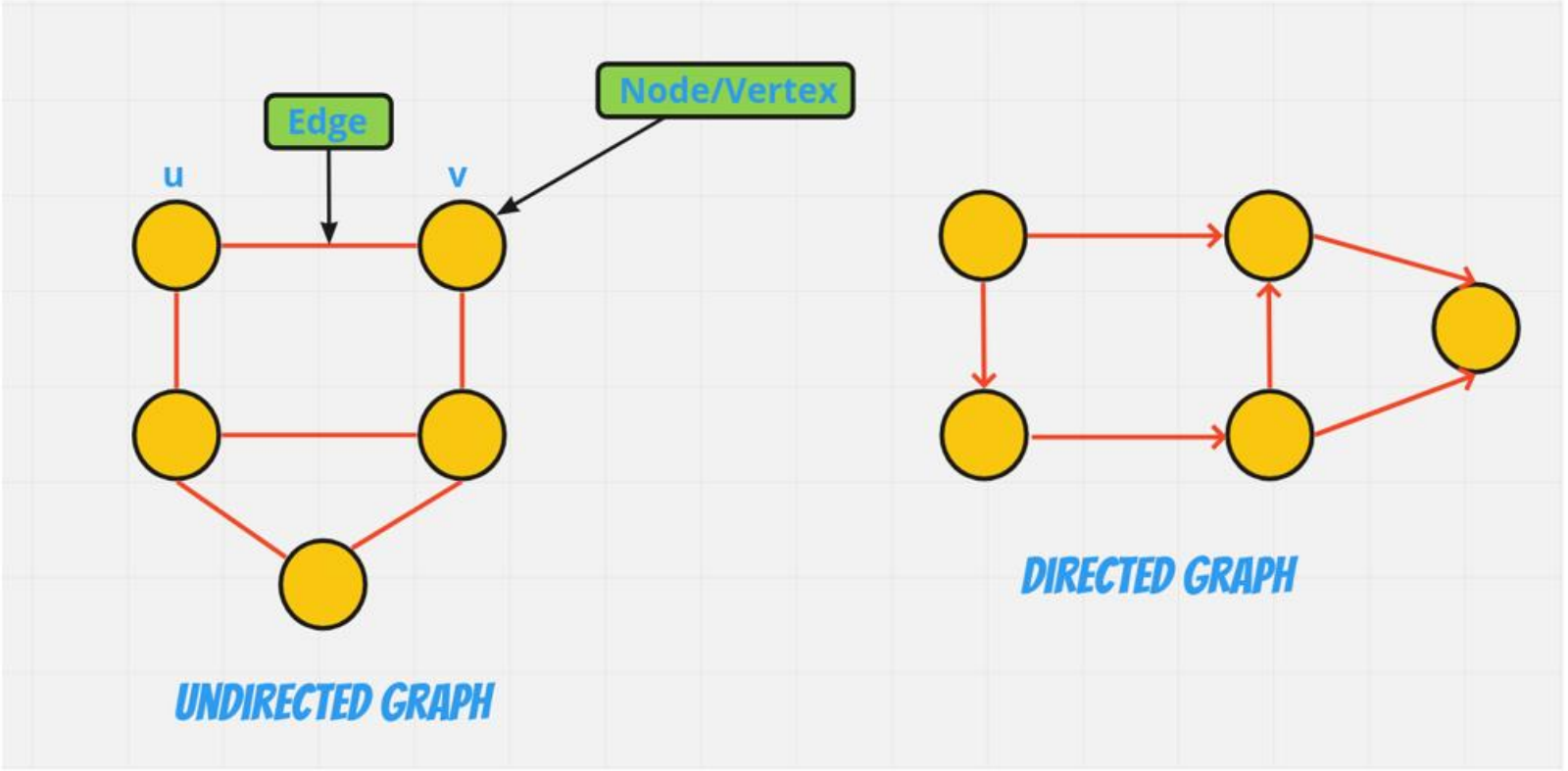
GRAPHS
A-Z

More precisely, a **graph is a data structure (V, E) that consists of**

- A collection of **vertices V**
- A collection of **edges E**, represented as ordered **pairs of vertices (u,v)**

