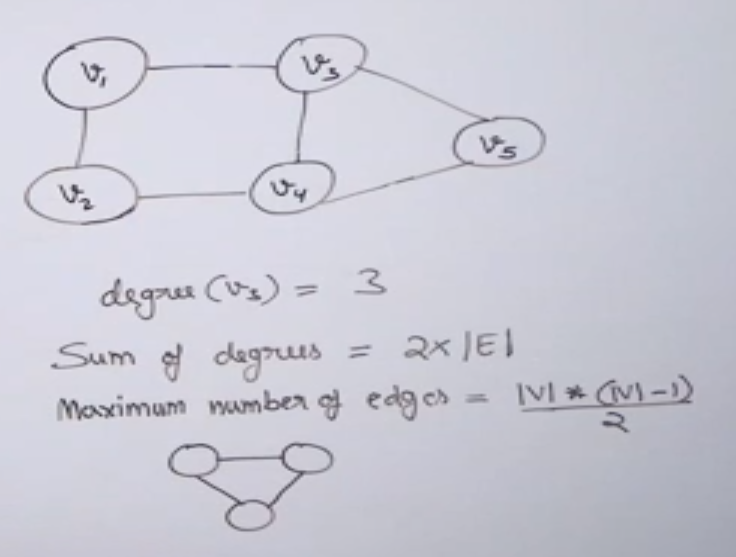
# Introduction

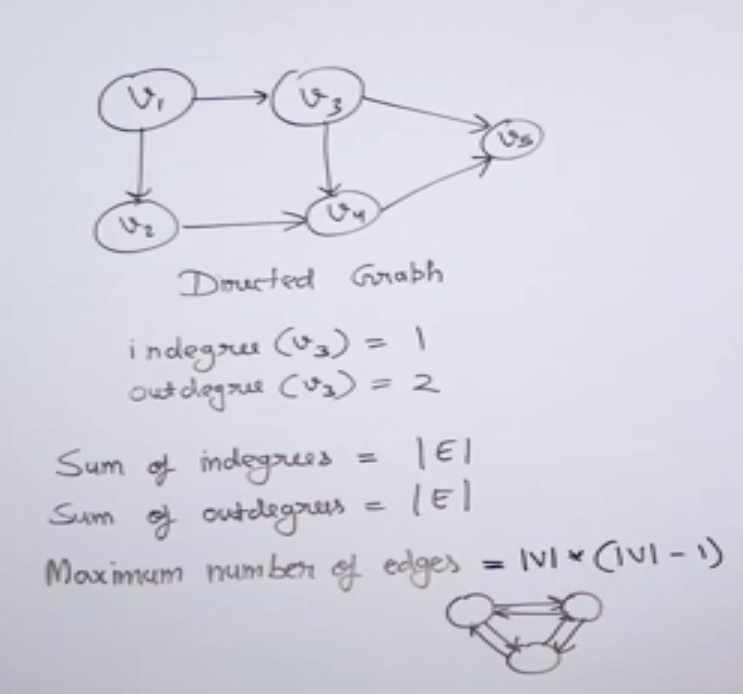
A ***Graph*** is a data structure that consists of the following two components:

1. A finite set of vertices also called nodes.
2. A finite set of ordered pair of the form (u, v) called as edge. The pair is ordered because (u, v) is not the same as (v, u) in case of a directed graph(digraph). The pair of the form (u, v) indicates that there is an edge from vertex u to vertex v. The edges may contain weight/value/cost.

## Degree of Graph:

1. Degree of undirected graph:



1. Degree of directed graph:  
   

In general, number of edges,

Undirected graph:

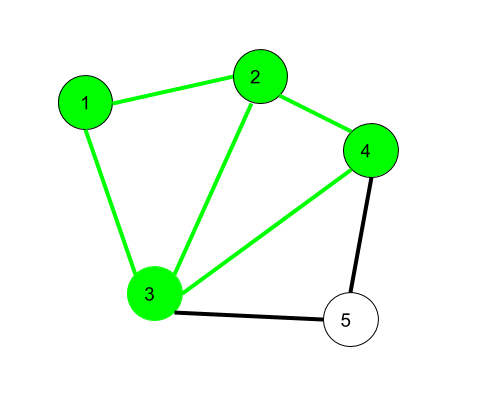
Directed graph:

# Walks, Circuits, Paths and Cycles in Graph

## Walk

A walk is a sequence of vertices and edges of a graph i.e. if we traverse a graph then we get a walk.

* Vertex can be repeated
* Edges can be repeated
* Walk can repeat anything (edges or vertices).



**Here 1->2->3->4->2->1->3 is a walk**

**Open walk-**A walk is said to be an open walk if the starting and ending vertices are different i.e. the origin vertex and terminal vertex are different.

1->2->3->4->5->3-> is an open walk.

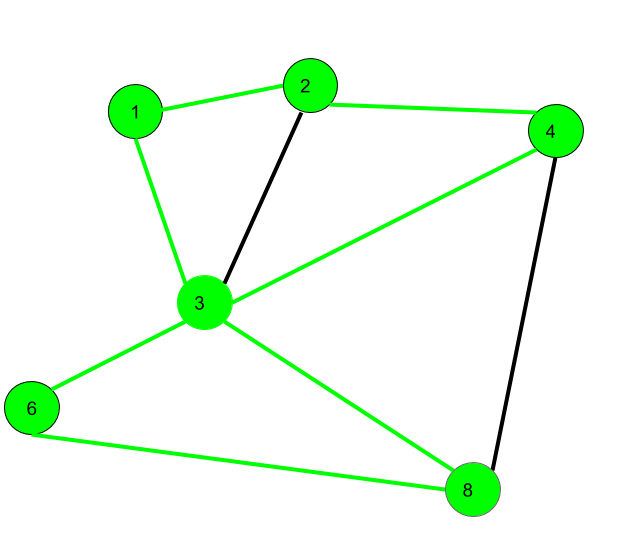
**Closed walk-**A walk is said to be a closed walk if the starting and ending vertices are identical i.e. if a walk starts and ends at the same vertex, then it is said to be a closed walk.

