Graph :

adjacency matric, list

connected components : outer most for loop

converting matrix to list : u → v and v → u

BFS

```
class Solution {
  public:
    vector<int> bfsOfGraph(int V, vector<int> adj[]) {
        int vis[V] = {0};
        vis[0] = 1;
        queue<int> q;
        // push the initial starting node
        q.push(0);
        vector<int> bfs;
        // iterate till the queue is empty
        while(!q.empty()) {
           // get the topmost element in the queue
            int node = q.front();
            q.pop();
            bfs.push_back(node);
            // traverse for all its neighbours
            for(auto it : adj[node]) {
                // if the neighbour has previously not been visited,
                // store in Q and mark as visited
                if(!vis[it]) {
                    vis[it] = 1;
                    q.push(it);
                }
            }
        }
        return bfs;
    }
};
```

DFS

```
class Solution {
    void dfs(int node, vector<int> &vis, vector<int> adj[], vector<int> &storeDfs)
{
        storeDfs.push_back(node);
        vis[node] = 1;
        for(auto it : adj[node]) {
```

```
            if(!vis[it]) {
                dfs(it, vis, adj, storeDfs);
            }
        }
    }
public:
    vector<int>dfsOfGraph(int V, vector<int> adj[]){
        vector<int> storeDfs;
        vector<int> vis(V+1, 0);
      for(int i = 1;i<=V;i++) {
        if(!vis[i]) dfs(i, vis, adj, storeDfs);
      }
        return storeDfs;
    }
};
```

- unlike in BFS we mark it visited inside dfs not before pushing

---

No.of Islands : Given a 2d grid, return the number of islands. It is given a Island is surrounded by water 0 by all 8 sides

```
class Solution {
  private:
  void bfs(int row, int col, vector<vector<int>> &vis, vector<vector<char>>&grid) {
      // mark it visited
      vis[row][col] = 1;
      queue<pair<int,int>> q;
      // push the node in queue
      q.push({row, col});
      int n = grid.size();
      int m = grid[0].size();

      // until the queue becomes empty
      while(!q.empty()) {
          int row = q.front().first;
          int col = q.front().second;
          q.pop();

          // traverse in the neighbours and mark them if its a land
          for(int delrow = -1; delrow <= 1;delrow++) {
              for(int delcol = -1; delcol <= 1; delcol++) {
                  int nrow = row + delrow;
                  int ncol = col + delcol;
                  // neighbour row and column is valid, and is an unvisited land
                  if(nrow >= 0 && nrow < n && ncol >= 0 && ncol < m
                  && grid[nrow][ncol] == '1' && !vis[nrow][ncol]) {
                      vis[nrow][ncol] = 1;
                      q.push({nrow, ncol});
```

```
                }
            }
        }
    }
  }
  public:
    // Function to find the number of islands.
    int numIslands(vector<vector<char>>& grid) {
        int n = grid.size();
        int m = grid[0].size();
        // create visited array and initialise to 0
        vector<vector<int>> vis(n, vector<int>(m, 0));
        int cnt = 0;
        for(int row = 0; row < n ; row++) {
            for(int col = 0; col < m ;col++) {
                // if not visited and is a land
                if(!vis[row][col] && grid[row][col] == '1') {
                // not visited and a land for starting point of dfs
                    cnt++;
                    bfs(row, col, vis, grid);
                }
            }
        }
        return cnt;
    }
};
```

Flood fill old with new color

```
class Solution {
private:
    void dfs(int row, int col, vector<vector<int>>&ans,
     vector<vector<int>>& image, int newColor, int delRow[], int delCol[],
     int iniColor) {
        // color with new color
        ans[row][col] = newColor;
        int n = image.size();
        int m = image[0].size();
        // there are exactly 4 neighbours
        for(int i = 0;i<4;i++) {
            int nrow = row + delRow[i];
            int ncol = col + delCol[i];
            // check for valid coordinate
            // then check for same initial color and unvisited pixel
            if(nrow>=0 && nrow<n && ncol>=0 && ncol < m &&
            image[nrow][ncol] == iniColor && ans[nrow][ncol] != newColor) {
                dfs(nrow, ncol, ans, image, newColor, delRow, delCol, iniColor);
```

```
            }
        }
    }
public:
    vector<vector<int>> floodFill(vector<vector<int>>& image,
    int sr, int sc, int newColor) {
        // get initial color
        int iniColor = image[sr][sc];
        vector<vector<int>> ans = image;
        // delta row and delta column for neighbours
        int delRow[] = {-1, 0, +1, 0};
        int delCol[] = {0, +1, 0, -1};
        dfs(sr, sc, ans, image, newColor, delRow, delCol, iniColor);
        return ans;
    }
};
```

Rotten Oranges

```
class Solution {
public:
    int orangesRotting(vector<vector<int>>& grid) {

        int n = grid.size();
        int m = grid[0].size();

        queue<pair<int, int>> q;
        int fresh_oranges = 0;

        for(int i = 0;i < n;i++) {
            for(int j = 0;j < m;j++) {
                if(grid[i][j] == 2) q.push({i, j});
                else if(grid[i][j] == 1) fresh_oranges++;
            }
        }

        int del_row[] = {-1, 0, 1, 0};
        int del_col[] = {0, 1, 0, -1};

        int time = 0;
        while(!q.empty() && fresh_oranges > 0) {
        // There could be some starting points but not fresh oranges
            int no_sources = q.size();
            for(int s = 0;s < no_sources;s++) {
                int x = q.front().first;
                int y = q.front().second;
                q.pop();
```

```
                for(int i = 0;i < 4;i++) {
                    int new_x = x + del_row[i];
                    int new_y = y + del_col[i];
                    if(new_x >= 0 && new_x < n && new_y >= 0 && new_y < m &&
                        grid[new_x][new_y] == 1) {
                            grid[new_x][new_y] = 2;
                            q.push({new_x, new_y});
                            fresh_oranges--;
                        }
                }
            }
            time++;
        }
        if(fresh_oranges > 0) return -1;
        return time;
    }
};
```

- Try using a copy of grid as visited
- We can also solve this by using pair<pair<int, int>, int> where 1st pair denotes co-ordinates and second time.

Distance of nearest cell having 1 | 0/1 Matrix |

- Similar to prev question, do simultaneous BFS with 1 and add time in new grid till all zeroes are ones!