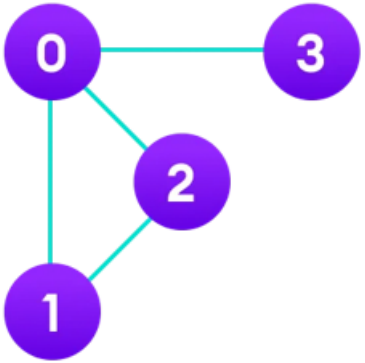
And generally there are **2 types** of Graphs :- `Undirected Graph` and `Directed Graph`



There are two ways to store Graph in a Data Structure.

1. `Adjacency Matrix` :- An adjacency matrix is a **2D array of V x V vertices**. Each row and column represent a vertex. If the value of any element `a[i][j]` is **1**, it represents that there is an edge connecting **vertex i** and **vertex j**.
   Let's understand how we store data in matrix.

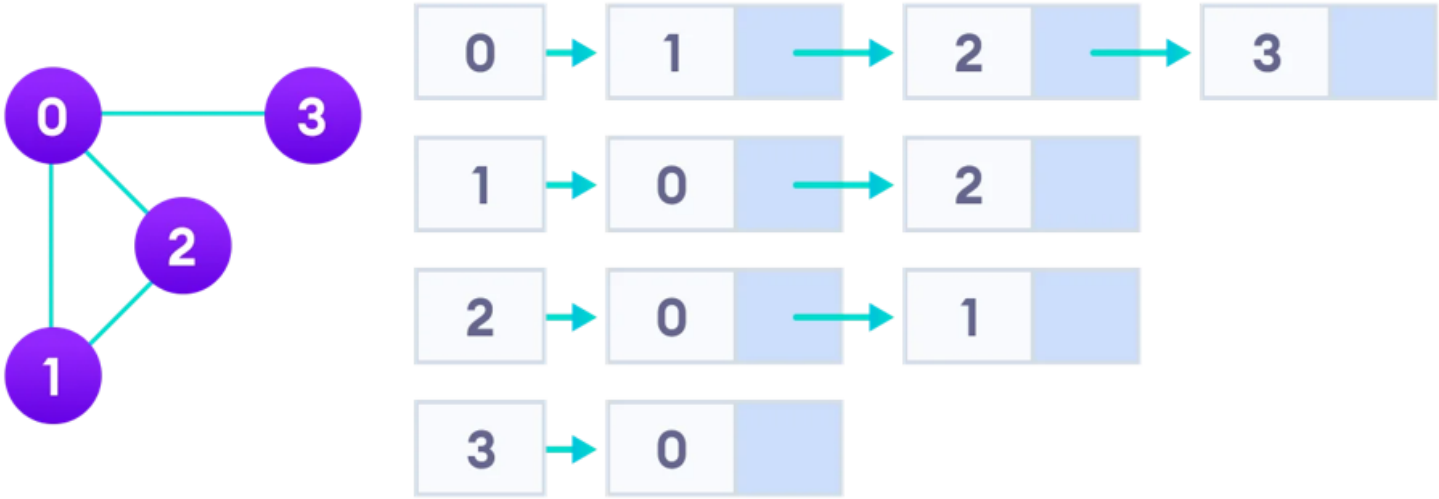
- First of all, create a matrix of `n + 1 X n + 1`
- And Intially our 2-D Array will be filled with `0`
- Since it is an **undirected graph**, for edge `(0,2)`, we also need to mark edge `(2,0)`
- And as weight of the edge is not given, we'll mark it with `1`



	0	1	2	3
0	0	1	1	1
1	1	0	1	0
2	1	1	0	0
3	1	0	0	0

Disadvantage of using Adjacency Matrix : Edge lookup(checking if an edge exists between vertex A and vertex B) is extremely fast in adjacency matrix representation but we have to **reserve space** for every possible link between all `vertices(V x V), so it requires more space.`
For example the range of matrix we have given is **10^5** then, `10^5 X 10^5` will be out of bound [M.L.E]

2. To Optimise it we use >> `Adjacency List` : An adjacency list represents a graph as an array of **linked lists**. The index of the array represents a vertex and each element in its linked list represents the other vertices that form an edge with the vertex.
   Let's understand how we store data in list, taken an example.