1->2->3->4->5->3->1-> is a closed walk.

## Circuit

Traversing a graph such that not an edge is repeated but vertex can be repeated.

* Vertex can be repeated
* Edge not repeated

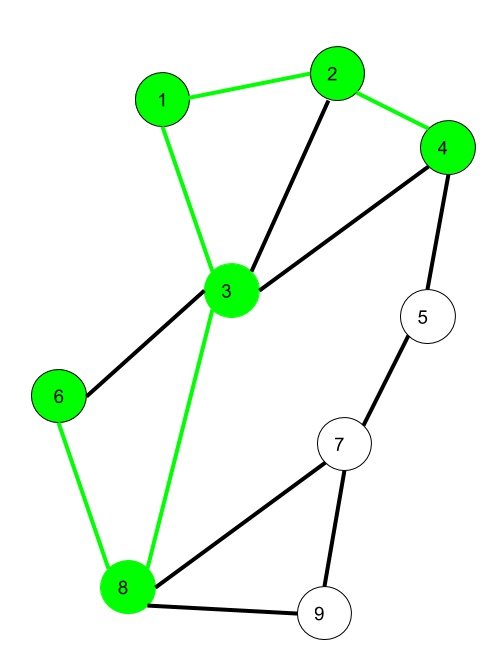


Here 1->2->4->3->6->8->3->1 is a circuit

## Path

Neither vertices nor edges are repeated i.e. if we traverse a graph such that we do not repeat a vertex and nor we repeat an edge.

* It is also an open walk
* Vertex not repeated
* Edge not repeated

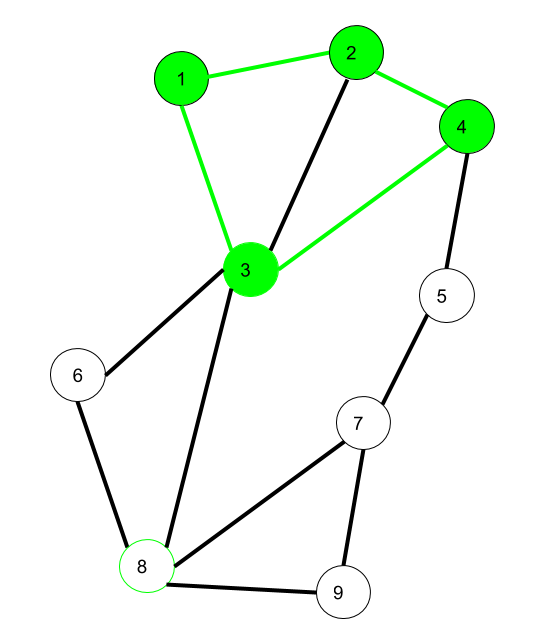


Here 6->8->3->1->2->4 is a Path

## Cycle

Traversing a graph such that we do not repeat a vertex nor we repeat a edge but the starting and ending vertex must be same i.e. we can repeat starting and ending vertex only then we get a cycle.

* Vertex not repeated
* Edge not repeated



Here 1->2->4->3->1 is a cycle.

# Graph Representation

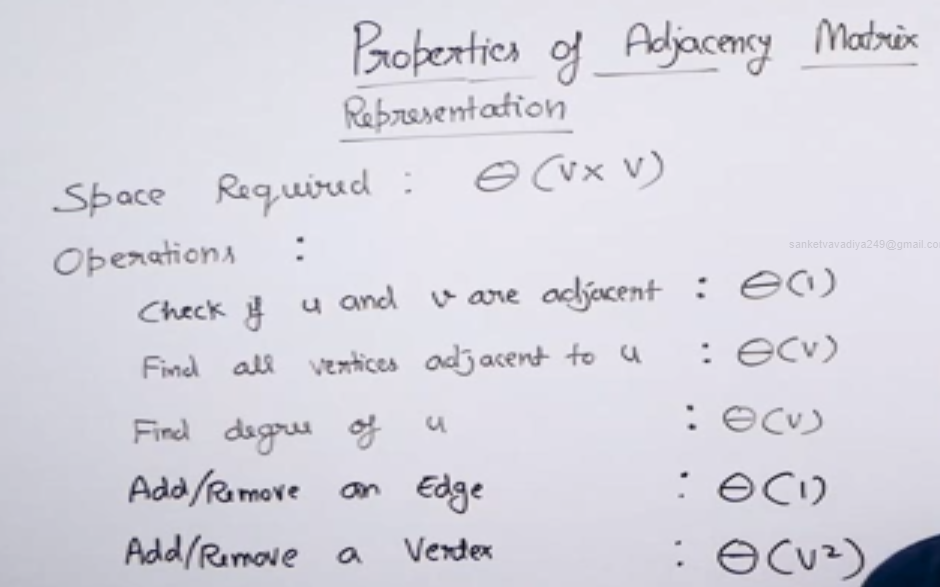
## Adjacency matrix

The Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph.

Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j.

Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs.

If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.



To add/remove vertex from graph takes time. In both the we first need to create new matrix and then copy all the element of matrix to new matrix.

