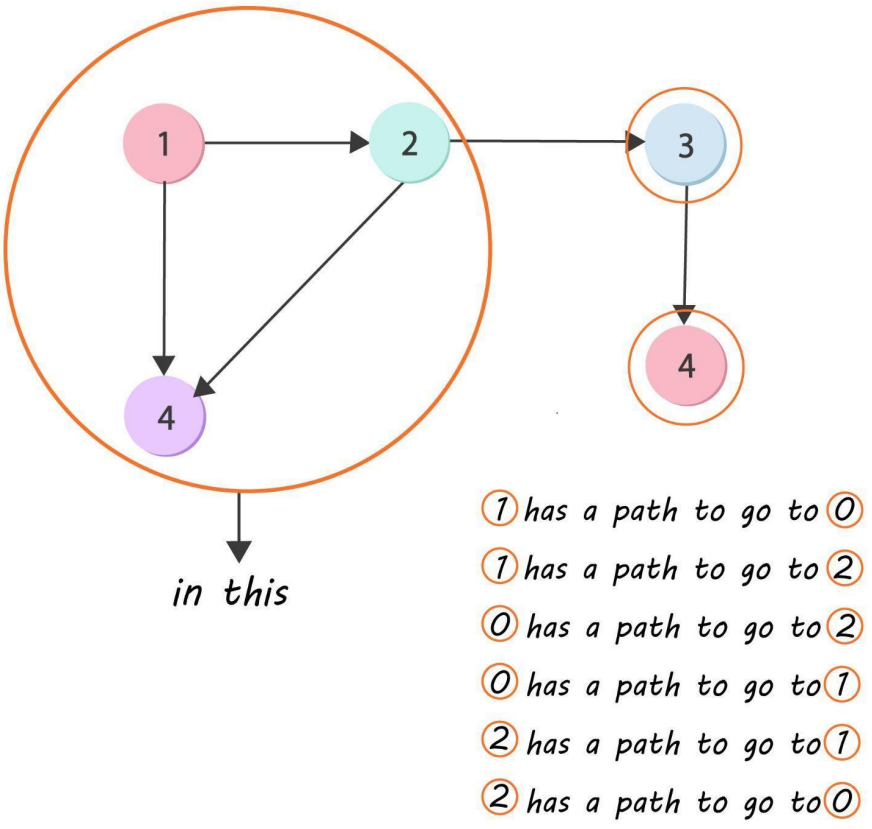
O(V+E), where V is the number of vertices and E is the number of edges in the graph.

Space Complexity:

O(V), where V is the number of vertices in the graph.

Strongly Connected Components

A directed graph is strongly connected if there is a path between all pairs of vertices. A strongly connected component (SCC) of a **directed graph** is a maximal strongly connected subgraph. For example, there are 3 SCCs in the following graph.



Problem statement:

You are given an unweighted directed graph of 'V' vertices and 'E' edges. Your task is to print all the strongly connected components (SCCs) present in the graph.

Approach:

Tarjan's Algorithm:

- Tarjan's algorithm can determine the SCCs using just a single DFS. Before understanding Tarjan's algorithm we must understand the meaning of low-link value and discovery time of a node.
- The discovery time of a node is the time at which the node was discovered during the DFS. This can be represented by an integer value which also specifies the order in which the nodes are visited during the DFS.
- The low-link value of a node is the smallest (lowest) discovery time reachable from that node during the DFS, including the node itself.
- For a SCC in a graph, every node which is a part of the SCC will have the same low-link value. But this may not be true all the time.
- As the discovery time of a node depends on the order in which the graph is traversed in a DFS, hence it may be possible that multiple nodes may have the same low-link value even though they are a part of different SCC.
- This is where Tarjan's SCC algorithm comes in. The algorithm maintains an invariant which prevents the low-link values of two SCCs from interfering with each other.
- The invariant used is a stack, which is different from the stack being used for DFS. The stack stores the nodes forming a subtree from which the SCC will be found. Nodes are placed in the stack in the order in which they are visited. But the nodes are removed only when a complete SCC is found.

Code:

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

const int N = 1e5;

int low[N] , in[N];
int timer;
bool vis[N] , onStack[N];
stack<int> st;

vector<int> adj[N];

void dfs(int node){
	in[node] = low[node] = timer++;
	vis[node] = onStack[node] = true;
	st.push(node);

	for(int v : adj[node]){
		if(vis[v]){
			if(onStack[v]) low[node] = min(low[node] , in[v]);
		}
		else{
			dfs(v);
			low[node] = min(low[node] , low[v]);
		}
	}

	if(in[node] == low[node]){
		while(st.top() != node){
			cout<<st.top()<<" ";
			onStack[st.top()] = false;
			st.pop();
		}

		cout<<st.top()<<" ";
		onStack[st.top()] = false;
		st.pop();

		cout<<endl;
	}

}
```

```
int main(){

        #ifndef ONLINE_JUDGE
        freopen("input.txt" , "r" , stdin);
        freopen("output.txt" , "w" , stdout);
        #endif

        int t ,n ,m;
        int a ,b;

        cin>>t;
        while(t--){
              cin>>n>>m;

