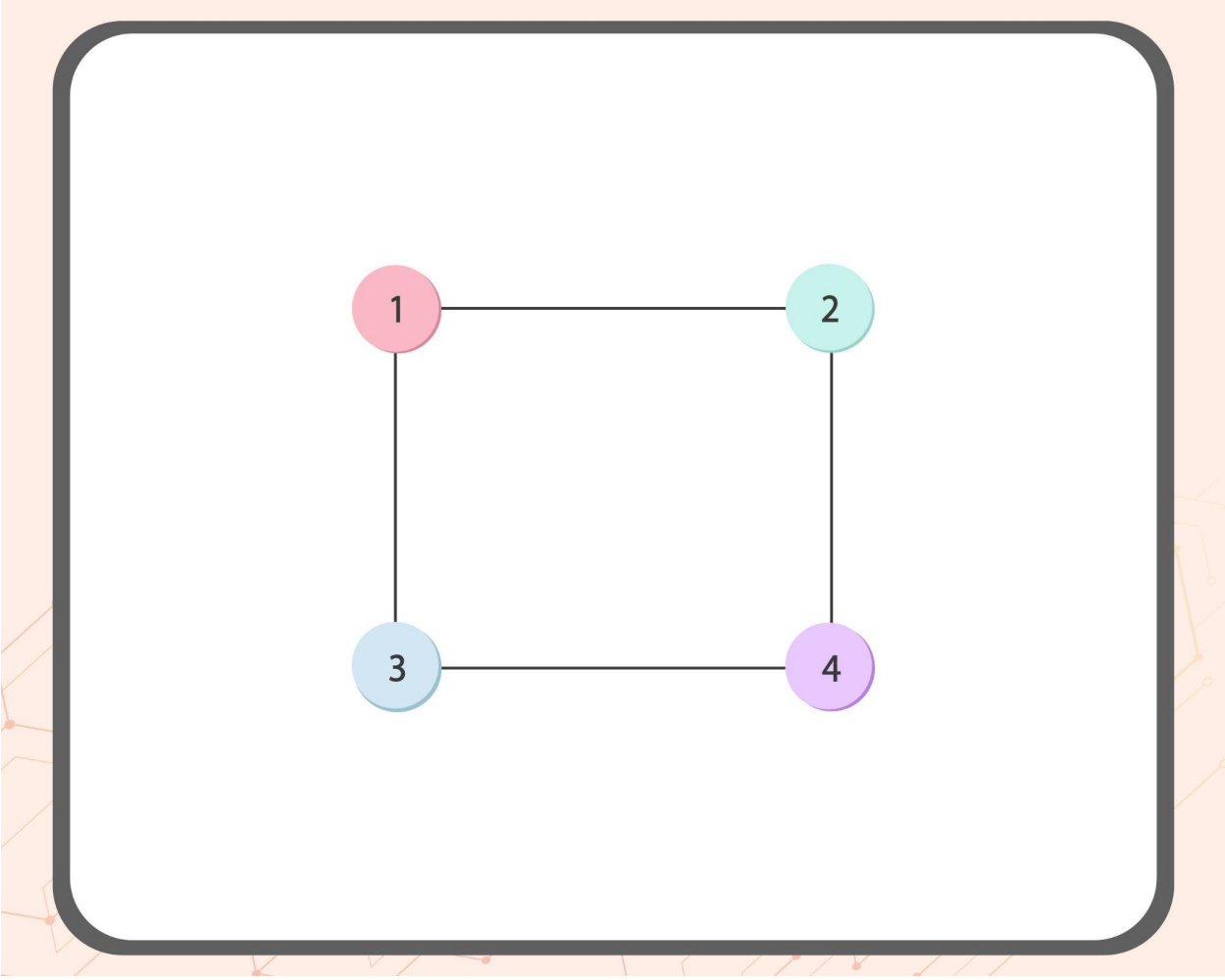
Advance Graphs

Connected Components

Problem Statement: We have to find out the number of connected components in an undirected graph.