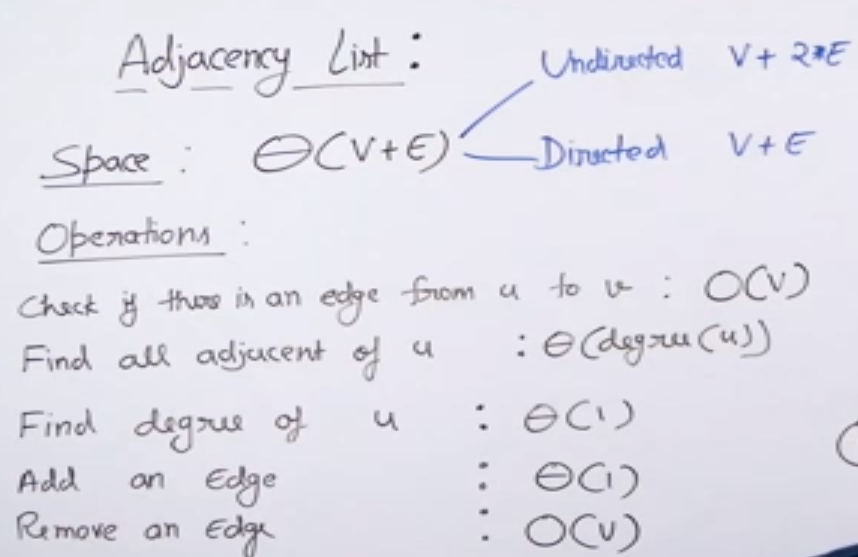
***Pros***: Representation is easier to implement and follow. Removing an edge takes O(1) time. Queries like whether there is an edge from vertex 'u' to vertex 'v' are efficient and can be done O(1).  
  
***Cons***: Consumes more space O(V^2). Even if the graph is sparse(contains less number of edges), it consumes the same space. Adding a vertex is O(V^2) time.

## Adjacency list:

Graph can also be implemented using an array of lists.

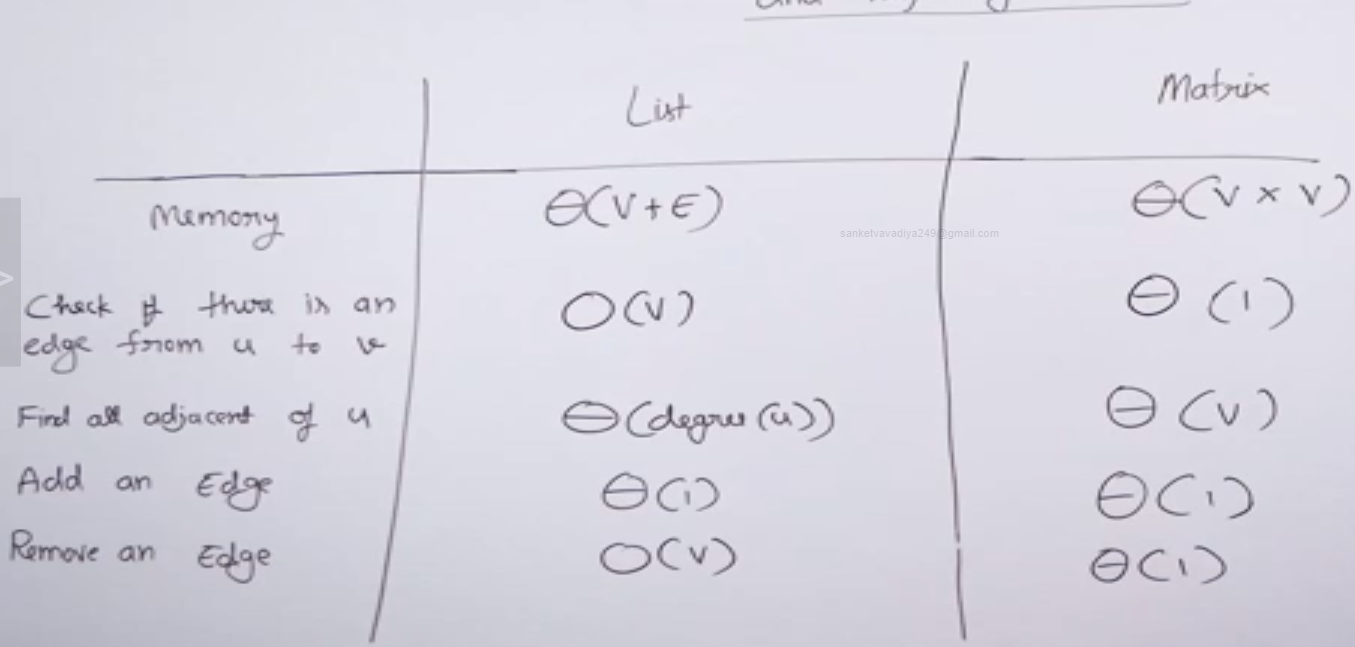
That is every index of the array will contain a complete list. Size of the array is equal to the number of vertices and every index **i** in the array will store the list of vertices connected to the vertex numbered i. Let the array be array[].

An entry array[i] represents the list of vertices adjacent to the**i**th vertex. This representation can also be used to represent a weighted graph. The weights of edges can be represented as lists of pairs. Following is the adjacency list representation of the above example undirected graph.



***Pros***: Saves space O(|V|+|E|). In the worst case, there can be C(V, 2) number of edges in a graph thus consuming O(V^2) space. Adding a vertex is easier.  
  