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

              for(int i=0;i<m;i++){
                    cin>>a>>b;
                    adj[a].push_back(b);
              }

              for(int i=0;i<n;i++){
                    if(vis[i] == false) dfs(i);
              }
        }
}
```

Time Complexity:

O(V + E) per test case, where V is the number of vertices and E is the number of edges in the graph.

In the worst case, Tarjan's Algorithm performs a single DFS to obtain all the SCCs.

Space Complexity:

O(V) per test case.

In the worst case, O(V) space is required for storing the low-link values, discovery time, invariant stack and the recursion stack of DFS.

Edges In MST

Problem statement:

Your task is to determine the following for each edge of the given graph: whether it is either included in any MST, included at least in one MST, or not included in any MST.

Approach:

Let's take a look at Kruskal Algorithm which solve MST in O(m log m) time:-

- Sort the edges first in weight non-decreasing order
- Then process each edges. if the edge connects two different connected compoments, add this edge to MST then combine two compoments. Like DSU.
- The main point is that only those edges with same weight may replace each other in MST.
- First of all, sort edges as what Kruskal do. To get the answer, we construct MST in weight non-decreasing order, and process all edges with same weight together.
- Now on each step we are to face some edges with same weight x and a forest of connected compoments.

- Note that for an edge, what points it connects does not matter, we only need to know what compoments it connects.
- Now build a new graph G', each point in G' is a connected compoment in the original forest,and edges are added to connect compoments that it connected before.
- Let's answer queries on these edges. First of all, if an edge in G' is a loop(connects the same compoment), this edge must not appear in any MSTs.
- If after deleting an edge V in G', G's connectivity is changed (A connected compoment in G' spilt into two. We call these edges bridge), V must be in any of MST. All edges left can appear in some MSTs, but not any.
- Now we can simply use Tarjan's algorithm to get the answer quickly.

Code:

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

const int N = 1e5;

struct Edge{
      int a , b , w;
      int index;
};

//list of edges
Edge edges[N+1];

//for dsu
int _par[N+1];

//to store result
int result[N+1];
```

```
//for graph
vector<pair<int,int> > adj[N+1];

//for bridges
int in[N+1] , low[N+1] , timer;
bool vis[N+1];


bool comp(Edge a , Edge b){
      return a.w < b.w;
}

int find(int a){
      if(_par[a] == a) return a;

      return _par[a] = find(_par[a]);
}

void addEdge(int a , int b , int index){
      a = find(a);
      b = find(b);

      if(a == b) return;

      result[index] = 1;
      adj[a].push_back({b , index});
      adj[b].push_back({a , index});
}

void merge(int a , int b){
      a = find(a);
      b = find(b);

      adj[a].clear();
      adj[b].clear();

      vis[a] = vis[b] = false;

      //merge
      if(a != b) _par[a] = b;
}
```

```
void dfs(int node , int edgeIndex){
	in[node] = low[node] = timer++;
	vis[node] = true;

	for(pair<int,int> e : adj[node]){
		int index = e.second;
		int v = e.first;

		if(index == edgeIndex) continue;

		if(vis[v]){
			low[node] = min(low[node] , in[v]);
		}else{
			dfs(v , index);
			low[node] = min(low[node] , low[v]);
		}
	}

	if(edgeIndex != 0){
		if(in[node] == low[node]) result[edgeIndex] = 2;
	}

}

int main(){

	// #ifndef ONLINE_JUDGE
	// freopen("input.txt" , "r" , stdin);
	// freopen("output.txt" , "w" , stdout);
	// #endif

	int n , m;

	cin>>n>>m;

	for(int i=1;i<=n;i++) _par[i] = i;

	for(int i=1;i<=m;i++){
		cin>>edges[i].a>>edges[i].b>>edges[i].w;
		edges[i].index = i;
```

```
	}

	sort(edges + 1 , edges + m + 1 , comp);

	int i = 1;
	while(i <= m){

		int j;
		//STEP 1
		for(j=i; edges[i].w == edges[j].w ; j++)
			addEdge(edges[j].a , edges[j].b , edges[j].index);


		//STEP 2 : find all bridges
		timer = 0;
		for(j=i; edges[i].w == edges[j].w ; j++) {
			int node = find(edges[j].a);
			if(vis[node] == false) dfs(node , 0);
		}

		//STEP 3 : merge and remove edges
		for(j=i; edges[i].w == edges[j].w ; j++) merge(edges[j].a ,
edges[j].b);

		i = j;

	}

	for(int i=1;i<=m;i++){
		if(result[i] == 0) cout<<"none\n";
		else
		if(result[i] == 1) cout<<"at least one\n";
		else 			cout<<"any\n";
	}
}
```

Time Complexity:

O(N*log(N)) , where N is the number of nodes in the tree.

Space Complexity:

O(N), Where N is the number of vertex.