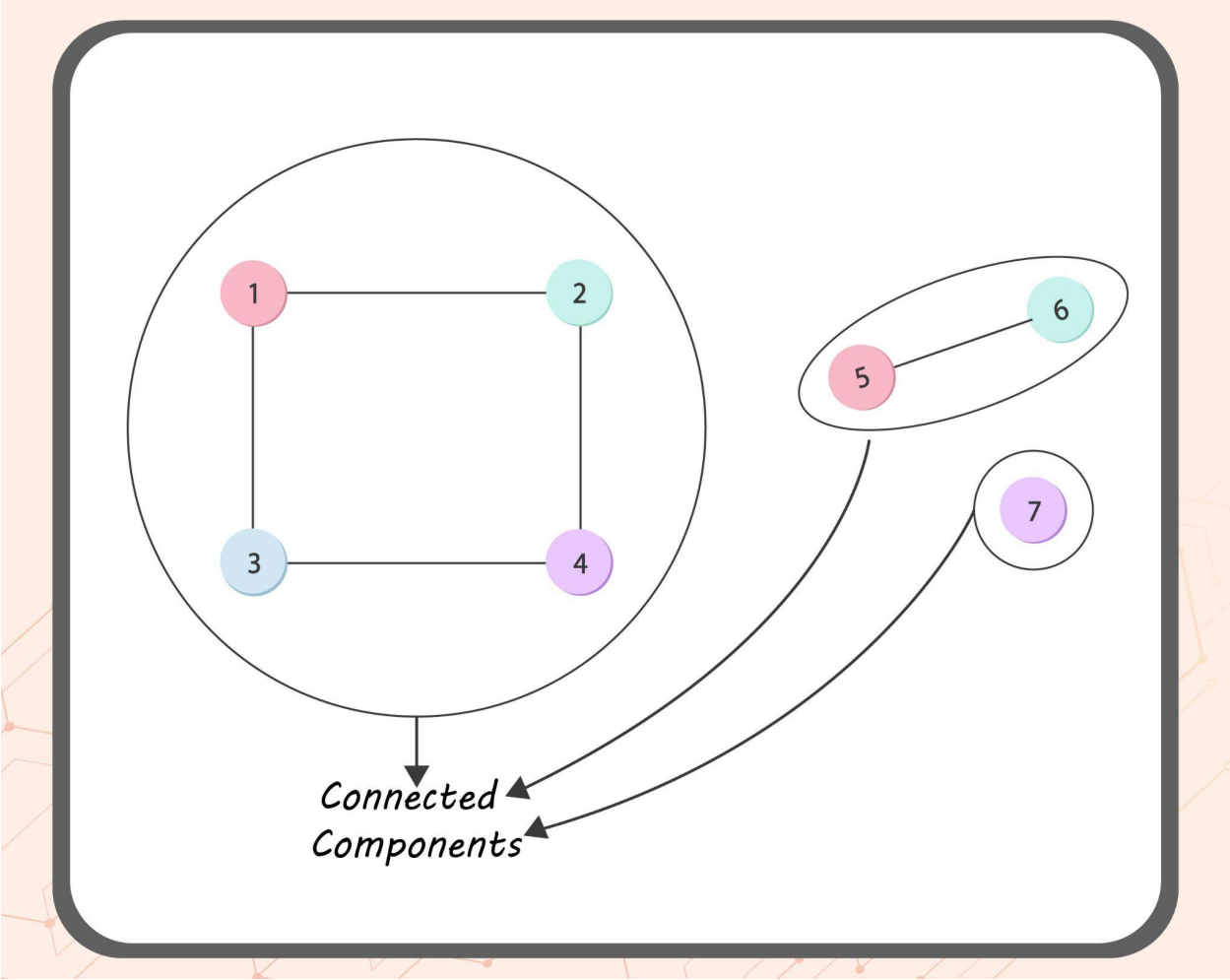
Explanation:

Let us consider an undirected graph



In the above graph there is a path from every vertex to every other vertex, therefore this is a connected graph. In this case we say that the graph has only one connected component.

There may be graphs which do not satisfy the above conditions and are called disconnected graphs. However in such cases, there may still be several components which are connected within themselves. For example,



In the above figure vertices 1, 2, 3, 4 form a connected component. Another connected component is formed by vertices 5, 6 and vertex 7 itself is a connected component.

To find the connected components in an undirected graph, we can use either DFS or BFS for this task.

Let's take a look at the algorithm that uses DFS to solve this problem:

```
Data: Given an undirected graph G (V, E)
Result: Number of Connected Components
Component_Count = 0;
for each vertex k ϵ V do
   if Visited[k] == False;
end
for each vertex k ϵ V do
   if Visited[k] == False  then
      DFS (V, k);
      Component_Count = Component_Count + 1;
   end
end
Print Component_Count;
Procedure DFS (V, k)
Visited[k] = True;
for each Vertex P ϵ V.Adj[k] do
   if Visited[p] == False  then
      DFS (V, p);
   end
end
```

We have already taken a look at how DFS and BFS work in the previous lectures along with their codes.

Permutations Swap

Problem Statement:

Kevin has a permutation P of N integers 1, 2, ..., N, but he doesn't like it. Kevin wants to get a permutation Q.

He also believes that there are M good pairs of integers (ai, bi). Kevin can perform following operation with his permutation:

Swap Px and Py only if (x, y) is a good pair.

Help him and tell if Kevin can obtain permutation Q using such operations.

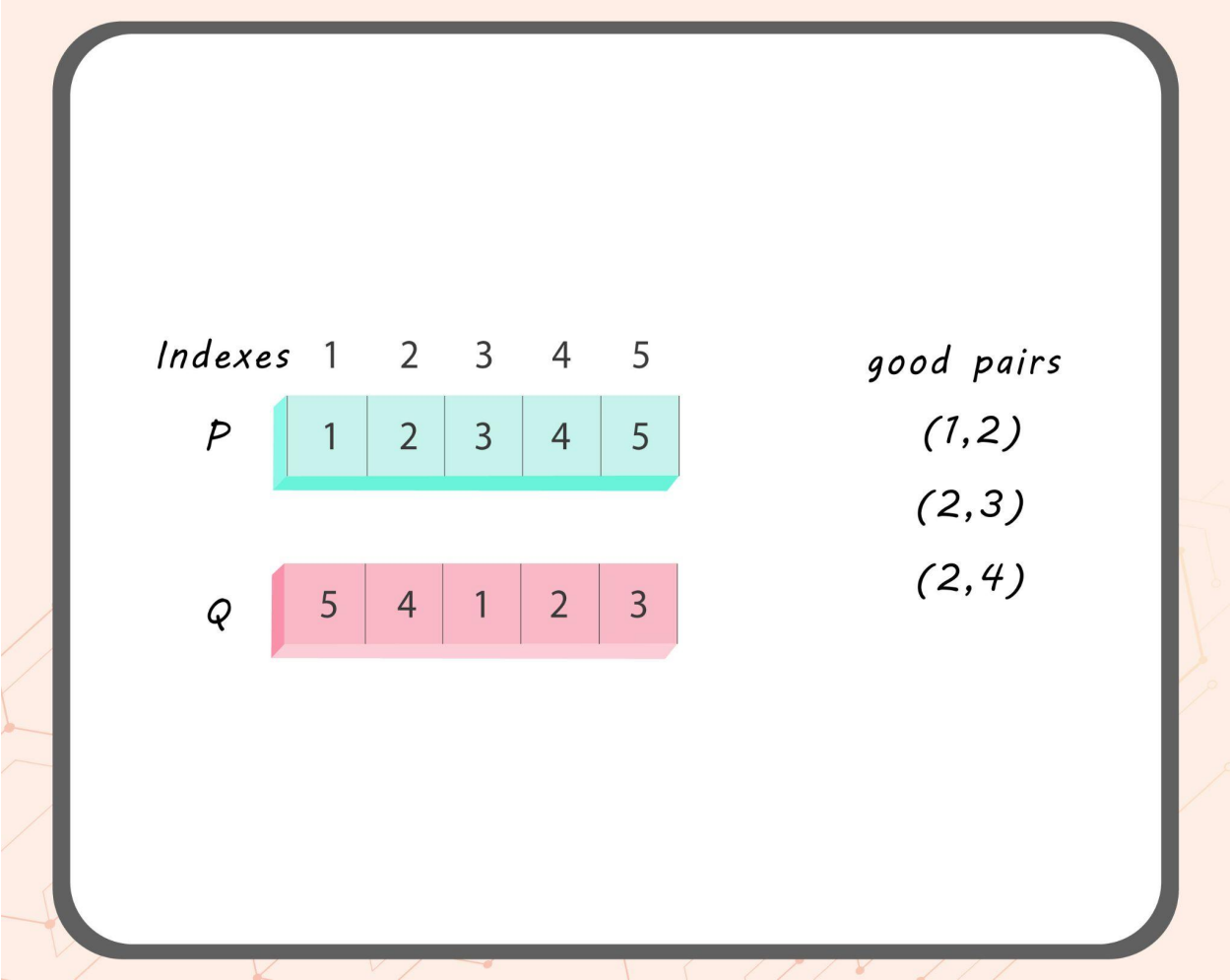
Explanation:

Let's take an example to better understand the problem statement.

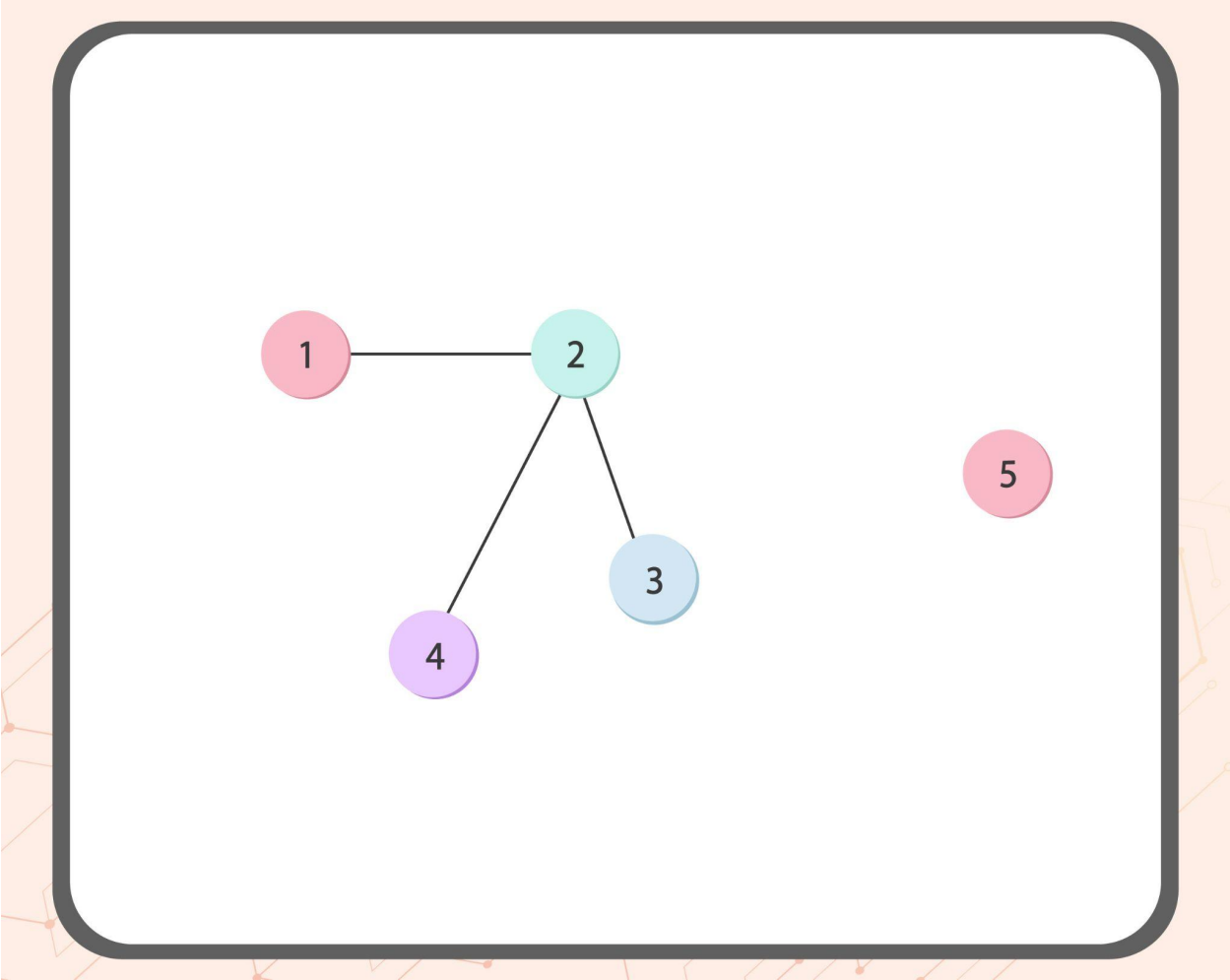
We are given N= 5 numbers **{1, 2, 3, 4, 5}** and we also have permutation **P = {1, 2, 3, 4, 5}** and permutation **Q = {5, 4, 1, 2, 3}** of these N numbers.