Replace O with X, Surrounded Region gets flooded

```
class Solution{
private:
    void dfs(int row, int col, vector<vector<int>> &vis,
    vector<vector<char>> &mat, int delrow[], int delcol[]) {
        vis[row][col] = 1;
        int n = mat.size();
        int m = mat[0].size();

        // check for top, right, bottom, left
        for(int i = 0;i<4;i++) {
            int nrow = row + delrow[i];
            int ncol = col + delcol[i];
            // check for valid coordinates and unvisited Os
            if(nrow >=0 && nrow <n && ncol >= 0 && ncol < m
            && !vis[nrow][ncol] && mat[nrow][ncol] == 'O') {
                dfs(nrow, ncol, vis, mat, delrow, delcol);
            }
        }
    }
```

```
public:
    vector<vector<char>> fill(int n, int m,
    vector<vector<char>> mat)
    {
        int delrow[] = {-1, 0, +1, 0};
        int delcol[] = {0, 1, 0, -1};
        vector<vector<int>> vis(n, vector<int>(m,0));
        // traverse first row and last row
        for(int j = 0 ; j<m;j++) {
            // check for unvisited Os in the boundary rows
            // first row
            if(!vis[0][j] && mat[0][j] == 'O') {
                dfs(0, j, vis, mat, delrow, delcol);
            }

            // last row
            if(!vis[n-1][j] && mat[n-1][j] == 'O') {
                dfs(n-1,j,vis,mat, delrow, delcol);
            }
        }

        for(int i = 0;i<n;i++) {
            // check for unvisited Os in the boundary columns
            // first column
            if(!vis[i][0] && mat[i][0] == 'O') {
                dfs(i, 0, vis, mat, delrow, delcol);
            }

            // last column
            if(!vis[i][m-1] && mat[i][m-1] == 'O') {
                dfs(i, m-1, vis, mat, delrow, delcol);
            }
        }

        // if unvisited O then convert to X
        for(int i = 0;i<n;i++) {
            for(int j= 0 ;j<m;j++) {
                if(!vis[i][j] && mat[i][j] == 'O')
                    mat[i][j] = 'X';
            }
        }

        return mat;
    }
};
```

No.of Enclaves : return number of grids which are part of a Island.

- I am for sure all those lands which are connected to the boundary would not be a part of my ans. SAME PREV QUE

No.of distinct Islands. Similar to no.of Islands but the structure of each island is different.

```
class Solution {
  private:
    void dfs(int row, int col, vector < vector < int >> & vis,
      vector < vector < int >> & grid, vector < pair < int, int >> & vec, int row0,
      int col0) {
      // mark the cell as visited
      vis[row][col] = 1;

      // coordinates - base coordinates //**
      vec.push_back({
        row - row0,
        col - col0
      });
      int n = grid.size();
      int m = grid[0].size();

      // delta row and delta column
      int delrow[] = {-1, 0, +1, 0};
      int delcol[] = {0, -1, 0, +1};

      // traverse all 4 neighbours
      for (int i = 0; i < 4; i++) {
        int nrow = row + delrow[i];
        int ncol = col + delcol[i];
        // check for valid unvisited land neighbour coordinates
        if (nrow >= 0 && nrow < n && ncol >= 0 && ncol < m &&
          !vis[nrow][ncol] && grid[nrow][ncol] == 1) {
          dfs(nrow, ncol, vis, grid, vec, row0, col0);
        }
      }
    }
  public:
    int countDistinctIslands(vector < vector < int >> & grid) {
      int n = grid.size();
      int m = grid[0].size();
      vector < vector < int >> vis(n, vector < int > (m, 0));
      set < vector < pair < int, int >>> st;
      //** vector < pair < int, int >> structure of each island in base subtract form

      // traverse the grid
      for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
```

```
          // if not visited and is a land cell
          if (!vis[i][j] && grid[i][j] == 1) {
            vector < pair < int, int >> vec;
            dfs(i, j, vis, grid, vec, i, j);
            // store in set
            st.insert(vec);
          }
        }
      }
      return st.size();
    }
};
```

Bipartite Graphs : In simpler terms, it's a graph that can be colored using two colors such that no two adjacent vertices share the same color.

- It should not have any cycles
- If it has a cycle then it should be of even length
- Keep a visited array of -1. Start coloring with starting node as 0, then keep altering it. At any of the state you encounter an already colored node check if it is different.

```
// BFS
for(auto it : adj[node]) {
    // if the adjacent node is yet not colored
    // you will give the opposite color of the node
    if(color[it] == -1) {
        color[it] = !color[node];
        q.push(it);
    }
    // is the adjacent guy having the same color
    // someone did color it on some other path
    else if(color[it] == color[node]) {
        return false;
    }
}
```

```
// DFS
for(auto it : adj[node]) {
    // if uncoloured
    if(color[it] == -1) {
        if(dfs(it, !col, color, adj) == false) return false;
    }
    // if previously coloured and have the same colour
    else if(color[it] == col) {
        return false;
```

```
    }
}
```

- Make sure to take care of muplite connected comp, then do same with source always with 0

TOPOLOGICAL SORTING

- Topological sorting is possible only for directed acyclic graphs.
- ordering of vertices such that for every directed edge U → V ⇒ U comes before V.

```
// DFS
void topologicalSortUtil(int v, vector<bool>& visited, stack<int>& Stack,
const vector<vector<int>>& adj) {
    visited[v] = true;

    for (int neighbor : adj[v]) {
        if (!visited[neighbor]) {
            topologicalSortUtil(neighbor, visited, Stack, adj);
        }
    }

    Stack.push(v);
    // Once you are done with the dfs you can push it in the
    //confidently knowing that the forward node are already processed
}

vector<int> topologicalSort(int n, const vector<vector<int>>& adj) {
    stack<int> Stack;
    vector<bool> visited(n, false);

    // Call the recursive helper function for all vertices
    for (int i = 0; i < n; i++) {
        if (!visited[i]) {
            topologicalSortUtil(i, visited, Stack, adj);
        }
    }

    vector<int> ans;
        while (!Stack.empty()) {
            ans.push_back(st.top());
        Stack.pop();
    }

    return ans;
}
```

Khans Algo :

