**Find if there is a path between two vertices in a directed graph**

Given a Directed Graph and two vertices in it, check whether there is a path from the first given vertex to second.   
**Example:**

**Consider the following Graph:**

**Input :** (u, v) = (1, 3)

**Output:** Yes

**Explanation:** There is a path from 1 to 3, 1 -> 2 -> 3

**Input :** (u, v) = (3, 6)

**Output:** No

**Explanation:** There is no path from 3 to 6

**Approach:** Either [Breadth First Search (BFS)](https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/) or [Depth First Search (DFS)](https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/) can be used to find path between two vertices. Take the first vertex as source in BFS (or DFS), follow the standard BFS (or DFS). If the second vertex is found in our traversal, then return true else return false.  
**Algorithm:**

1. The implementation below is using BFS.
2. Create a queue and a visited array initially filled with 0, of size V where V is number of vertices.
3. Insert the starting node in the queue, i.e. push u in the queue and mark u as visited.
4. Run a loop until the queue is not empty.
5. Dequeue the front element of the queue. Iterate all its adjacent elements. If any of the adjacent element is the destination return true. Push all the adjacent and unvisited vertices in the queue and mark them as visited.
6. Return false as the destination is not reached in BFS.

<https://github.com/hareramcse/Datastructure/blob/master/Graph/src/com/hs/connectivity/FindIfThereIsPathBetweenTwoVertices.java>

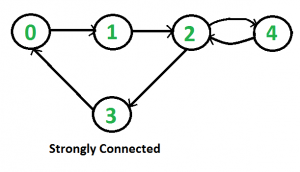
**Complexity Analysis:**

* **Time Complexity:**O(V+E) where V is number of vertices in the graph and E is number of edges in the graph.
* **Space Compelxity:** O(V).   
  There can be atmost V elements in the queue. So the space needed is O(V).

**Trade-offs between BFS and DFS:** Breadth-First search can be useful to find the shortest path between nodes, and depth-first search may traverse one adjacent node very deeply before ever going into immediate neighbours.

**Check if a graph is strongly connected**

Given a directed graph, find out whether the graph is strongly connected or not**. A directed graph is strongly connected if there is a path between any two pair of vertices**. For example, following is a strongly connected graph.



**It is easy for undirected graph**, we can just do a BFS and DFS starting from any vertex. If BFS or DFS visits all vertices, then the given undirected graph is connected. This approach won’t work for a directed graph. For example, consider the following graph which is not strongly connected. If we start DFS (or BFS) from vertex 0, we can reach all vertices, but if we start from any other vertex, we cannot reach all vertices.



**How to do for directed graph?**

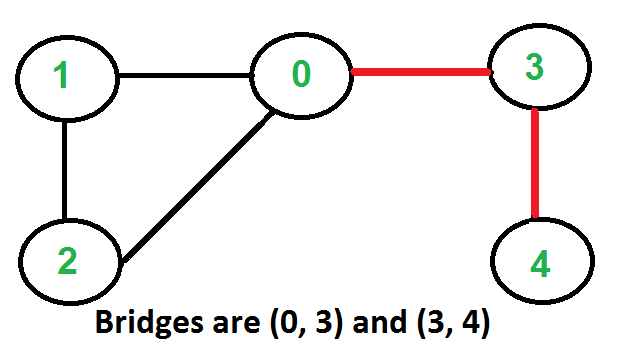
A simple idea is to use a all pair shortest path algorithm like [**Floyd Warshall**](https://www.geeksforgeeks.org/dynamic-programming-set-16-floyd-warshall-algorithm/)**or find**[**Transitive Closure**](https://www.geeksforgeeks.org/transitive-closure-of-a-graph/) of graph. Time complexity of this method would be O(v3).  
We can also **do**[**DFS**](https://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/)**V times** starting from every vertex. If any DFS, doesn’t visit all vertices, then graph is not strongly connected. This algorithm takes O(V\*(V+E)) time which can be same as transitive closure for a dense graph.  
A better idea can be [**Strongly Connected Components (SCC)**](https://www.geeksforgeeks.org/strongly-connected-components/)**algorithm**. We can find all SCCs in O(V+E) time. If number of SCCs is one, then graph is strongly connected. The algorithm for SCC does extra work as it finds all SCCs.   
Following is **Kosaraju’s DFS based simple algorithm that does two DFS traversals** of graph:   
**1)** Initialize all vertices as not visited.  
**2)** Do a DFS traversal of graph starting from any arbitrary vertex v. If DFS traversal doesn’t visit all vertices, then return false.  
**3)** Reverse all arcs (or find transpose or reverse of graph)   
**4)** Mark all vertices as not-visited in reversed graph.  
**5)** Do a DFS traversal of reversed graph starting from same vertex v (Same as step 2). If DFS traversal doesn’t visit all vertices, then return false. Otherwise return true.  
The idea is, if every node can be reached from a vertex v, and every node can reach v, then the graph is strongly connected. In step 2, we check if all vertices are reachable from v. In step 4, we check if all vertices can reach v (In reversed graph, if all vertices are reachable from v, then all vertices can reach v in original graph).