We are also given some pairs of numbers **{(1, 2), (2, 3), (2, 4)}**. Each number in a pair represents that the two indexes or positions can be switched with each other.

We need to tell whether permutation P can be converted to permutation Q.



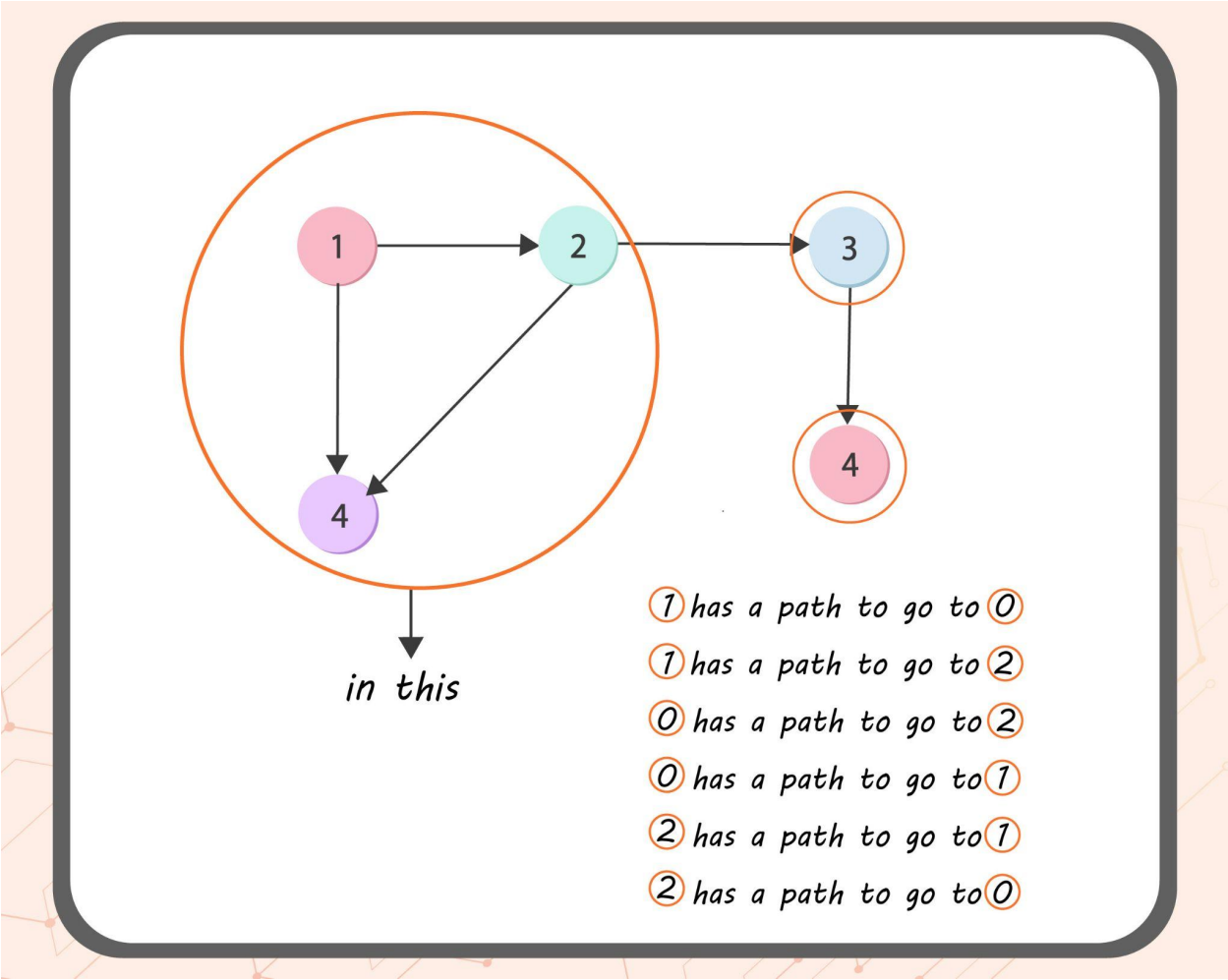
This problem can be approached using the concept of hasPath. This basically means that if we are given a pair (1, 2), then these can be considered as the vertices in a graph and since they are a pair that means there exists an edge between them. SImilarly there exists an edge between (2, 3) and (2, 4). So the corresponding graph for the above problem can be represented as the following figure:



Now, even if we can't directly perform some operations we can indirectly do them. For example, if we have to swap indexes 1 and 3 then they are not in our given pairs but what we could do is first perform swap on index 1 and 2 and then perform swap on index 2 and 3. So overall there needs to exist a path between the indexes (vertices) for swap operation.

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.