- It is based on the idea of processing nodes with zero in-degree and removing their outgoing edges while updating the in-degrees of the neighboring nodes.

```
vector<int> topoSort(int V, vector<int> adj[])
    {
        int indegree[V] = {0};
        for (int i = 0; i < V; i++) {
            for (auto it : adj[i]) {
                indegree[it]++;
            }
        }

        queue<int> q;
        for (int i = 0; i < V; i++) {
            if (indegree[i] == 0) {
                q.push(i);
            }
        }
        vector<int> topo;
        while (!q.empty()) {
            int node = q.front();
            q.pop();
            topo.push_back(node);
            // node is in your topo sort
            // so please remove it from the indegree

            for (auto it : adj[node]) {
                indegree[it]--;
                if (indegree[it] == 0) q.push(it);
            }
        }

        return topo;
    }
```

- Detection of cycles in directed Graph using topo sort
- Directed Graphs topo sort would be of size equal to no.of nodes
- else ⇒ there is a cycle

- Course Scheduling / Pre-requisites
- Build a DAG using the given pairs as edges
- Check if topo sort exists ⇒ is it a DAG ⇒ Above condition

Find Eventual Safe States

- A node is a safe node if every possible path starting from that node leads to a terminal node.
- A terminal node is a node without any outgoing edges
- If we closely observe, all possible paths starting from a node are going to end at some terminal node unless there exists a cycle and the paths return back to themselves.
- the intuition is to figure out the nodes which are neither a part of a cycle nor connected to the cycle.
- The node with outdegree 0 is considered to be a terminal node but the topological sort algorithm deals with the indegrees of the nodes. So, to use the topological sort algorithm, we will reverse every edge of the graph. Now, the nodes with indegree 0 become the terminal nodes. After this step, we will just follow the topological sort algorithm as it is.
- Store all nodes which eventually had 0 indegree (Eventual Safe States)

Alien Dictionary

- Has N words, k characters : Find its lexicographical order

```
string findOrder(string dict[], int N, int K) {
        vector<int> adj[K];
        for (int i = 0; i < N - 1; i++) {
            string s1 = dict[i];
            string s2 = dict[i + 1];
            int len = min(s1.size(), s2.size());
            for (int ptr = 0; ptr < len; ptr++) {
                if (s1[ptr] != s2[ptr]) {
                    adj[s1[ptr] - 'a'].push_back(s2[ptr] - 'a');
                    break;
                }
            }
        }

        vector<int> topo = topoSort(K, adj);
        string ans = "";
        for (auto it : topo) {
            ans = ans + char(it + 'a');
        }

        return ans;
}
```

- when would be there no possible order here ?
- When there is a cyclic dependency {topo size < n}

- Or the case where the small string comes later than big where small entirely matches with starting big string

---

Shortest Path :

```
// when you need to handle cases where the graph might have weighted edges or
// when you need to potentially revisit nodes to find a shorter path.
if (dist[node] + 1 < dist[it]) {
    dist[it] = 1 + dist[node];
    q.push(it);
}
```

```
 // for standard BFS in unweighted graphs where all edges have the same weight,
 // and you are only concerned with visiting nodes once.
 // Re-visiting is not needed because the first time
 // we visit it it will be the shortest
 if (distances[neighbor] == -1) { // Not visited
    distances[neighbor] = distances[node] + 1;
    q.push(neighbor);
}
```

Dijkstra’s Algo - Priority Queue

```
class Solution
{
public:
    vector<int> dijkstra(int V, vector<vector<int>> adj[], int S)
    {

        // Create a priority queue for storing the nodes as a pair {dist,node}
        // where dist is the distance from source to the node.
        priority_queue<pair<int, int>, vector<pair<int, int>>,
        greater<pair<int, int>>> pq;

        // Initialising distTo list with a large number to
        // indicate the nodes are unvisited initially.
        // This list contains distance from source to the nodes.
        vector<int> distTo(V, INT_MAX);

        // Source initialised with dist=0.
        distTo[S] = 0;
        pq.push({0, S});

        // Now, pop the minimum distance node first from the min-heap
        // and traverse for all its adjacent nodes.
        while (!pq.empty())
        {
```

```
            int node = pq.top().second;
            int dis = pq.top().first;
            pq.pop();

            // Check for all adjacent nodes of the popped out
            // element whether the prev dist is larger than current or not.
            for (auto it : adj[node])
            {
                int v = it[0];
                int w = it[1];
                if (dis + w < distTo[v])
                //** It is a Greedy-Based Algorithm where a node with higher dis,
                // waits for a shorter distance to come around
                // (BFS karre na baki node with shorter dis)
                // But if no nood reaches after specified then it assumes
                // itself to be shortest and continues the algo
                {
                    distTo[v] = dis + w;

                    // If current distance is smaller,
                    // push it into the queue.
                    // ** If there was already a pair with same node with more
                    // dis that will be given less priority so it will
                    // do nothing in the end
                    pq.push({dis + w, v});
                }
            }
        }

        return distTo;
    }
};
```

- What if I use set instead of PQ ?

```cpp
vector <int> dijkstra(int V, vector<vector<int>> adj[], int S)
{
    set<pair<int,int>> st;
    vector<int> dist(V, 1e9);

    st.insert({0, S});
    dist[S] = 0;

    while(!st.empty()) {
        auto it = *(st.begin());
        int node = it.second;
        int dis = it.first;
        st.erase(it);

        for(auto it : adj[node]) {
            int adjNode = it[0];
            int edgW = it[1];

            if(dis + edgW < dist[adjNode]) {
                // erase if it existed
                if(dist[adjNode] != 1e9)
                    st.erase({dist[adjNode], adjNode});

                dist[adjNode] = dis + edgW;
                st.insert({dist[adjNode], adjNode});
            }
        }
    }
    return dist;
}
```

- We are saving some iterations, by removing {higher_dis, same_node};
- but that to takes logn time so there are pros and cons

- Now I know, the shortest distance but what about the path ?
- We will the ALIAS method from arrays to do this.
- keep a parent array and backtrack from the destination

```cpp
parent(n + 1); // 1 based indexing

// inside while
if (dis + edW < dist[adjNode])
{
    dist[adjNode] = dis + edW;
    pq.push({dis + edW, adjNode});

    // Update the parent of the adjNode to the recent
    // node where it came from.
    parent[adjNode] = node;
}

vector<int> path;
int node = n; // destination

// Iterate backwards from destination to source through the parent array.
while (parent[node] != node)
{
    path.push_back(node);
    node = parent[node];
```

```
}
path.push_back(1); // source

// Since the path stored is in a reverse order, we reverse the array
// to get the final answer and then return the array.
reverse(path.begin(), path.end());
return path;
```