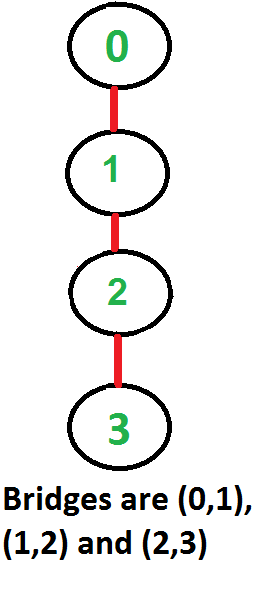
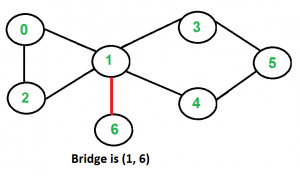
<https://github.com/hareramcse/Datastructure/blob/master/Graph/src/com/hs/connectivity/CheckIfGraphIsStronglyConnected.java>

**Bridges in a graph**

An edge in an undirected connected graph is a bridge iff removing it disconnects the graph. For a disconnected undirected graph, definition is similar, a bridge is an edge removing which increases number of disconnected components.   
Like [Articulation Points](https://www.geeksforgeeks.org/articulation-points-or-cut-vertices-in-a-graph/), bridges represent vulnerabilities in a connected network and are useful for designing reliable networks. For example, in a wired computer network, an articulation point indicates the critical computers and a bridge indicates the critical wires or connections.

Following are some example graphs with bridges highlighted with red color.





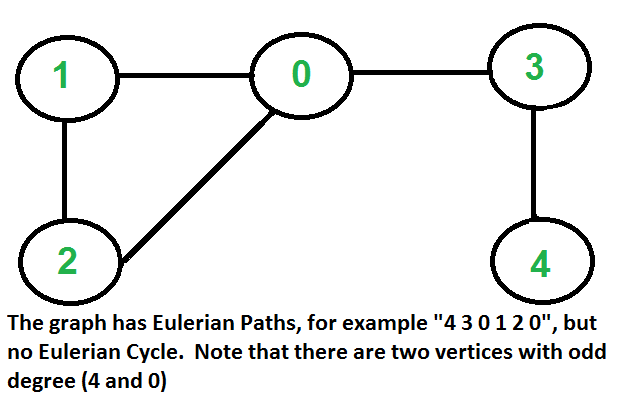
**How to find all bridges in a given graph?**   
A simple approach is to one by one remove all edges and see if removal of an edge causes disconnected graph. Following are steps of simple approach for connected graph.  
1) For every edge (u, v), do following   
…..a) Remove (u, v) from graph   
..…b) See if the graph remains connected (We can either use BFS or DFS)   
…..c) Add (u, v) back to the graph.  
Time complexity of above method is O(E\*(V+E)) for a graph represented using adjacency list. Can we do better?

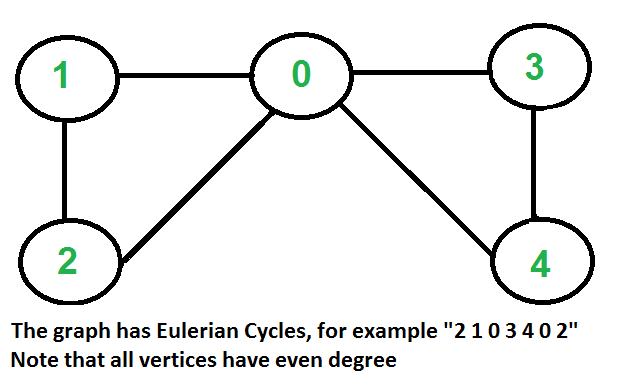
**A O(V+E) algorithm to find all Bridges**   
The idea is similar to [O(V+E) algorithm for Articulation Points](https://www.geeksforgeeks.org/articulation-points-or-cut-vertices-in-a-graph/). We do DFS traversal of the given graph. In DFS tree an edge (u, v) (u is parent of v in DFS tree) is bridge if there does not exist any other alternative to reach u or an ancestor of u from subtree rooted with v. As discussed in the [previous post](https://www.geeksforgeeks.org/articulation-points-or-cut-vertices-in-a-graph/), the value low[v] indicates earliest visited vertex reachable from subtree rooted with v. *The condition for an edge (u, v) to be a bridge is, “low[v] > disc[u]”*.