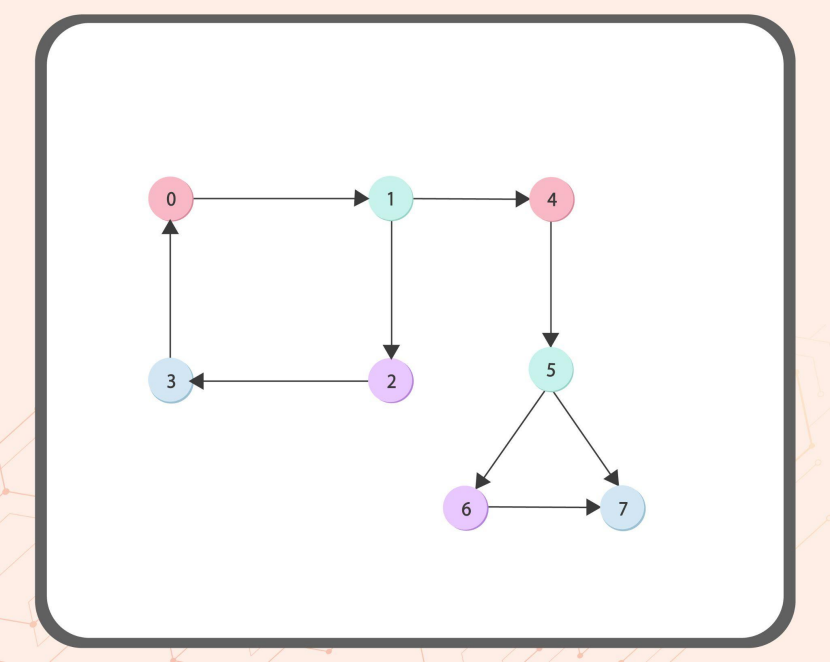


Kosaraju's Algorithm

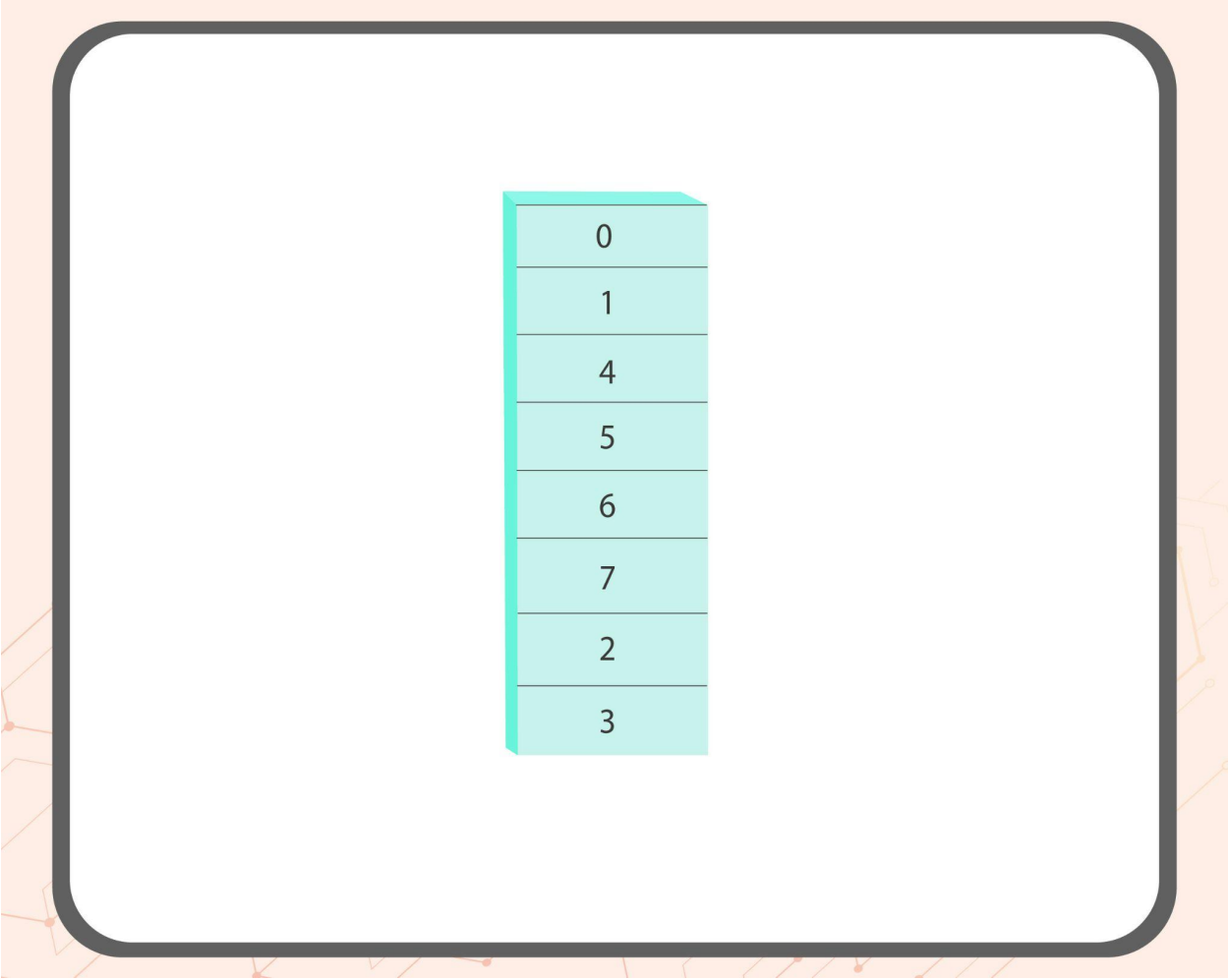
Kosaraju's algorithm is used to find strongly connected components in a graph. The algorithm consists of three steps:

1. **Do a DFS on the original graph**: Do a DFS on the original graph, keeping track of the finish times of each node. This can be done using a stack, when a DFS finishes put the source vertex on the stack. This way the node with the highest finishing time will be on top of the stack.
2. **Reverse the original graph**: Reverse the graph using an adjacency list.
3. **Do DFS again**: Do DFS on the reversed graph, with the source vertex as the vertex on top of the stack. When DFS finishes, all nodes visited will form one Strongly Connected Component. If any more nodes remain unvisited, this means there are more Strongly Connected Components, so pop vertices from top of the stack until a valid unvisited node is found. This will have the highest finishing time of all currently unvisited nodes. This step is repeated until all nodes are visited.