Given a Grid of 0 and 1 return the min dis req to reach destination given source.

```
class Solution {
  public:
    int shortestPath(vector<vector<int>> &grid, pair<int, int> source,
                     pair<int, int> destination) {
        int n = grid.size();
        int m = grid[0].size();

        vector<vector<int>> dis(n, vector<int>(m, INT_MAX));
        queue<pair<int, pair<int, int>>> q;
        // Why not priority queue ?
        // As the weights are all of same weight all nodes in queue
        // at a moment will have same curr_dis
        q.push({0, source});
        dis[source.first][source.second] = 0;

        int del_row[] = {-1, 0, 1, 0};
        int del_col[] = {0, 1, 0, -1};

        while(!q.empty()) {
            int curr_dis = q.front().first;
            int row = q.front().second.first;
            int col = q.front().second.second;
            q.pop();

            for(int i = 0;i < 4;i++) {
                int new_row = row + del_row[i];
                int new_col = col + del_col[i];

                if(new_row >= 0 && new_row < n && new_col >= 0 && new_col < m &&
                    grid[new_row][new_col] == 1 &&
                    dis[new_row][new_col] > curr_dis + 1)
                    // As the weights are all same we can just do
                    // dis[new_row][new_col] == INT_MAX
                {
                    dis[new_row][new_col] = curr_dis + 1;
                    q.push({curr_dis + 1, {new_row, new_col}});
                }
```

```
            }
        }

        return dis[destination.first][destination.second]
        == INT_MAX ? -1 : dis[destination.first][destination.second];
    }
};
```

- Say now, we are given a grid with some values in it. Then the weight is given by absolute difference between them.
- Use priority queue here instead of a normal queue

Given a standard directed weighted graph, return me the cheapest flight within k stops. If there exists a much cheaper flight at more number of stops I don’t care, TIME is constrained here

▼ Few changes

```
class Solution
{
public:
    int CheapestFLight(int n, vector<vector<int>> &flights,
                       int src, int dst, int K)
    {
        vector<pair<int, int>> adj[n];
        for (auto it : flights)
        {
            adj[it[0]].push_back({it[1], it[2]});
        }

        // Create a queue which stores the node and their distances from the
        // source in the form of {stops, {node, dist}} with ‘stops’ indicating
        // the no. of nodes between src and current node.
        queue<pair<int, pair<int, int>>> q; // by queue ? here we are giving prior

        q.push({0, {src, 0}});

        vector<int> dist(n, 1e9);
        dist[src] = 0;

        while (!q.empty())
        {
            auto it = q.front();
            q.pop();
            int stops = it.first;
            int node = it.second.first;
            int cost = it.second.second;
```

```
            // We stop the process as soon as the limit for the stops reaches.
            if (stops > K)
                continue;
            for (auto iter : adj[node])
            {
                int adjNode = iter.first;
                int edW = iter.second;

                if (cost + edW < dist[adjNode] && stops <= K) //** =K is also allo
                {
                    dist[adjNode] = cost + edW;
                    q.push({stops + 1, {adjNode, cost + edW}});
                }
            }
        }

        if (dist[dst] == 1e9) return -1;
        return dist[dst];
    }
};
```

Return me the number of ways i can reach a destination in the shortest path possible

```
class Solution
{
public:
    int countPaths(int n, vector<vector<int>> &roads)
    {
        vector<pair<int, int>> adj[n];
        for (auto it : roads)
        {
            adj[it[0]].push_back({it[1], it[2]});
            adj[it[1]].push_back({it[0], it[2]});
        }

        priority_queue<pair<int, int>,
                       vector<pair<int, int>>, greater<pair<int, int>>> pq;

        // Initializing the dist array and the ways array
        // along with their first indices.
        vector<int> dist(n, INT_MAX), ways(n, 0);
        dist[0] = 0;
        ways[0] = 1;
        pq.push({0, 0});

        // Define modulo value
```

```
        int mod = (int)(1e9 + 7);

        while (!pq.empty())
        {
            int dis = pq.top().first;
            int node = pq.top().second;
            pq.pop();

            for (auto it : adj[node])
            {
                int adjNode = it.first;
                int edW = it.second;

                // This ‘if’ condition signifies that this is the first
                // time we’re coming with this short distance, so we push
                // in PQ and keep the no. of ways the same.
                if (dis + edW < dist[adjNode])
                {
                    dist[adjNode] = dis + edW;
                    pq.push({dis + edW, adjNode});
                    ways[adjNode] = ways[node];
                }

                // If we again encounter a node with the same short distance
                // as before, we simply increment the no. of ways.
                else if (dis + edW == dist[adjNode])
                {
                    ways[adjNode] = (ways[adjNode] + ways[node]) % mod;
                    // why not push in queue ? we are using a priority queue
                    // => will not update unless it itself is the shortest path
                }
            }
        }
        // Finally, we return the no. of ways to reach
        // (n-1)th node modulo 10^9+7.
        return ways[n - 1] % mod;
    }
};
```

Difference between vector<int> adj[K] and vector<vector<int>> adj(k)

- Here first one is a static allocation where k has to be known during compilation not execution. second one is dynamic where k is declared during runtime

- A graph consisting of N nodes and M edges. A 2D array E. In each row here are two Integers u, and v, denoting an edge of length 1 between nodes u, and v