Java

```java
import java.io.*;
import java.util.*;

class himalik {
    public static void main (String[] args) {
        int n = 3, m = 3;
        ArrayList<ArrayList<Integer> > adj = new ArrayList<>();

        for (int i = 0; i <= n; i++)
            adj.add(new ArrayList<Integer>());

        // edge 1---2
        adj.get(1).add(2);
        adj.get(2).add(1);


        adj.get(u).add(v);
        adj.get(v).add(u);

        // edge 2---3
        adj.get(2).add(3);
        adj.get(3).add(2);


        // adge 1--3
        adj.get(1).add(3);
        adj.get(3).add(1);


        for (int i = 1; i < n; i++) {
            for (int j = 0; j < adj.get(i).size(); j++) {
                System.out.print(adj.get(i).get(j)+" ");
            }
            System.out.println();
        }
```

LEETCODE ARTICLES

https://leetcode.com/discuss/general-discussion/655708/Graph-For-Beginners-Problems-or-Pattern-or-Sample-Solutions

https://leetcode.com/discuss/study-guide/2360573/Become-Master-In-Graph

UNION FIND

Union Find:

Identify if problems talks about finding groups or components.

https://leetcode.com/problems/friend-circles/
https://leetcode.com/problems/redundant-connection/
https://leetcode.com/problems/most-stones-removed-with-same-row-or-column/
https://leetcode.com/problems/number-of-operations-to-make-network-connected/
https://leetcode.com/problems/satisfiability-of-equality-equations/
https://leetcode.com/problems/accounts-merge/

```
class Solution {
    vector<int>parent;
    int find(int x) {
        return parent[x] == x ? x : find(parent[x]);
    }
public:
    vector<int> findRedundantConnection(vector<vector<int>>& edges) {

        int n = edges.size();

        parent.resize(n+1, 0);
        for (int i = 0; i <= n; i++)
            parent[i] = i;

        vector<int>res(2, 0);
        for (int i = 0; i < n; i++) {
            int x = find(edges[i][0]);
            int y = find(edges[i][1]);
            if (x != y)
                parent[y] = x;
            else {
                res[0] = edges[i][0];
                res[1] = edges[i][1];
            }
        }

        return res;
    }
};
```

DFS

Start DFS from nodes at boundary:

https://leetcode.com/problems/surrounded-regions/

https://leetcode.com/problems/number-of-enclaves/

```
class Solution {
    int rows, cols;
    void dfs(vector<vector<int>>& A, int i, int j) {
        if (i < 0 || j < 0 || i >= rows || j >= cols)
            return;

        if (A[i][j] != 1)
            return;

        A[i][j] = -1;
        dfs(A, i+1, j);
        dfs(A, i-1, j);
        dfs(A, i, j+1);
        dfs(A, i, j-1);
    }
public:
    int numEnclaves(vector<vector<int>>& A) {

        if (A.empty()) return 0;

        rows = A.size();
        cols = A[0].size();
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                if (i == 0 || j == 0 || i == rows-1 || j == cols-1)
                    dfs(A, i, j);
            }
        }

        int ans = 0;
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                if (A[i][j] == 1)
                    ans++;
            }
        }

        return ans;
    }
};
```