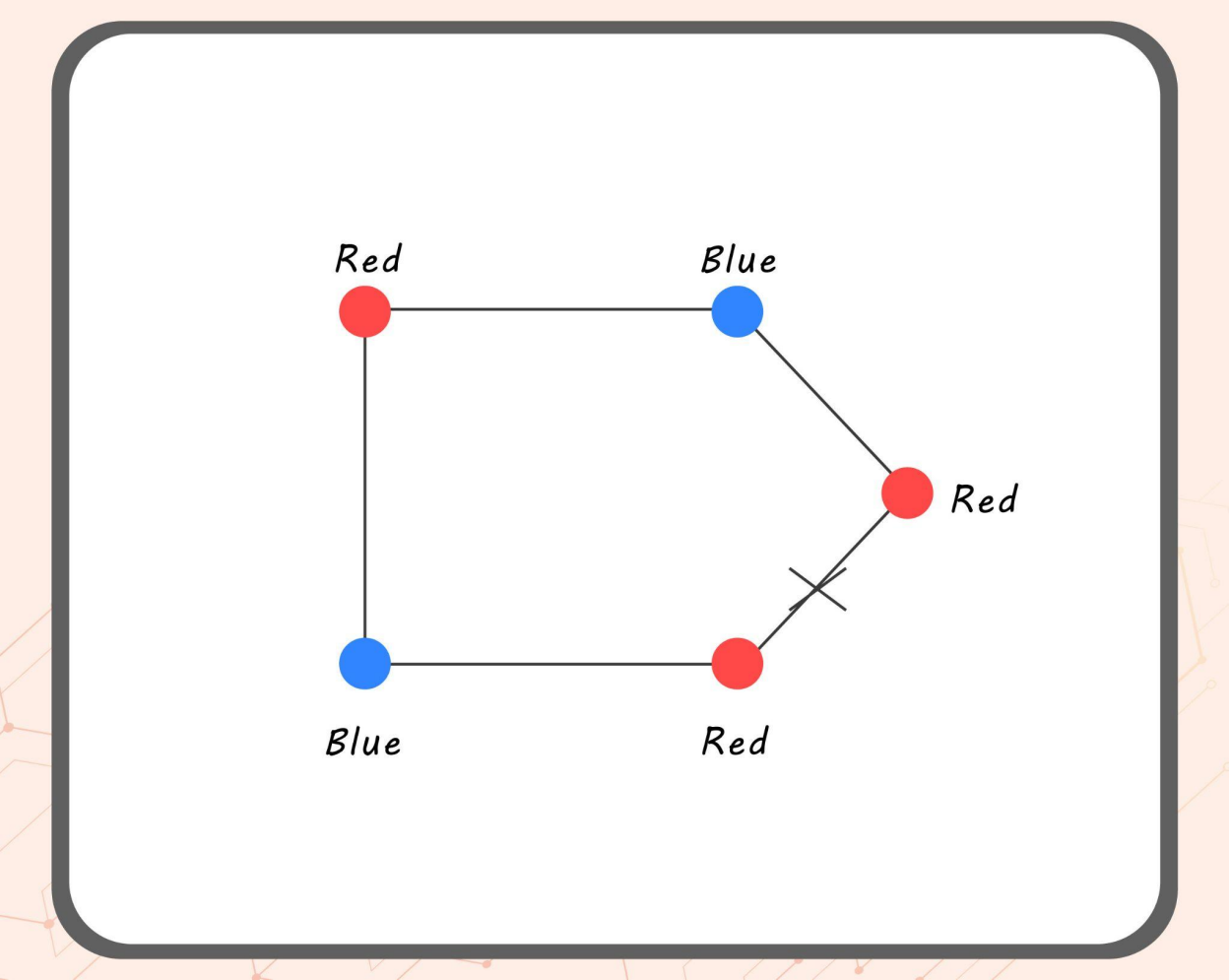
Let us take a look at an example to see the working of the above algorithm:



1. After applying DFS on this graph starting from node 0, and keeping track of the finishing time of each node with the help of the stack, our stack becomes



2. Now reversing the original graph gives us



3. Pop every element out of the stack. If it's not visited, apply DFS on it and that will give us the SCC.

   Step 1. We pop an element at the top that is node 0. We apply DFS(0) , hence it will visit node 1,2,3. Now we mark them as visited and this forms our first group.
   Step 2. We pop the second element which is 1 but since it is visited we skip this.
   Step 3. We pop the third element which is 4. Since it is not visited we apply DFS(4). Now it will visit only 4 as there are no adjacent nodes. Hence 4 is a separate group.

Step 4. We pop the fourth element which is 5. Since it is not visited we apply DFS(5). It will visit 5,6,7. Hence 5,6,7 will form a separate group.

Step 5. We pop the fifth element which is 6 , since it is already visited we skip. Now we pop the sixth element which is 7 , since it is already visited we skip it. Similarly we skip seventh and eight elements which are 2 and 3 as they are already visited.

Now , the stack is empty and our algorithm stops.

The nodes printed are:

0 3 2 1

4

5 7 6

Code:

```
#include <iostream>
#include <vector>
#include <stack>
#include <unordered_set>
using namespace std;

void dfs(vector<int>* edges, int start, unordered_set<int> &visited,
stack<int> &finishStack) {
      visited.insert(start);
      for (int i = 0; i < edges[start].size(); i++) {
            int adjacent = edges[start][i];
            if (visited.count(adjacent) == 0) {
                  dfs(edges, adjacent, visited, finishStack);
            }
      }
      finishStack.push(start);
}

void dfs2(vector<int>* edges, int start, unordered_set<int>* component,
unordered_set<int> & visited) {
```