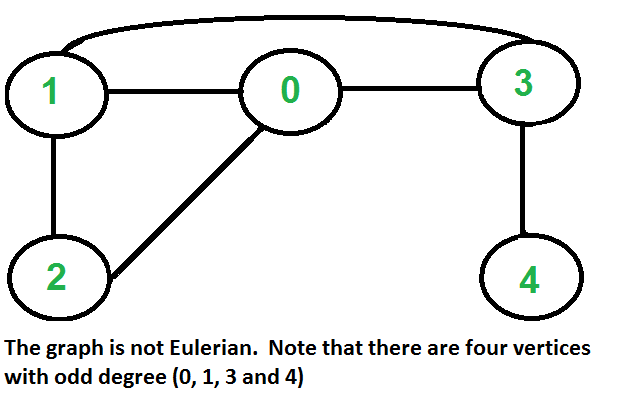
<https://github.com/hareramcse/Datastructure/blob/master/Graph/src/com/hs/connectivity/FindBridgeInGraph.java>

**Eulerian path and circuit for undirected graph**

[Eulerian Path](http://en.wikipedia.org/wiki/Eulerian_path)is a path in graph that visits every edge exactly once. Eulerian Circuit is an Eulerian Path which starts and ends on the same vertex. 







**How to find whether a given graph is Eulerian or not?**   
The problem is same as following question. “Is it possible to draw a given graph without lifting pencil from the paper and without tracing any of the edges more than once”.  
A graph is called Eulerian if it has an Eulerian Cycle and called Semi-Eulerian if it has an Eulerian Path. The problem seems similar to [Hamiltonian Path](https://www.geeksforgeeks.org/backtracking-set-7-hamiltonian-cycle/) which is NP complete problem for a general graph. Fortunately, we can find whether a given graph has a Eulerian Path or not in polynomial time. In fact, we can find it in O(V+E) time.   
Following are some interesting properties of undirected graphs with an Eulerian path and cycle. We can use these properties to find whether a graph is Eulerian or not.  
**Eulerian Cycle**   
An undirected graph has Eulerian cycle if following two conditions are true.   
….a) All vertices with non-zero degree are connected. We don’t care about vertices with zero degree because they don’t belong to Eulerian Cycle or Path (we only consider all edges).   
….b) All vertices have even degree.  
**Eulerian Path**   
An undirected graph has Eulerian Path if following two conditions are true.   
….a) Same as condition (a) for Eulerian Cycle   
….b) If zero or two vertices have odd degree and all other vertices have even degree. Note that only one vertex with odd degree is not possible in an undirected graph (sum of all degrees is always even in an undirected graph)  
Note that a graph with no edges is considered Eulerian because there are no edges to traverse.  
**How does this work?**   
In Eulerian path, each time we visit a vertex v, we walk through two unvisited edges with one end point as v. Therefore, all middle vertices in Eulerian Path must have even degree. For Eulerian Cycle, any vertex can be middle vertex, therefore all vertices must have even degree.

<https://github.com/hareramcse/Datastructure/blob/master/Graph/src/com/hs/connectivity/EulerianPathForUnDirectedGraph.java>

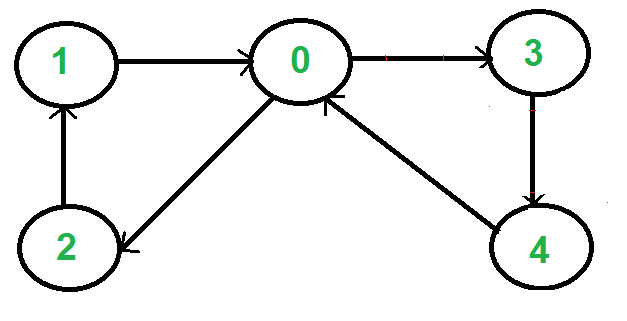
<https://github.com/hareramcse/Datastructure/blob/master/Graph/src/com/hs/connectivity/EulerianPathInUnDirectedGraph.java>

**Euler Circuit in a Directed Graph**

[Eulerian Path](http://en.wikipedia.org/wiki/Eulerian_path) is a path in graph that visits every edge exactly once. Eulerian Circuit is an Eulerian Path which starts and ends on the same vertex.