***Cons***: Queries like whether there is an edge from vertex u to vertex v are not efficient and can be done O(V).

## Comparison of adjacency list and matrix:



# Breadth First Search of Graph

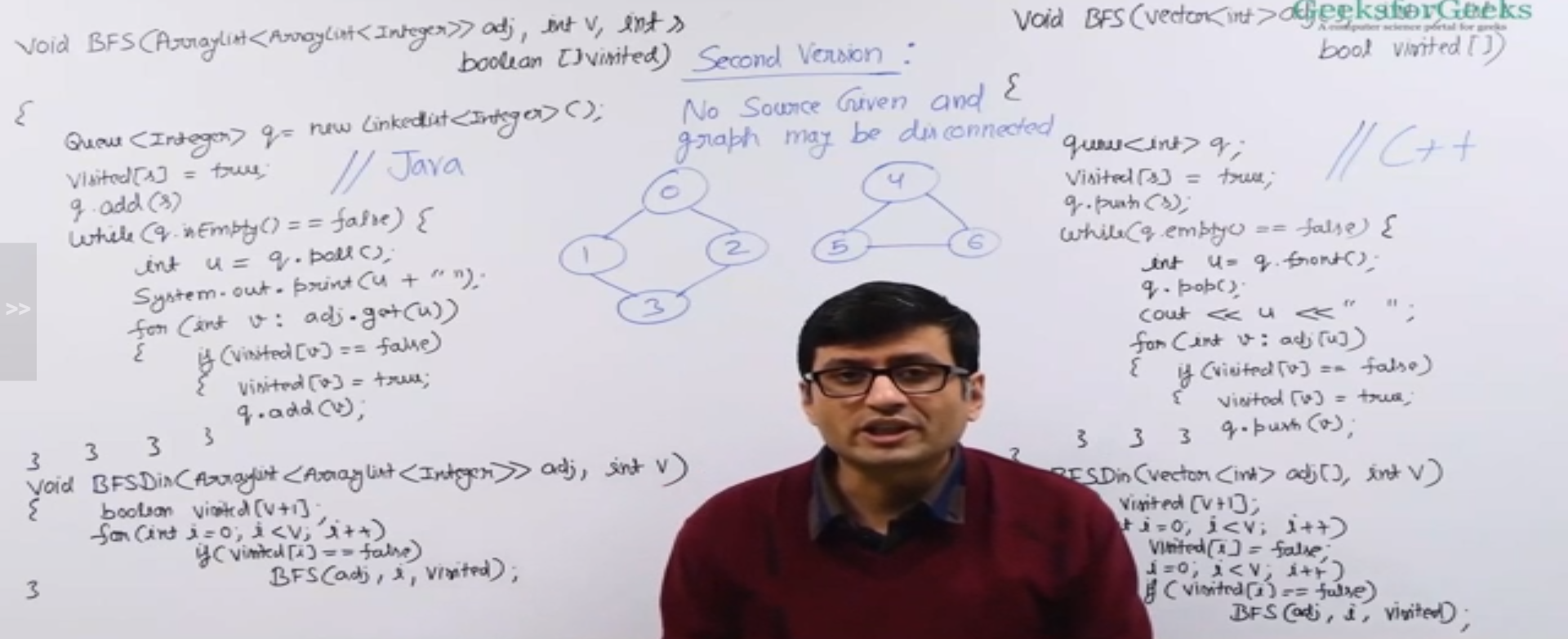
The **Breadth First Traversal** or **BFS** traversal of a graph is similar to that of the Level Order Traversal of Trees.

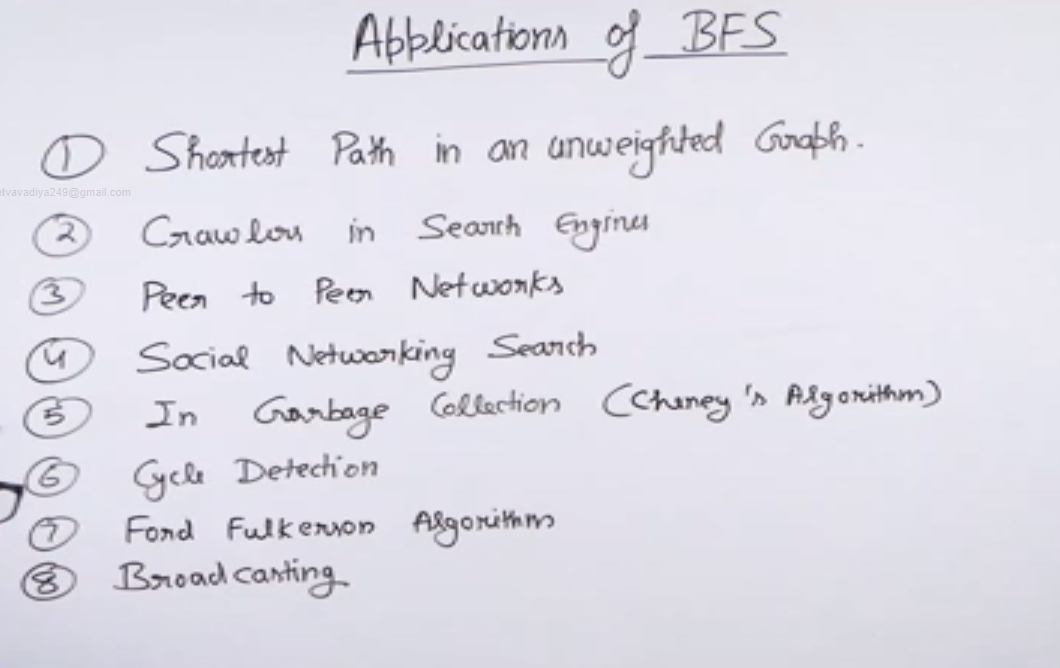
The BFS traversal uses an auxiliary boolean array say visited[] which keeps track of the visited vertices. That is if **visited[i] = true** then it means that the **i-th** vertex is already visited.

**Complete Algorithm**:

1. Create a boolean array say ***visited[]*** of size **V+1** where *V* is the number of vertices in the graph.
2. Create a Queue, mark the source vertex visited as **visited[s] = true** and push it into the queue.
3. Until the Queue is non-empty, repeat the below steps:  
   * Pop an element from the queue and print the popped element.
   * Traverse all of the vertices adjacent to the vertex poped from the queue.
   * If any of the adjacent vertex is not already visited, mark it visited and push it to the queue.