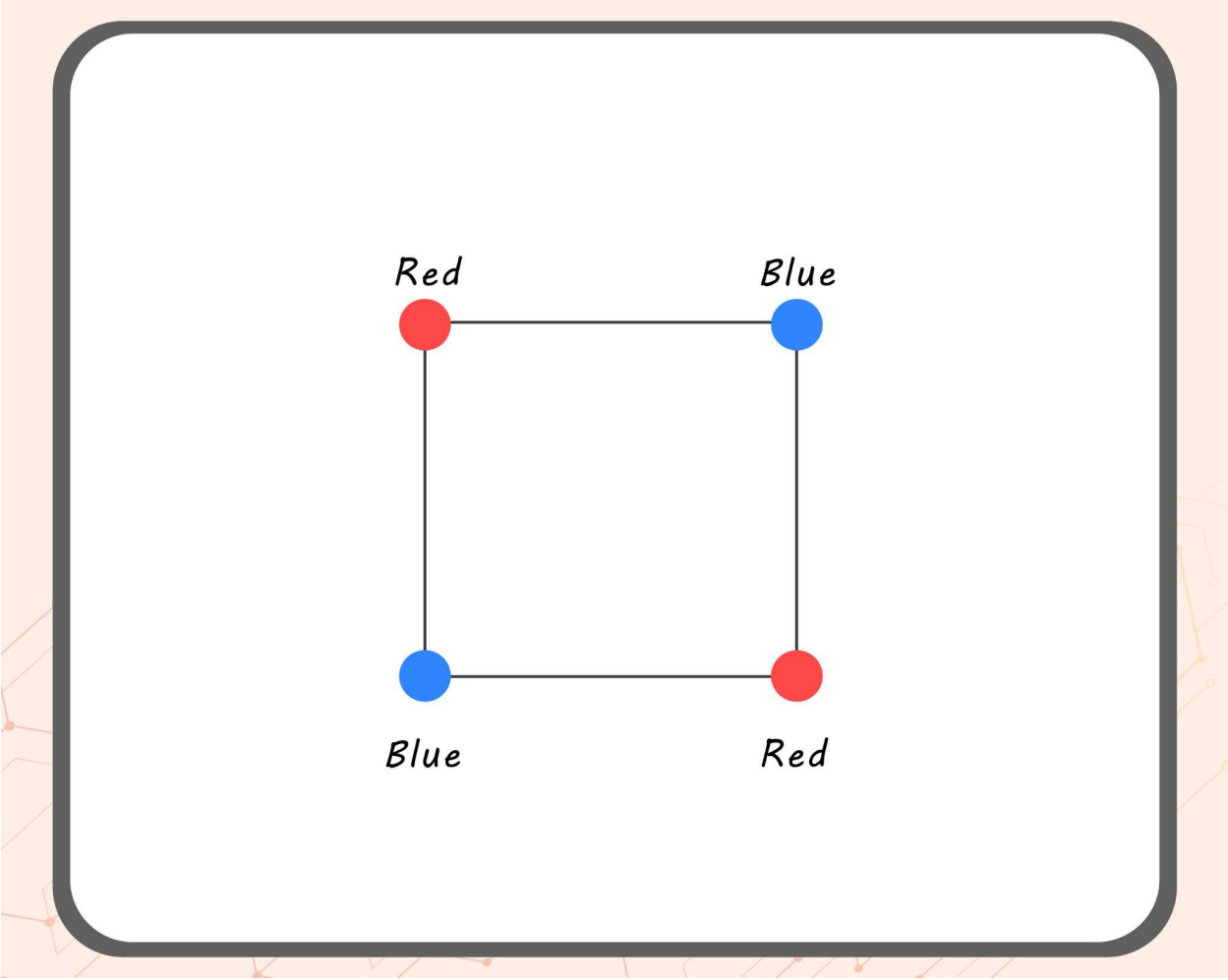
```
	visited.insert(start);
	component->insert(start);
	for (int i = 0; i < edges[start].size(); i++) {
		int adjacent = edges[start][i];
		if (visited.count(adjacent) == 0) {
			dfs2(edges, adjacent, component, visited);
		}
	}
}

unordered_set<unordered_set<int>*>* getSCC(vector<int>* edges, vector<int>*
edgesT, int n) {
	unordered_set<int> visited;
	stack<int> finishedVertices;
	for (int i = 0; i < n; i++) {
		if (visited.count(i) == 0) {
			dfs(edges, i, visited, finishedVertices);
		}
	}
	unordered_set<unordered_set<int>*>* output = new
unordered_set<unordered_set<int>*>();
	visited.clear();
	while (finishedVertices.size() != 0) {
		int element = finishedVertices.top();
		finishedVertices.pop();
		if (visited.count(element) != 0) {
			continue;
		}
		unordered_set<int>* component = new unordered_set<int>();
		dfs2(edgesT, element, component, visited);
		output->insert(component);
	}
	return output;
}

int main() {
	int n;
	cin >> n;
	vector<int>* edges = new vector<int>[n];
	vector<int>* edgesT = new vector<int>[n];
```

```
	int m;
	cin >> m;
	for (int i = 0; i < m; i++) {
		int j, k;
		cin >> j >> k;
		edges[j - 1].push_back(k - 1);
		edgesT[k - 1].push_back(j - 1);
	}
	unordered_set<unordered_set<int>*>* components = getSCC(edges,
edgesT, n);
	unordered_set<unordered_set<int>*>::iterator it =
components->begin();
	while (it != components->end()) {
		unordered_set<int>* component = *it;
		unordered_set<int>::iterator it2 = component->begin();
		while (it2 != component->end()) {
			cout << *it2  + 1<< " ";
			it2++;
		}
		cout << endl;
		delete component;
		it++;
	}
	delete components;
	delete [] edges;
	delete [] edgesT;
}
```

Bipartite Graphs

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.