A graph is said to be eulerian if it has a eulerian cycle. We have discussed [eulerian circuit for an undirected graph](https://www.geeksforgeeks.org/eulerian-path-and-circuit/). In this post, the same is discussed for a directed graph.

For example, the following graph has eulerian cycle as {1, 0, 3, 4, 0, 2, 1}



**How to check if a directed graph is eulerian?**   
A directed graph has an eulerian cycle if following conditions are true (Source: [Wiki](http://en.wikipedia.org/wiki/Eulerian_path#Properties))   
1) All vertices with nonzero degree belong to a single [strongly connected component](https://www.geeksforgeeks.org/strongly-connected-components/).   
2) In degree is equal to the out degree for every vertex.

We can detect singly connected component using [Kosaraju’s DFS based simple algorithm](https://www.geeksforgeeks.org/connectivity-in-a-directed-graph/).

To compare in degree and out-degree, we need to store in degree and out-degree of every vertex. Out degree can be obtained by the size of an adjacency list. In degree can be stored by creating an array of size equal to the number of vertices.

<https://github.com/hareramcse/Datastructure/blob/master/Graph/src/com/hs/connectivity/EulerCircuitDirectedGraph.java>

**Transitive closure of a graph**

Given a directed graph, find out if a vertex j is reachable from another vertex i for all vertex pairs (i, j) in the given graph. Here reachable mean that there is a path from vertex i to j. The reach-ability matrix is called the transitive closure of a graph.

For example, consider below graph

transitiveclosure

Transitive closure of above graphs is

1 1 1 1

1 1 1 1

1 1 1 1

0 0 0 1

The graph is given in the form of adjacency matrix say ‘graph[V][V]’ where graph[i][j] is 1 if there is an edge from vertex i to vertex j or i is equal to j, otherwise graph[i][j] is 0.  
[Floyd Warshall Algorithm](https://www.geeksforgeeks.org/floyd-warshall-algorithm-dp-16/) can be used, we can calculate the distance matrix dist[V][V] using [Floyd Warshall](https://www.geeksforgeeks.org/floyd-warshall-algorithm-dp-16/), if dist[i][j] is infinite, then j is not reachable from I. Otherwise, j is reachable and the value of dist[i][j] will be less than V.   
Instead of directly using Floyd Warshall, we can optimize it in terms of space and time, for this particular problem. Following are the optimizations:

1. Instead of an integer resultant matrix ([dist[V][V] in floyd warshall](https://www.geeksforgeeks.org/floyd-warshall-algorithm-dp-16/)), we can create a boolean reach-ability matrix reach[V][V] (we save space). The value reach[i][j] will be 1 if j is reachable from i, otherwise 0.
2. Instead of using arithmetic operations, we can use logical operations. For arithmetic operation ‘+’, logical and ‘&&’ is used, and for a min, logical or ‘||’ is used. (We save time by a constant factor. Time complexity is the same though)

<https://github.com/hareramcse/Datastructure/blob/master/Graph/src/com/hs/connectivity/TransitiveClosure.java>

**Find the number of islands | Set 1 (Using DFS)**

Given a boolean 2D matrix, find the number of islands. A group of connected 1s forms an island. For example, the below matrix contains 5 islands  
**Example:** 