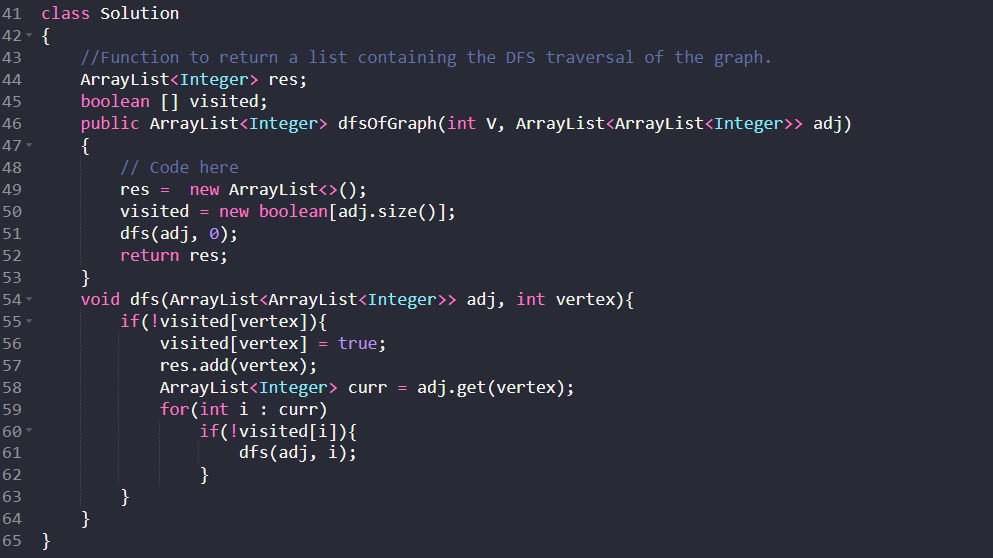
If graph has more one connected component





# Depth First Search of Graph

The Depth-First Traversal or the DFS traversal of a Graph is used to traverse a graph depth wise. That is, it in this traversal method, we start traversing the graph from a node and keep on going in the same direction as far as possible. When no nodes are left to be traversed along the current path, backtrack to find a new possible path and repeat this process until all of the nodes are visited.