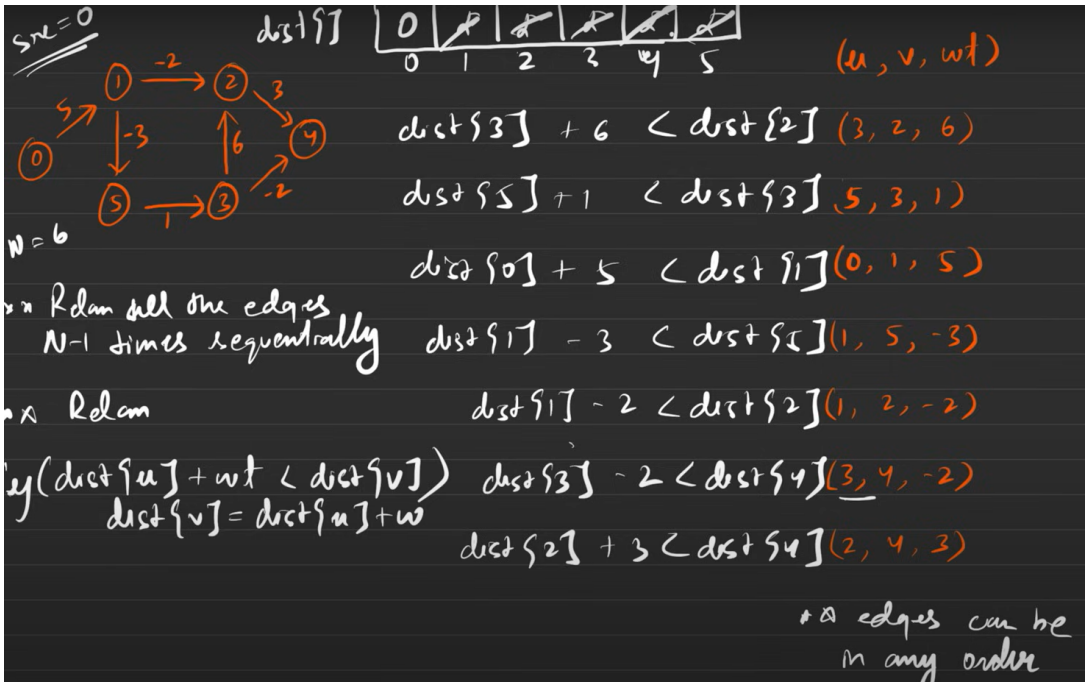
- An array A of length N which means there are A[i] people at the ith node. You need to bring at-least K people at node 1.
- The minimum sum of distance travelled by people such that there are at least K people at node 1.
- Return -1 if it is not possible to bring K people to node 1.

```
int minDistance(int N, int M, vector<pair<int, int>>& E, vector<int>& A, int K) {
    // Step 1: Build the graph using an adjacency list
    vector<vector<int>> graph(N);
    for (auto& edge : E) {
        int u = edge.first - 1; // Converting to 0-indexed
        int v = edge.second - 1;
        graph[u].push_back(v);
        graph[v].push_back(u);
    }

    // Step 2: Perform BFS from node 1 (index 0) to find shortest paths
    vector<int> distances(N, -1);
    queue<int> q;
    distances[0] = 0;
    q.push(0);

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

        for (int neighbor : graph[node]) {
            if (distances[neighbor] == -1) { // Not visited
                distances[neighbor] = distances[node] + 1;
                // Use this BFS way to calculate the dis only when
                // the weight are all equal else use dijkstras algo
                q.push(neighbor);
            }
        }
    }

    // Step 3: Sort nodes by distance from node 1
    vector<pair<int, int>> sortedNodes;
    for (int i = 0; i < N; i++) {
        if (distances[i] != -1) { // Consider only reachable nodes
            sortedNodes.push_back({distances[i], i});
        }
    }
    sort(sortedNodes.begin(), sortedNodes.end());

    // Step 4: Collect people greedily
    int totalPeople = 0;
```

```
    int totalDistance = 0;

    for (auto& [distance, node] : sortedNodes) {
        if (totalPeople >= K) break;
        int needed = min(K - totalPeople, A[node]);
        totalPeople += needed;
        totalDistance += needed * distance;
    }

    // Step 5: Check if we gathered enough people
    if (totalPeople < K) return -1;

    return totalDistance;
}
```

Bellman Ford Algo :

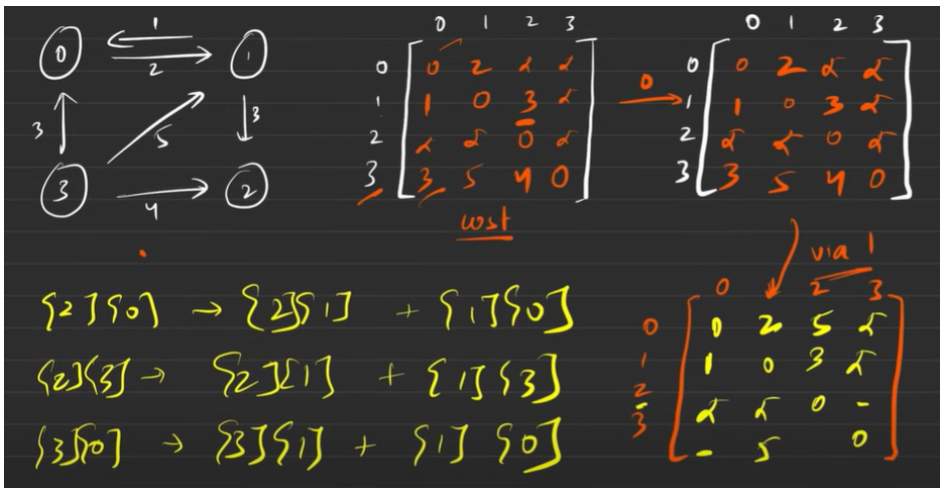
- Helps in detecting negative cycles (TLE happens because of this in dijks)
- Applicable on directed graphs, if undirected is given simply convert
- Relax all the edges n-1 times sequentially
- Relax ? {Simply process through all edges and keep updating their distances}
- The least possible distance will be attained after n-1 times for every node ⇒ if we do another iteration and the distances decrease ⇒ negative cycle exists



```cpp
vector<int> bellman_ford(int V, vector<vector<int>>& edges, int S) {
    vector<int> dist(V, 1e8);
    dist[S] = 0;
    // V x E
    for(int i = 0;i<V-1;i++) {
        for(auto it : edges) {
            int u = it[0];
            int v = it[1];
            int wt = it[2];
            if(dist[u] != 1e8 && dist[u] + wt < dist[v]) {
                dist[v] = dist[u] + wt;
            }
        }
    }
    // Nth relaxation to check negative cycle
    for(auto it : edges) {
        int u = it[0];
        int v = it[1];
        int wt = it[2];
        if(dist[u] != 1e8 && dist[u] + wt < dist[v]) {
            return {-1};
        }
    }


    return dist;
}
```

Floyd Warshall Algo :
* Multisource Shortest Path algo, which also helps in detecting negative edge cycles

- say we are reaching j from i via k them i → j = i → k + k → j
- now we can use i → j to compute something else where j is in middle
- DP {using pre-computed values}
- It is a brute force approach, where we keep finding and updating dis from i → j via different intermediate node k {including i and j}
- If dis of any node to itself is not 0 ⇒ negative cycle




```cpp
void shortest_distance(vector<vector<int>>&matrix) {
        int n = matrix.size();
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (matrix[i][j] == -1) {
```

```
                    matrix[i][j] = 1e9; // no edge
                }
                if (i == j) matrix[i][j] = 0; // source -> source
            }
        }

        for (int k = 0; k < n; k++) {
            for (int i = 0; i < n; i++) {
                for (int j = 0; j < n; j++) {
                    matrix[i][j] = min(matrix[i][j], matrix[i][k] + matrix[k][j]);
                    // k -> via {intermediate}
                }
            }
        }

        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (matrix[i][j] == 1e9) {
                    matrix[i][j] = -1; // negative cycle
                }
            }
        }
    }
};
```