DFS PART 2

DFS from each unvisited node/Island problems

https://leetcode.com/problems/number-of-closed-islands/

https://leetcode.com/problems/number-of-islands/

https://leetcode.com/problems/keys-and-rooms/

https://leetcode.com/problems/max-area-of-island/

https://leetcode.com/problems/flood-fill/

```cpp
class Solution {
    void dfs(vector<vector<char>>& grid, vector<vector<bool>>& visited, int i, int j, int m, int n) {
        if (i < 0 || i >= m || j < 0 || j >= n) return;
        if (grid[i][j] == '0' || visited[i][j]) return;
        visited[i][j] = true;
        dfs(grid, visited, i+1, j, m, n);
        dfs(grid, visited, i, j+1, m, n);
        dfs(grid, visited, i-1, j, m, n);
        dfs(grid, visited, i, j-1, m, n);
    }
    public:
    int numIslands(vector<vector<char>>& grid) {
        if (grid.empty()) return 0;

        int m = grid.size();
        int n = grid[0].size();
        vector<vector<bool>>visited(m, vector<bool>(n, false));

        int res = 0;
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (grid[i][j] == '1' && !visited[i][j]) {
                    dfs(grid, visited, i, j, m, n);
                    res++;
                }
            }
        }

        return res;
    }
    };
```

CYCLE FINDING

```cpp
class Solution {
    bool dfs(vector<vector<int>>& graph, int v, vector<int>& dp) {

        if (dp[v])
            return dp[v] == 1;

        dp[v] = -1;

        for (auto it = graph[v].begin(); it != graph[v].end(); it++)
            if (!dfs(graph, *it, dp))
                return false;

        dp[v] = 1;

        return true;
    }
 public:
    vector<int> eventualSafeNodes(vector<vector<int>>& graph) {

        int V = graph.size();

        vector<int>res;
        vector<int>dp(V, 0);

        for (int i = 0; i < V; i++) {
            if (dfs(graph, i, dp))
                res.push_back(i);
        }

        return res;
    }
```

Breadth First Search

1. Shortest Path:
   https://leetcode.com/problems/01-matrix/
   https://leetcode.com/problems/as-far-from-land-as-possible/
   https://leetcode.com/problems/rotting-oranges/
   https://leetcode.com/problems/shortest-path-in-binary-matrix/

```cpp
class Solution {
    public:
    vector<vector<int>> updateMatrix(vector<vector<int>>& matrix) {

        if (matrix.empty()) return matrix;
        int rows = matrix.size();
        int cols = matrix[0].size();
        queue<pair<int, int>>pq;
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                if (matrix[i][j] == 0) {
                    pq.push({i-1, j}), pq.push({i+1, j}), pq.push({i, j-1}), pq.push({i, j+1});
                }
            }
        }

        vector<vector<bool>>visited(rows, vector<bool>(cols, false));
        int steps = 0;
        while (!pq.empty()) {
            steps++;
            int size = pq.size();
            for (int i = 0; i < size; i++) {
                auto front = pq.front();
                int l = front.first;
                int r = front.second;
                pq.pop();
                if (l >= 0 && r >= 0 && l < rows && r < cols && !visited[l][r] && matrix[l][r] == 1) {
                    visited[l][r] = true;
                    matrix[l][r] = steps;
                    pq.push({l-1, r}), pq.push({l+1, r}), pq.push({l, r-1}), pq.push({l, r+1});
                }
            }
        }

        return matrix;
    }
};
```

GRAPH COLORING

Graph coloring/Bipartition
https://leetcode.com/problems/possible-bipartition/
https://leetcode.com/problems/is-graph-bipartite/

Problems asks to check if its possible to divide the graph nodes into 2 groups
Apply BFS for same. Below is a sample graph coloring approach.

```cpp
class Solution {
    public:
        bool isBipartite(vector<vector<int>>& graph) {
            int n = graph.size();
            vector<int>color(n, -1);

            for (int i = 0; i < n; i++) {
                if (color[i] != -1) continue;

                color[i] = 1;
                queue<int>q;
                q.push(i);

                while (!q.empty()) {
                    int t = q.front();
                    q.pop();

                    for (int j = 0; j < graph[t].size(); j++) {
                        if (color[graph[t][j]] == -1) {
                            color[graph[t][j]] = 1-color[t];
                            q.push(graph[t][j]);
                        } else if (color[graph[t][j]] == color[t]) {
                            return false;
                        }
                    }
                }
            }

            return true;
        }
    };
```