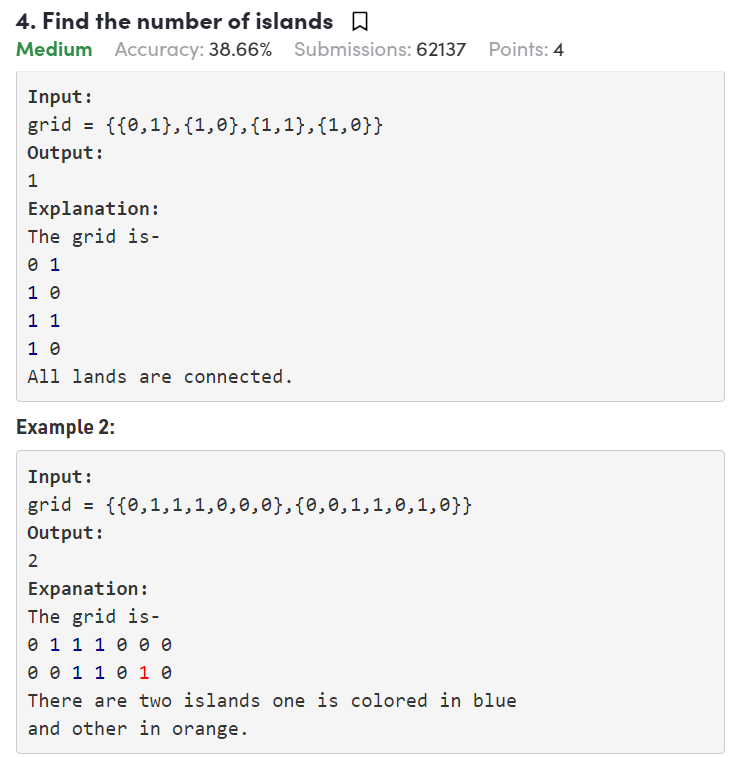


# Problems of BFS and DFS

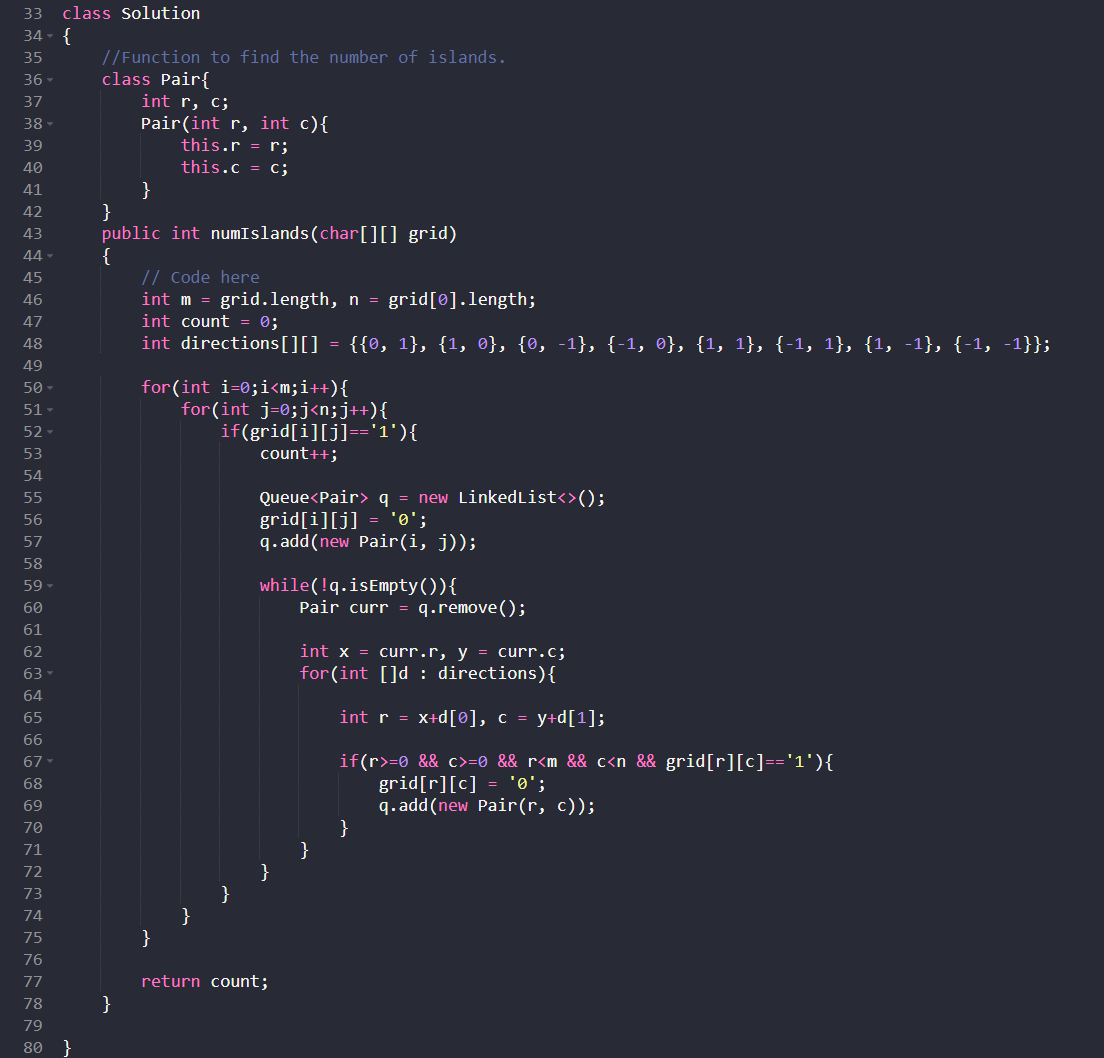
## Count connected component of graph

Also, useful to find number of islands.

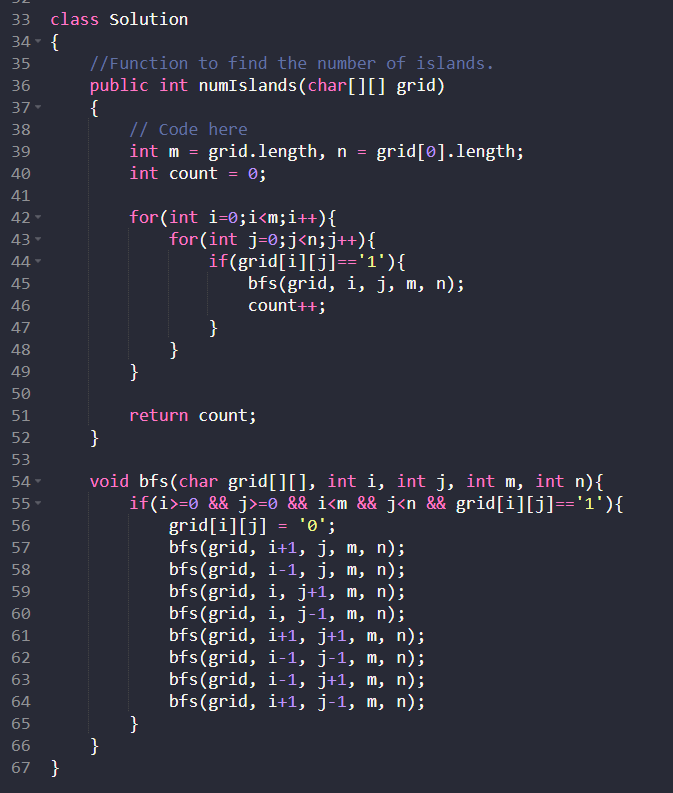
An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically or diagonally i.e., in all 8 directions.