- Given n cites, there distances in between. there is a max_dis one can travel from city to city. return in which city there are least cities to travel. If more than one city possible return high node.

```cpp
int findCity(int n, int m, vector<vector<int>>& edges,
             int distanceThreshold) {
      vector<vector<int>> dist(n, vector<int> (n, INT_MAX));
      for(auto it : edges) {
          dist[it[0]][it[1]] = it[2];
          dist[it[1]][it[0]] = it[2];
      }
      for(int i = 0;i<n;i++) dist[i][i] = 0;
      for(int k = 0;k<n;k++) {
          for(int i = 0;i<n;i++) {
              for(int j = 0;j<n;j++) {
                  if(dist[i][k] == INT_MAX || dist[k][j] == INT_MAX)
                    continue;
                  dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
              }
          }
      }

      int cntCity = n;
      int cityNo = -1;
      for(int city = 0; city < n; city++) {
          int cnt = 0;
          for(int adjCity = 0; adjCity < n; adjCity++) {
              if(dist[city][adjCity] <= distanceThreshold)
                cnt++;
          }

          if(cnt <= cntCity) {
              cntCity = cnt;
              cityNo = city;
          }
      }
      return cityNo;

}
```

- in cnt ≤ cntCity {< ⇒ new city and = ⇒ same number of cites that can be travelled but higher city number}

A tree in which we have N nodes and N-1 edges and all nodes are reachable from each other (Same connected component) → **Spanning Tree**

- For a give graph there could be multiple spanning trees, the one which has sum of all edge weights min ⇒ Min Spanning Tree

---

Prim's algorithm is a greedy algorithm used to find the minimum spanning tree (MST) of a weighted, undirected graph.

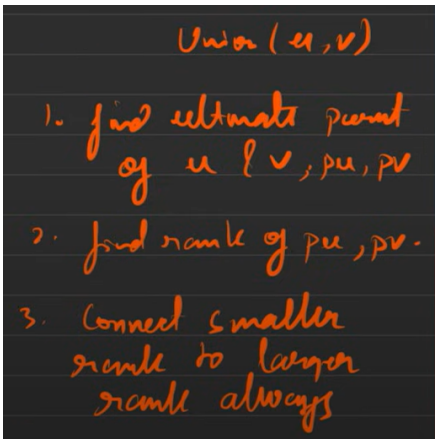
- At each step, the algorithm adds the smallest weight edge that connects a node in the MST to a node outside the MST. The key idea here is to always choose the edge with the minimum weight that expands the MST without forming a cycle.

```cpp
int spanningTree(int V, vector<vector<int>> adj[])
    {
        priority_queue<pair<int, int>,
                       vector<pair<int, int> >, greater<pair<int, int>>> pq;

        vector<int> vis(V, 0);
        // {wt, node}
        pq.push({0, 0});
        int sum = 0;
        while (!pq.empty()) {
```

```
            auto it = pq.top();
            pq.pop();
            int node = it.second;
            int wt = it.first;

            if (vis[node] == 1) continue;
            // add it to the mst
            vis[node] = 1;
            sum += wt;
            for (auto it : adj[node]) {
                int adjNode = it[0];
                int edW = it[1];
                if (!vis[adjNode]) { // greedily push the lightest edge into MST
                    pq.push({edW, adjNode});
                }
            }
        }
        return sum;
    }
};
```

Disjoint Sets are used in dynamic Graphs :

- uses-cases ? find parent and union {rank, size}
- We use Union to represent an edge between two nodes



- Now to check if two nodes are of same component ⇒ both of their ultimate parent should be same
- Now finding ultimate traversal requires us to travel the entire tree we've formed ⇒ logN. But we can make it constant by path compression, When checking for the first time we've to traverse logN then directly change the link of parent to ultimate parent.
- So basically union and find parent take constant time

- When ever we call find parent to one of the node, the node and its parent nodes also get path compressed
- Will the rank change ? NO

```
class DisjointSet {
    vector<int> rank, parent, size;
public:
    DisjointSet(int n) {
        rank.resize(n + 1, 0);
        parent.resize(n + 1);
        size.resize(n + 1);
        for (int i = 0; i <= n; i++) {
            parent[i] = i;
            size[i] = 1;
        }
    }

    int findUPar(int node) {
        if (node == parent[node])
            return node;
        return parent[node] = findUPar(parent[node]);
    }

    void unionByRank(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (rank[ulp_u] < rank[ulp_v]) {
            parent[ulp_u] = ulp_v;
        }
        else if (rank[ulp_v] < rank[ulp_u]) {
            parent[ulp_v] = ulp_u;
        }
        else {
            parent[ulp_v] = ulp_u;
            rank[ulp_u]++;
        }
    }

    void unionBySize(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (size[ulp_u] < size[ulp_v]) {
            parent[ulp_u] = ulp_v;
            size[ulp_v] += size[ulp_u];
        }
```

```
        else {
            parent[ulp_v] = ulp_u;
            size[ulp_u] += size[ulp_v];
        }
    }
};
int main() {
    DisjointSet ds(7);
    ds.unionBySize(1, 2);
    ds.unionBySize(2, 3);
    ds.unionBySize(4, 5);
    ds.unionBySize(6, 7);
    ds.unionBySize(5, 6);
    // if 3 and 7 same or not
    if (ds.findUPar(3) == ds.findUPar(7)) {
        cout << "Same\n";
    }
    else cout << "Not same\n";

    ds.unionBySize(3, 7);

    if (ds.findUPar(3) == ds.findUPar(7)) {
        cout << "Same\n";
    }
    else cout << "Not same\n";
    return 0;
}
```

- So why do we connect smaller to larger if we just want to connect those two components ? ⇒ if bigger gets attached to smaller the time to compute path compression increases.
- As rank gets distorted it doesn’t make sense to do this, rather we do by keeping track of size of component