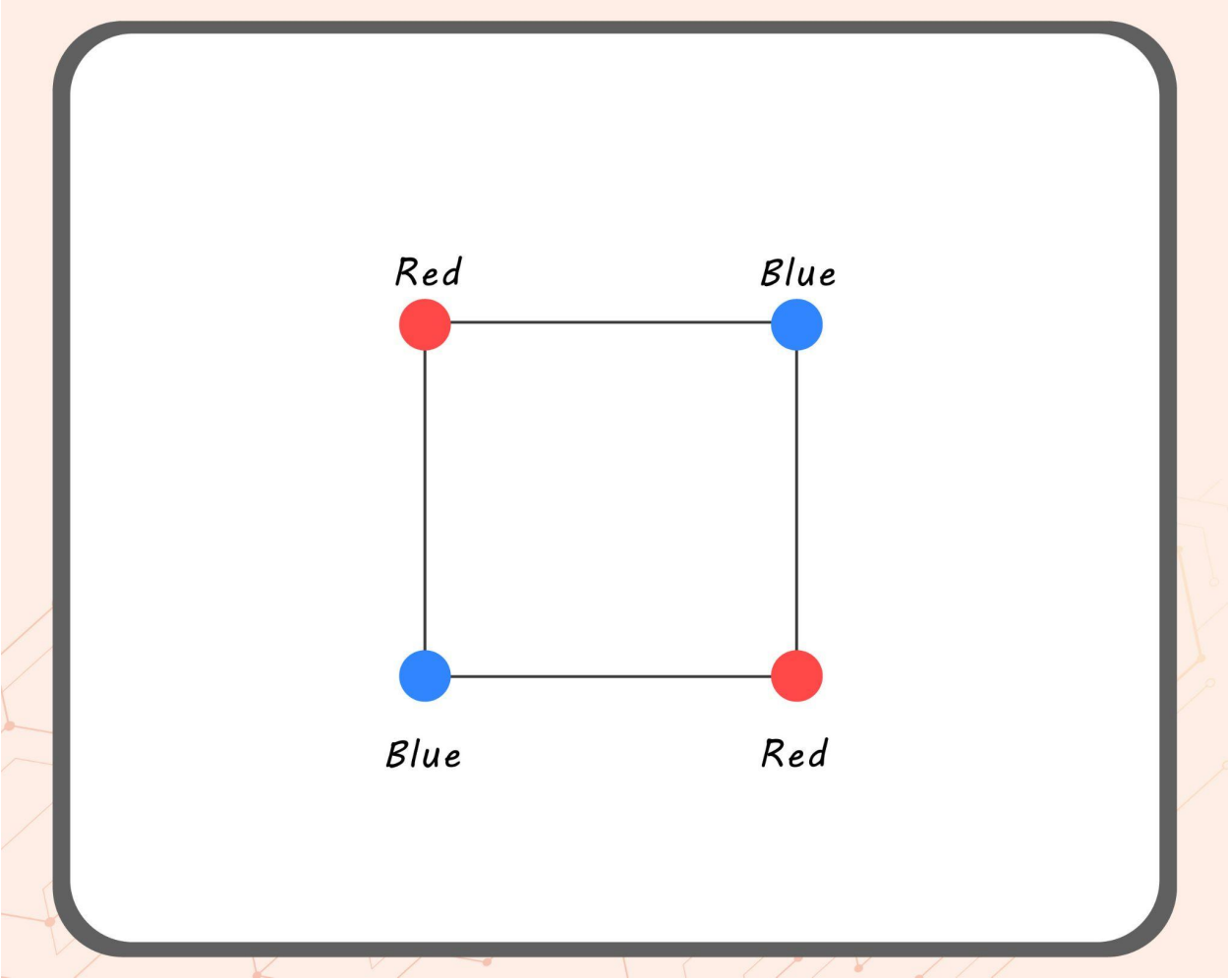
Let's take an example to understand the concept better



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.

To find whether a graph is bipartite or not, we follow the following steps:

1. We are going to maintain two sets and a queue.

2. Now, we are going to pick a starting vertex and place it in any of the sets (let's say set1) and we are going to push it in the queue.
3. Now, pop the vertex from the queue and place its neighbours in set2 and the queue.
4. On popping the next vertex from the queue, we have to check whether any of its neighbours is present in set2 or not.
5. If the 4th statement returns true, then the graph is not bipartite.

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

```
#include <iostream>
#include <vector>
#include <unordered_set>
using namespace std;

bool bipartite(vector<int>* edges, int n) {
	if (n == 0) {
		return true;
	}
	unordered_set<int> sets[2];
	vector<int> pending;
	sets[0].insert(0);
	pending.push_back(0);
	while (pending.size() > 0) {
		int current = pending.back();
		pending.pop_back();
		int currentSet = sets[0].count(current) > 0 ? 0 : 1;
		for (int i = 0; i < edges[current].size(); i++) {
			int neighbor = edges[current][i];
			if (sets[0].count(neighbor) == 0 &&
sets[1].count(neighbor) == 0) {
				sets[1 - currentSet].insert(neighbor);
				pending.push_back(neighbor);
			} else if (sets[currentSet].count(neighbor) > 0) {
				return false;
```

```
                }
            }
        }
        return true;
}

int main() {
        while (true) {
            int n;
            cin >> n;
            if (n == 0)
                break;
            vector<int>* edges = new vector<int>[n];
            int m;
            cin >> m;
            for (int i = 0; i < m; i++) {
                int j, k;
                cin >> j >> k;
                edges[j].push_back(k);
                edges[k].push_back(j);
            }
            bool ans = bipartite(edges, n);
            delete [] edges;
            if (ans) {
                cout << "BICOLORABLE." << endl;
            } else {
                cout << "NOT BICOLORABLE." << endl;
            }
        }
}
```