#### Using BFS:

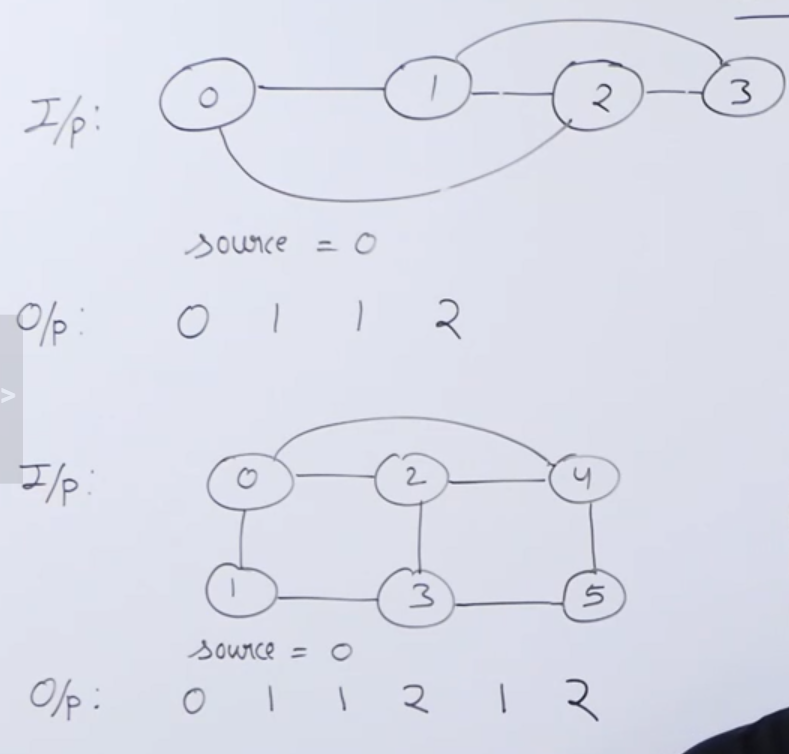


#### Using DFS:



## Shortest Path in an Unweighted Graph

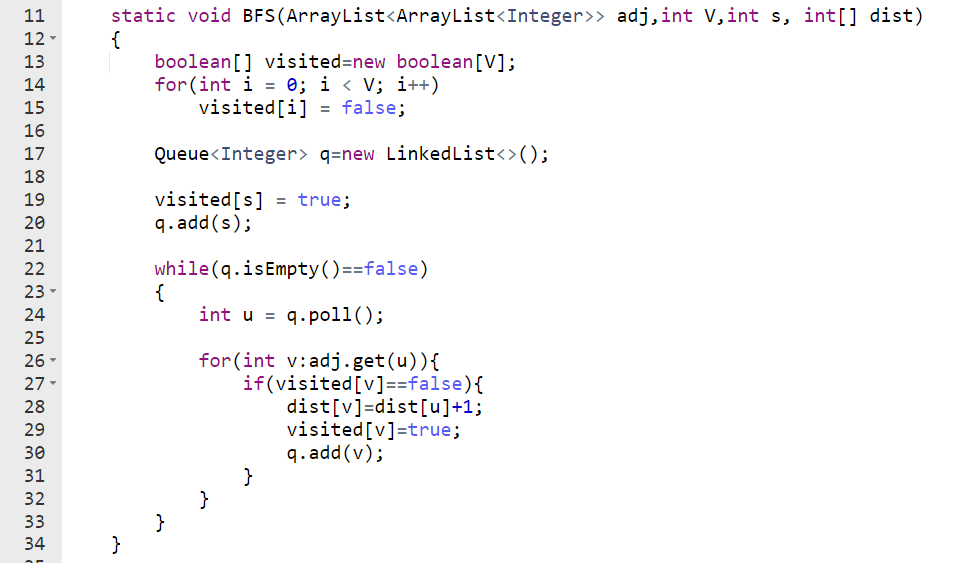
Problem: Given undirected-unweighted graph, Start from the any source vertex. Return an array that shows shortest path from source to that vertex.



**Approach:**

Initialize resultant array as distance INF for all vertex.

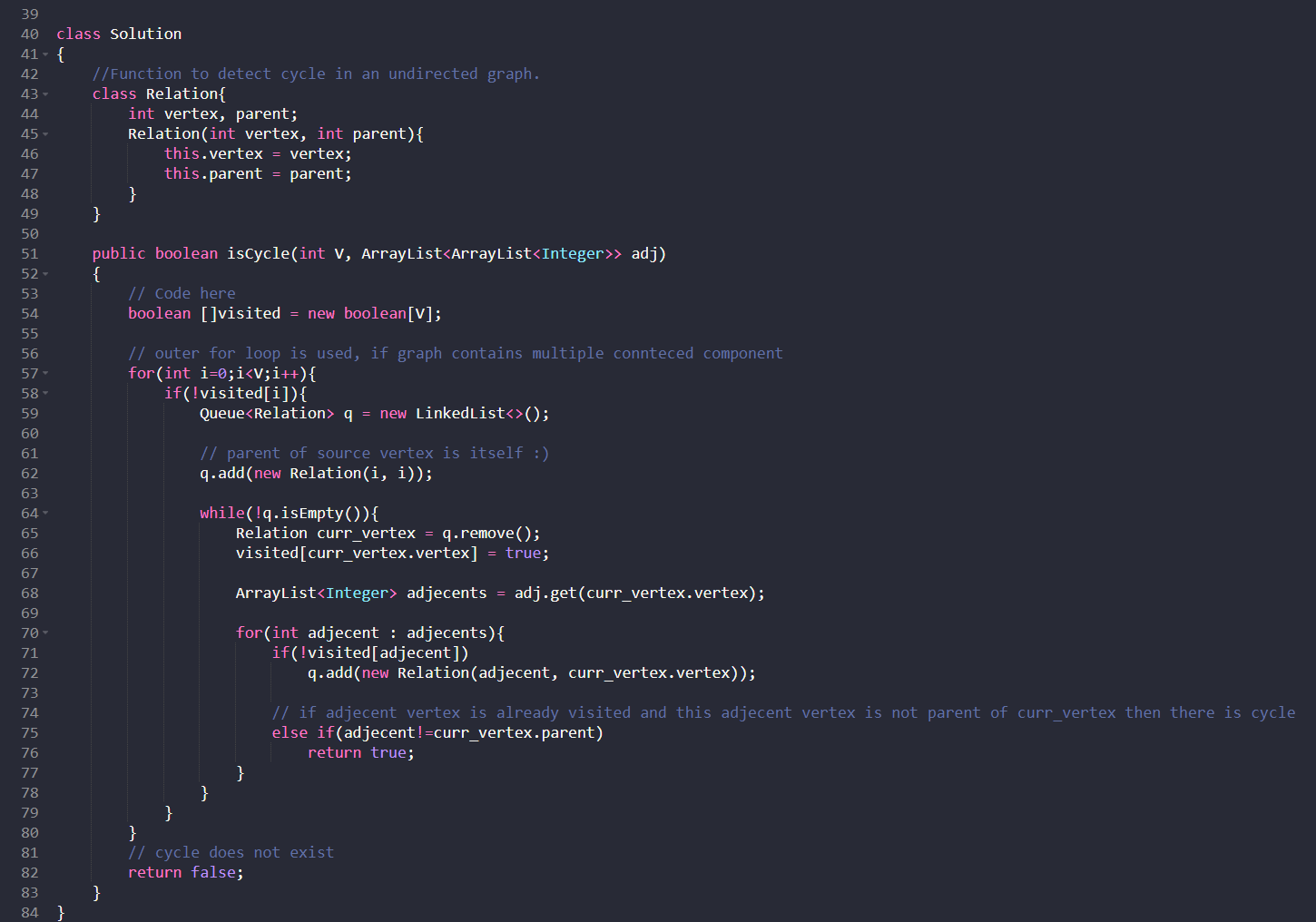
Use BFS traversal, increment distance by one during each next level of graph.