---

Kruskal's Algo :

```
int spanningTree(int V, vector<vector<int>> adj[])
    {
        // 1 - 2 wt = 5
        /// 1 - > (2, 5)
        // 2 -> (1, 5)

        // 5, 1, 2
        // 5, 2, 1
        vector<pair<int, pair<int, int>>> edges;
        for (int i = 0; i < V; i++) {
            for (auto it : adj[i]) {
```

```
                int adjNode = it[0];
                int wt = it[1];
                int node = i;

                edges.push_back({wt, {node, adjNode}});
            }
        }
        DisjointSet ds(V);
        sort(edges.begin(), edges.end());
        int mstWt = 0;
        for (auto it : edges) {
            int wt = it.first;
            int u = it.second.first;
            int v = it.second.second;

            if (ds.findUPar(u) != ds.findUPar(v)) {
                mstWt += wt;
                ds.unionBySize(u, v);
            }
        }

        return mstWt;
    }
```

In prims we always have a single component and we grow it. In Kruskal’s we have multiple small small components which end up getting added to make one single big component

- In finding no of provinces, rather than doing BFS, we can just build graph on DISJOINT set and then iterate over all nodes and find number of different ultimate parents and we are done with it.

No.of edges to add to make it a single whole connected component by removing and changing already existing edges

```
 void unionBySize(int u, int v) {
    int ulp_u = findUPar(u);
    int ulp_v = findUPar(v);
    if (ulp_u == ulp_v) return; // INSTEAD OF RETURNING HERE WE CAN STORE IT AS AN EX
    if (size[ulp_u] < size[ulp_v]) {
        parent[ulp_u] = ulp_v;
        size[ulp_v] += size[ulp_u];
    }
    else {
        parent[ulp_v] = ulp_u;
        size[ulp_u] += size[ulp_v];
```

```
    }
}
```

Accounts Merged :

- Given a list which starts with owners name and then all his mails.
- If in the upcoming lists if we find a common mail of the same owner we just merge them

```
vector<vector<string>> accountsMerge(vector<vector<string>> &details) {
  int n = details.size();
  DisjointSet ds(n);
  sort(details.begin(), details.end());
  unordered_map<string, int> mapMailNode;
  for (int i = 0; i < n; i++) {
      for (int j = 1; j < details[i].size(); j++) {
          string mail = details[i][j];
          if (mapMailNode.find(mail) == mapMailNode.end()) {
              mapMailNode[mail] = i;
          }
          else {
              ds.unionBySize(i, mapMailNode[mail]);
          }
      }
  }

  vector<string> mergedMail[n];
  for (auto it : mapMailNode) {
      string mail = it.first;
      int node = ds.findUPar(it.second);
      mergedMail[node].push_back(mail);
  }

  vector<vector<string>> ans;

  for (int i = 0; i < n; i++) {
      if (mergedMail[i].size() == 0) continue;
      sort(mergedMail[i].begin(), mergedMail[i].end());
      vector<string> temp;
      temp.push_back(details[i][0]);
      for (auto it : mergedMail[i]) {
          temp.push_back(it);
      }
      ans.push_back(temp);
  }
  sort(ans.begin(), ans.end());
```

```
  return ans;
}
```

Number Of Islands 2 :

Given a matrix, we keep adding lands in water. On addition the lands might come together to form one big island. keep returning the current number of islands while we are adding land.

```
class Solution {
private:
    bool isValid(int adjr, int adjc, int n, int m) {
        return adjr >= 0 && adjr < n && adjc >= 0 && adjc < m;
    }
public:
    vector<int> numOfIslands(int n, int m,
                             vector<vector<int>> &operators) {
        DisjointSet ds(n * m);
        int vis[n][m];
        // To check if adjascent land is island or node and to ensure
        // duplicate additions are ignored
        memset(vis, 0, sizeof vis);
        int cnt = 0;
        vector<int> ans;
        for (auto it : operators) {
            int row = it[0];
            int col = it[1];
            if (vis[row][col] == 1) {
                ans.push_back(cnt);
                continue;
            }
            vis[row][col] = 1;
            cnt++;
            // row - 1, col
            // row , col + 1
            // row + 1, col
            // row, col - 1;
            int dr[] = { -1, 0, 1, 0};
            int dc[] = {0, 1, 0, -1};
            for (int ind = 0; ind < 4; ind++) {
                int adjr = row + dr[ind];
                int adjc = col + dc[ind];
                if (isValid(adjr, adjc, n, m)) {
                    if (vis[adjr][adjc] == 1) { // we found a land connect it!
                        int nodeNo = row * m + col; // decompress 2d to 1d
                        int adjNodeNo = adjr * m + adjc;
                        if (ds.findUPar(nodeNo) != ds.findUPar(adjNodeNo)) {
                            cnt--;
```

```
                            ds.unionBySize(nodeNo, adjNodeNo);
                        }
                    }
                }
            }
            ans.push_back(cnt);
        }
        return ans;
    }
};
```

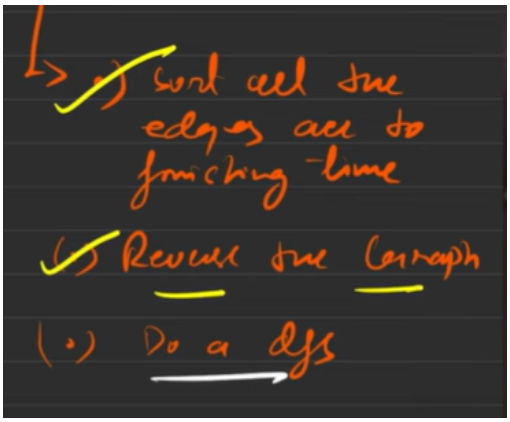
- Given a 2d matrix with 0 and 1’s. You can turn at most one 0 → 1. After doing this return the max number of connected land as Island.

```
class Solution {
private:
    bool isValid(int newr, int newc, int n) {
        return newr >= 0 && newr < n && newc >= 0 && newc < n;
    }
public:
    int MaxConnection(vector<vector<int>>& grid) {
        int n = grid.size();
        DisjointSet ds(n * n);
        // step - 1 (Make DSU)
        for (int row = 0; row < n ; row++) {
            for (int col = 0; col < n ; col++) {
                if (grid[row][col] == 0) continue;
                int dr[] = { -1, 0, 1, 0};
                int dc[] = {0, -1, 0, 1};
                for (int ind = 0; ind < 4; ind++) {
                    int newr = row + dr[ind];
                    int newc = col + dc[ind];
                    if (isValid(newr, newc, n) && grid[newr][newc] == 1) {
                        int nodeNo = row * n + col;
                        int adjNodeNo = newr * n + newc;
                        ds.unionBySize(nodeNo, adjNodeNo);
                    }
                }
            }
        }
        // step 2 (We will not add the change node directly to DSU rather
        // we make the attributes of DSU public and access them here)
        int mx = 0;
        for (int row = 0; row < n; row++) {
            for (int col = 0; col < n; col++) {
                if (grid[row][col] == 1) continue;
                int dr[] = { -1, 0, 1, 0};
```

```
                int dc[] = {0, -1, 0, 1};
                set<int> components;
                for (int ind = 0; ind < 4; ind++) {
                    int newr = row + dr[ind];
                    int newc = col + dc[ind];
                    if (isValid(newr, newc, n)) {
                        if (grid[newr][newc] == 1) {
                            components.insert(ds.findUPar(newr * n + newc));
                        }
                    }
                }
                int sizeTotal = 0;
                for (auto it : components) {
                    sizeTotal += ds.size[it];
                }
                mx = max(mx, sizeTotal + 1);
            }
        }
        for (int cellNo = 0; cellNo < n * n; cellNo++) {
            mx = max(mx, ds.size[ds.findUPar(cellNo)]);
        }
        return mx;
    }
};
```