Input : mat[][] = {{1, 1, 0, 0, 0},

{0, 1, 0, 0, 1},

{1, 0, 0, 1, 1},

{0, 0, 0, 0, 0},

{1, 0, 1, 0, 1}}

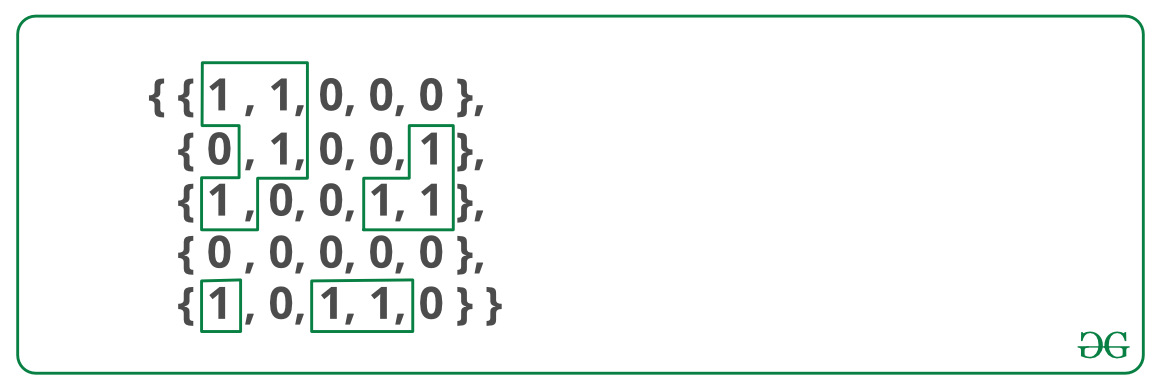
Output : 5

This is a variation of the standard problem: “Counting the number of connected components in an undirected graph”. 

Before we go to the problem, let us understand what is a connected component. A [connected component](https://www.geeksforgeeks.org/connected-components-in-an-undirected-graph/) of an undirected graph is a subgraph in which every two vertices are connected to each other by a path(s), and which is connected to no other vertices outside the subgraph.   
For example, the graph shown below has three connected components. 

Find the number of islands

A graph where all vertices are connected with each other has exactly one connected component, consisting of the whole graph. Such a graph with only one connected component is called a Strongly Connected Graph.  
The problem can be easily solved by applying DFS() on each component. In each DFS() call, a component or a sub-graph is visited. We will call DFS on the next un-visited component. The number of calls to DFS() gives the number of connected components. BFS can also be used.  
***What is an island?***   
A group of connected 1s forms an island. For example, the below matrix contains 4 islands



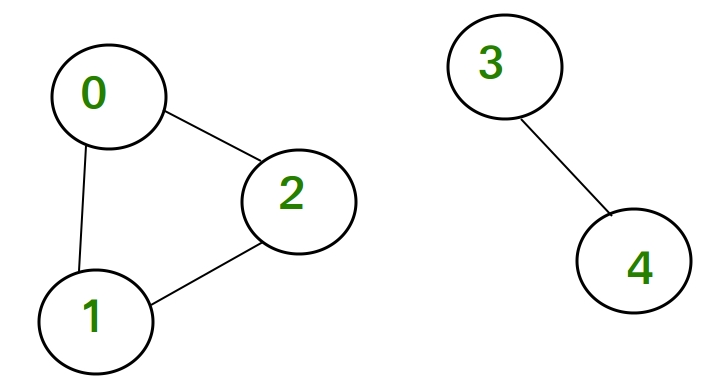
A cell in 2D matrix can be connected to 8 neighbours. So, unlike standard DFS(), where we recursively call for all adjacent vertices, here we can recursively call for 8 neighbours only. We keep track of the visited 1s so that they are not visited again.

<https://github.com/hareramcse/Datastructure/blob/master/Graph/src/com/hs/connectivity/FindNumberOfIceLand.java>

**Count the number of non-reachable nodes**

Given an undirected graph and a set of vertices, we have to count the number of non-reachable nodes from the given head node using a depth-first search.

Consider below undirected graph with two disconnected components:



In this graph, if we consider 0 as a head node, then the node 0, 1 and 2 are reachable. We mark all the reachable nodes as visited. All those nodes which are not marked as visited i.e, node 3 and 4 are non-reachable nodes. Hence their count is 2.

**Example:**

Input : 5

0 1

0 2

1 2

3 4

Output : 2

We can either use BFS or DFS for this purpose. In the below implementation, DFS is used. We do DFS from a given source. Since the given graph is undirected, all the vertices that belong to the disconnected component are non-reachable nodes. We use the visited array for this purpose, the array which is used to keep track of non-visited vertices in DFS. In DFS, if we start from the head node it will mark all the nodes connected to the head node as visited. Then after traversing the graph, we count the number of nodes that are not marked as visited from the head node.

<https://github.com/hareramcse/Datastructure/blob/master/Graph/src/com/hs/connectivity/FindNonReachableNodes.java>

**Find the Degree of a Particular vertex in a Graph**

Given a graph G(V,E) as an [adjacency matrix representation](https://www.geeksforgeeks.org/graph-and-its-representations/) and a vertex, find the degree of the vertex v in the graph.