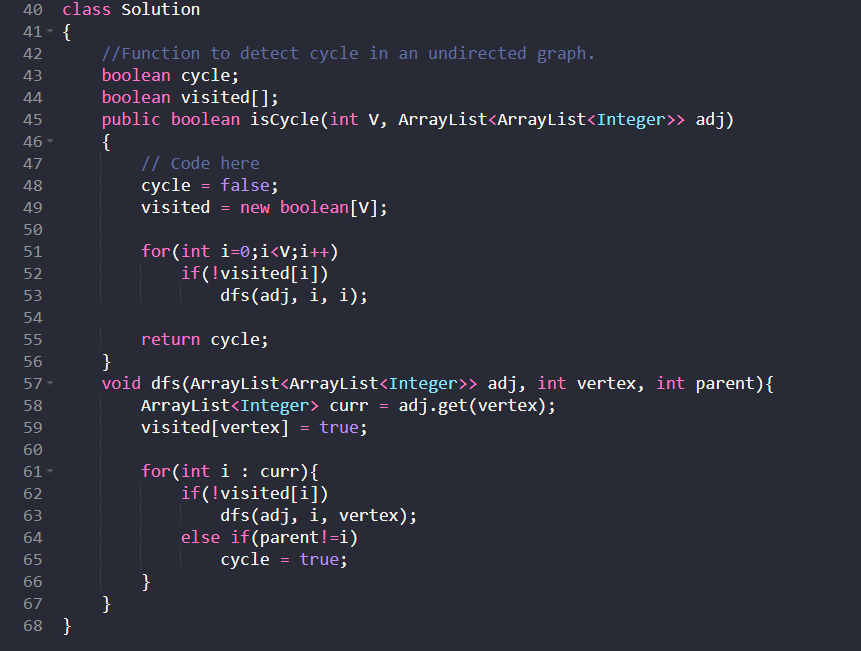
## Detect cycle in undirected graph

Given an undirected graph with V vertices and E edges, check whether it contains any cycle or not.

**BFS approach:**



**DFS approach**

****

## Detect cycle in directed graph

Approach use in undirected graph won’t work in directed graph.

We need to keep track of ancestors of current vertex. If from the current vertex there is an edge to of its ancestors (parent) then there exist cycle in graph.

**DFS approach**

Boolean array recursion\_stack used to keep track of recursion call stack. Which is used to check parents of current vertex.

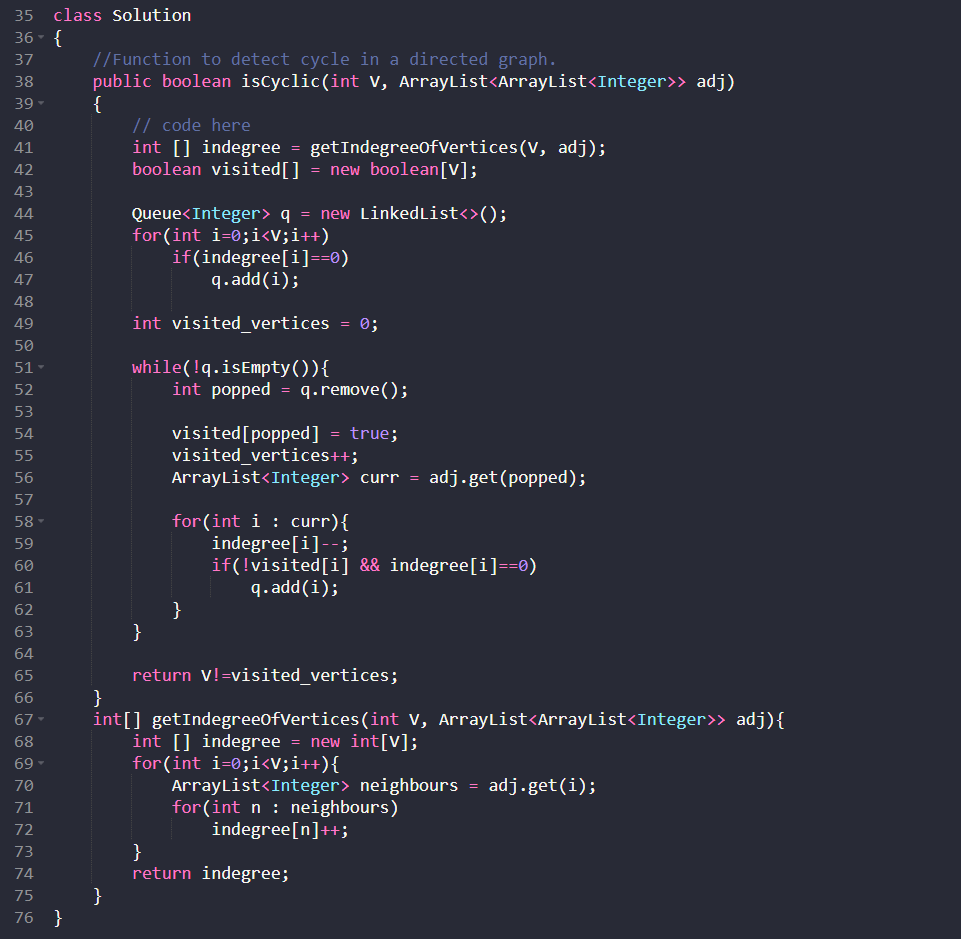
****

**BFS approach**

BFS approach is based on Kahn’s algorithm. Kahn’s algorithm is used for fining topological order of graph.

Algorithm:

1. Store indegree of each vertex in array
2. Enqueue all the vertices having indegree zero.
3. While q is not empty
   1. Remove first element from queue
   2. Mark this popped element as visited
   3. Check for all the neighbours of popped element if indegree is zeros enqueue that neighbour
4. If all vertices are visited there is no cycle in directed graph hence return false. If any of the vertices remain not visited return true as there exists cycle.



## 5. Topological sort (Kahn’s algorithm)

**BFS approach**

Kahn’s algorithm is used for fining topological order of graph.

Algorithm:

1. Store indegree of each vertex in array
2. Enqueue all the vertices having indegree zero.
3. While q is not empty
   1. Remove first element from queue
   2. Print popped element
   3. Mark this popped element as visited
   4. Traverse all the neighbours all popped element
      1. Decrement of indegree of current neighbour
      2. If indegree is zero
         1. Add element to queue



**DFS approach**

The idea is to do simple DFS. After completion of one DFS (after exiting from recursive call) we need to vertex to stack.

After visiting all the vertices pop one element from stack and print it until all the element of stack popped.