No of strongly connected components

Kosaraju’s Algo



By reversing the graph we are now restricted to a single CC, ⇒ Traversing to other CC is not possible.

```
class Solution {
private:
    void dfs(int node, vector<vector<int>>& adj, vector<int>& vis, vector<int>& finis
```

```
        vis[node] = 1;
        for (int it : adj[node]) {
            if (!vis[it]) {
                dfs(it, adj, vis, finishOrder);
            }
        }
        finishOrder.push_back(node);
    }

    void dfs2(int node, vector<vector<int>>& adj, vector<int>& vis) {
        vis[node] = 1;
        for (int it : adj[node]) {
            if (!vis[it]) {
                dfs2(it, adj, vis);
            }
        }
    }

public:
    // Function to find the number of strongly connected components in the graph.
    int kosaraju(int V, vector<vector<int>>& adj) {
        vector<int> vis(V, 0);
        vector<int> finishOrder;
        for (int i = 0; i < V; i++) {
            if (!vis[i]) {
                dfs(i, adj, vis, finishOrder);
            }
        }

        // Creating the transpose graph (adjT)
        vector<vector<int>> adjT(V);
        for (int i = 0; i < V; i++) {
            vis[i] = 0;
            for (int it : adj[i]) {
                adjT[it].push_back(i);
            }
        }

        // Last DFS using the finish order
        int scc = 0;
        for (int i = V - 1; i >= 0; i--) {
            int node = finishOrder[i];
            if (!vis[node]) {
                scc++;
                dfs2(node, adjT, vis);
            }
        }
        return scc;
```

```
    }
};
```

Bridges in Graph :

Tarjan's Algorithm

- Tarjan's algorithm for finding strongly connected components (SCCs) in a directed graph makes use of back edges
- If during DFS you encounter a node that is already on the stack (part of the current path of DFS), it indicates a back edge. This back edge helps in updating the low values to ensure that all reachable nodes (in the same SCC) are considered.
- if the low[] of next node is less than the one which calls dfs for it, then there is a other node through which that node can be reached and which appeared before the one which calls the dfs

```
class Solution {
private:
    int timer = 1;
    void dfs(int node, int parent, vector<int> &vis,
             vector<int> adj[], int tin[], int low[], vector<vector<int>> &bridges) {
        vis[node] = 1;
        tin[node] = low[node] = timer;
        timer++;
        for (auto it : adj[node]) {
            if (it == parent) continue;
            if (vis[it] == 0) {
                dfs(it, node, vis, adj, tin, low, bridges);
                low[node] = min(low[it], low[node]);
                // node --- it
                if (low[it] > tin[node]) {
                    bridges.push_back({it, node});
                }
            }
            else {
                low[node] = min(low[node], low[it]);
            }
        }
    }
public:
    vector<vector<int>> criticalConnections(int n,
    vector<vector<int>>& connections) {
        vector<int> adj[n];
        for (auto it : connections) {
            int u = it[0], v = it[1];
            adj[u].push_back(v);
            adj[v].push_back(u);
        }
```

```
        vector<int> vis(n, 0);
        int tin[n];
        int low[n];
        vector<vector<int>> bridges;
        dfs(0, -1, vis, adj, tin, low, bridges);
        return bridges;
    }
};
```

Articulation Point : (Cut Vertex)

- it's a critical node that, if removed, would split the graph into separate parts.

```
class Graph {
public:
    Graph(int vertices);
    void addEdge(int u, int v);
    void findArticulationPoints();

private:
    int V; // Number of vertices
    int timer; // Timer to keep track of discovery times
    vector<vector<int>> adj; // Adjacency list representation of the graph
    vector<int> tin, low, vis, mark;
    // Arrays to store discovery time, low values,
    // visited nodes, and articulation points

    void dfs(int node, int parent);
};

Graph::Graph(int vertices) {
    this->V = vertices;
    adj.resize(V);
    tin.assign(V, -1);
    low.assign(V, -1);
    vis.assign(V, 0);
    mark.assign(V, 0);
    timer = 0;
}

void Graph::addEdge(int u, int v) {
    adj[u].push_back(v);
    adj[v].push_back(u); // Because the graph is undirected
}

void Graph::dfs(int node, int parent) {
    vis[node] = 1;
    tin[node] = low[node] = timer++;
```

```
    int children = 0;

    for (auto it : adj[node]) {
        if (it == parent) continue; // Skip the parent node

        if (!vis[it]) { // If it is not visited
            dfs(it, node);
            low[node] = min(low[node], low[it]);

            if (low[it] >= tin[node] && parent != -1) {
                mark[node] = 1; // Mark node as an articulation point
            }
            children++;
        } else {
            low[node] = min(low[node], tin[it]); // Update low[node] for back edges
        }
    }

    // If the node is the root of DFS and has more than one child
    if (parent == -1 && children > 1) {
        mark[node] = 1;
    }
}

void Graph::findArticulationPoints() {
    for (int i = 0; i < V; i++) {
        if (!vis[i]) {
            dfs(i, -1);
        }
    }

    cout << "Articulation points are: ";
    for (int i = 0; i < V; i++) {
        if (mark[i]) {
            cout << i << " ";
        }
    }
    cout << endl;
}
```