**Examples :**

0-----1

|\ |

| \ |

| \|

2-----3

Input : ver = 0

Output : 3

Input : ver = 1

Output : 2

Algorithm:-

1. Create the graphs adjacency matrix from src to des

2. For the given vertex then check if

a path from this vertices to other exists then

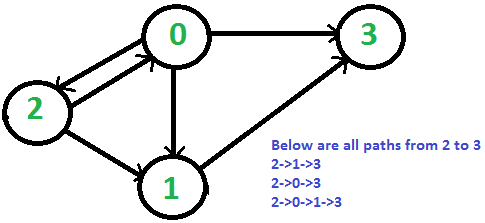
increment the degree.

3. Return degree

<https://github.com/hareramcse/Datastructure/blob/master/Graph/src/com/hs/connectivity/FindDegreeOfVertex.java>

**Print all paths from a given source to a destination**

Given a directed graph, a source vertex ‘s’ and a destination vertex ‘d’, print all paths from given ‘s’ to ‘d’.   
Consider the following directed graph. Let the s be 2 and d be 3. There are 3 different paths from 2 to 3.

[](https://media.geeksforgeeks.org/wp-content/cdn-uploads/allPaths.png)

**Approach:**

1. The idea is to do [Depth First Traversal](https://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/) of given directed graph.
2. Start the DFS traversal from source.
3. Keep storing the visited vertices in an array or HashMap say ‘path[]’.
4. If the destination vertex is reached, print contents of path[].
5. The important thing is to mark current vertices in the path[] as visited also so that the traversal doesn’t go in a cycle.

<https://github.com/hareramcse/Datastructure/blob/master/Graph/src/com/hs/connectivity/PrintAllPathFromSourceToDestination.java>

**Find the minimum cost to reach destination using a train**

There are N stations on route of a train. The train goes from station 0 to N-1. The ticket cost for all pair of stations (i, j) is given where j is greater than i. Find the minimum cost to reach the destination.  
Consider the following example: 

Input:

cost[N][N] = { {0, 15, 80, 90},

{INF, 0, 40, 50},

{INF, INF, 0, 70},

{INF, INF, INF, 0}

};

There are 4 stations and cost[i][j] indicates cost to reach j

from i. The entries where j < i are meaningless.

Output:

The minimum cost is 65

The minimum cost can be obtained by first going to station 1

from 0. Then from station 1 to station 3.

[Recommended: Please try your approach on ***{IDE}*** first, before moving on to the solution.](https://ide.geeksforgeeks.org/)

The minimum cost to reach N-1 from 0 can be recursively written as following:

minCost(0, N-1) = MIN { cost[0][n-1],

cost[0][1] + minCost(1, N-1),

minCost(0, 2) + minCost(2, N-1),

........,

minCost(0, N-2) + cost[N-2][n-1] }

<https://github.com/hareramcse/Datastructure/blob/master/Graph/src/com/hs/connectivity/MinimumCostToReachDestination.java>

**Number of groups formed in a graph of friends**

Given n friends and their friendship relations, find the total number of groups that exist. And the number of ways of new groups that can be formed consisting of people from every existing group.   
If no relation is given for any person then that person has no group and singularly forms a group. If a is a friend of b and b is a friend of c, then a b and c form a group.

**Examples:**

Input : Number of people = 6

Relations : 1 - 2, 3 - 4

and 5 - 6

Output: Number of existing Groups = 3

Number of new groups that can

be formed = 8

Explanation: The existing groups are

(1, 2), (3, 4), (5, 6). The new 8 groups

that can be formed by considering a

member of every group are (1, 3, 5),

(1, 3, 6), (1, 4, 5), (1, 4, 6), (2,

3, 5), (2, 3, 6), (2, 4, 5) and (2, 4,

6).

Input: Number of people = 4

Relations : 1 - 2 and 2 - 3

Output: Number of existing Groups = 2

Number of new groups that can

be formed = 3

Explanation: The existing groups are

(1, 2, 3) and (4). The new groups that

can be formed by considering a member

of every group are (1, 4), (2, 4), (3, 4).

To count number of groups, we need to simply count [connected components in the given undirected graph](https://www.geeksforgeeks.org/connected-components-in-an-undirected-graph/). Counting connected components can be easily done using [DFS](https://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/) or [BFS](https://www.geeksforgeeks.org/breadth-first-traversal-for-a-graph/). Since this is an undirected graph, the number of times a Depth First Search starts from an unvisited vertex for every friend is equal to the number of groups formed.

To count number of ways in which we form new groups can be done using simply formula which is (N1)\*(N2)\*….(Nn) where Ni is the no of people in i-th group.

<https://github.com/hareramcse/Datastructure/blob/master/Graph/src/com/hs/connectivity/NoOfGroupsFromFriends.java>