TOPO SORTING

Topological Sort:
Check if its directed acyclic graph and we have to arrange the elements in an order in which we need to select the most independent node at first.
Number of in-node 0

https://leetcode.com/problems/course-schedule/
https://leetcode.com/problems/course-schedule-ii/

```
class Solution {
    int V;
    list<int>*adj;

    bool isCyclicUtil(int v, vector<bool>&visited, vector<bool>&recStack) {

        visited[v] = true;
        recStack[v] = true;

        for (auto it = adj[v].begin(); it != adj[v].end(); it++) {
            if (!visited[*it] && isCyclicUtil(*it, visited, recStack))
                return true;
            else if (recStack[*it])
                return true;
        }

        recStack[v] = false;
        return false;
    }

    bool isCyclic() {
        vector<bool>visited(V, false);
        vector<bool>recStack(V, false);

        for (int i = 0; i < V; i++) {
            if (isCyclicUtil(i, visited, recStack))
                return true;
        }

        return false;
    }
```

TOPO SORTING PART 2

```
void topologicalSortUtil(int v, vector<bool>&visited, vector<int>& res) {
        visited[v] = true;

        for (auto it = adj[v].begin(); it != adj[v].end(); it++)
            if (!visited[*it])
                topologicalSortUtil(*it, visited, res);

        res.push_back(v);
    }

    vector<int>topologicalSort(int v) {
        vector<int>res;

        vector<bool>visited(V, false);
        topologicalSortUtil(v, visited, res);

        for (int i = 0; i < V; i++) {
            if (!visited[i])
                topologicalSortUtil(i, visited, res);
        }

        return res;
    }

    public:
    vector<int> findOrder(int numCourses, vector<vector<int>>& prerequisites) {
        V = numCourses;
        adj = new list<int>[V];

        unordered_map<int, vector<int>>hm;

        for (int i = 0; i < prerequisites.size(); i++) {
            adj[prerequisites[i][0]].push_back(prerequisites[i][1]);
            hm[prerequisites[i][1]].push_back(prerequisites[i][0]);
        }

        if (isCyclic()) return vector<int>();

        int i = 0;
        for (i = 0; i < V; i++) {
            if (hm.find(i) == hm.end())
                break;
        }

        return topologicalSort(i);
    }
```

DFS - Iterative and recursive solutions

Java Auto

```java
import java.util.Stack;

public class Solution {
    public void dfs(int[][] M, int[] visited, int start) {
        Stack<Integer> stack = new Stack<>();
        stack.push(start);

        while (!stack.isEmpty()) {
            int i = stack.pop();
            visited[i] = 1;

            for (int j = 0; j < M.length; j++) {
                if (M[i][j] == 1 && visited[j] == 0) {
                    stack.push(j);
                }
            }
        }
    }

    public int findCircleNum(int[][] M) {
        int[] visited = new int[M.length];
        int count = 0;
        for (int i = 0; i < M.length; i++) {
            if (visited[i] == 0) {
                dfs(M, visited, i);
                count++;
            }
        }
        return count;
    }
}
```

Java Auto

```java
public class Solution {
    public void dfs(int[][] M, int[] visited, int i) {
        for (int j = 0; j < M.length; j++) {
            if (M[i][j] == 1 && visited[j] == 0) {
                visited[j] = 1;
                dfs(M, visited, j);
            }
        }
    }
    public int findCircleNum(int[][] M) {
        int[] visited = new int[M.length];
        int count = 0;
        for (int i = 0; i < M.length; i++) {
            if (visited[i] == 0) {
                dfs(M, visited, i);
                count++;
            }
        }
        return count;
    }
}
```

LEETCODE ARTICLES

https://leetcode.com/discuss/general-discussion/655708/Graph-For-Beginners-Problems-or-Pattern-or-Sample-Solutions

https://leetcode.com/discuss/study-guide/2360573/Become-Master-In-Graph