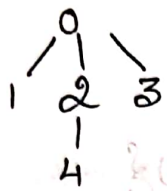
Graphs

① BFS Traversal

I/P :-



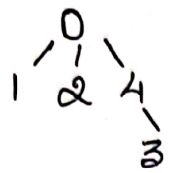
O/P :- 0 1 2 3 4

Code

```
vector<int> bfs(int V, vector<int> adj[]){
    vector<int> res;
    vector<int> visited(V,0);

    queue<int> q;
    q.push(0);

    visited[0]=1;

    while(!q.empty()){
        int temp = q.front();
        res.push_back(temp);
        for(auto x: adj[temp]){
            if(!visited[x]){
                visited[x]=1;
                q.push(x);
            }
        }
        q.pop();
    }
    return res;
}
```

#② DFS Traversal

I/P:-



O/P:- 0, 1, 2, 4, 3

Code

```
vector<int> dfs(int V, vector<int> adj[]){
    vector<int> visited(V, 0);
    vector<int> res;
    for(int i=0; i<V; i++){ // Use for loop to check
        if(!visited[i])          for disconnected
            dfsUtil(i, adj, visited, res);   graph comp.
    }
    return res;
}

void dfsUtil(int src, vector<int> adj[], vector<int> &vis,
                                    vector<int> & res){
    visited[src] = 1;
    res.push_back(src)
    for(auto x: adj[src]){
        if(!visited[x])
            dfsUtil(x, adj, vis, res);
    }
}
```

#③ Rat in a Maze Problem (Search in a Maze)

I/P:- $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 \end{bmatrix}$

O/P:- DDRDRR DRDDRR

① Cell with 0 are blocked
② Movement in 4 directions.

Code

```
vector<string> res;
vector<string> findpath(vector<vector<int>>& m, int n){
    res.clear();   // always perform when declared globally
    vector<vector<int>> visited(n, vector<int>(n,0));
    if(m[0][0]==0 || m[n-1][n-1]==0)
        return res;

    string s="";
    dfsUtil(0,0,s,m,n,visited);
    sort(res.begin, res.end);
    return res;
}

void dfsUtil(int i, int j, string s, &m, int n, &visited){
    if(i<0 || j<0 || i>=n || j>=n)   // Search within bounds
        return;
    if(m[i][j]==0 || visited[i][j]==1)   // Prevent going in a loop.
        return;
    // Destination reached
    if(i==n-1 && j==n-1){
        res.push_back(s);
        return;
    }
    visited[i][j]=1;

    dfsUtil(i-1, j, s+"U", m, n, visited);
    dfsUtil(i+1, j, s+"D", m, n, visited);
    dfsUtil(i, j-1, s+"L", m, n, visited);
    dfsUtil(i, j+1, s+"R", m, n, visited);
    visited[i][j]=0;   // Backtrack.
}
```

#④ Flood Fill Algorithm

I/P:- $\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$ O/P:- $\begin{bmatrix} 2 & 2 & 2 \\ 2 & 2 & 0 \\ 2 & 0 & 1 \end{bmatrix}$

Code

```
vector<vector<int>> floodFill(vector<vector<int>>& image, int sr,
                              int sc, int newColor){
    int x = image.size();
    int y = image[0].size();
    vector<vector<int>> visited (x, vector<int>(y, 0));
    int oldColor = image[sr][sc];
    dfsUtil(sr, sc, x, y, image, visited, oldColor, newColor);
    return image;
}

void dfsUtil(int i, int j, int x, int y, &image, &visited,
             int oldColor, int newColor){
    if(i<0 || j<0 || i>=x || j>=y)
        return;
    if(image[i][j] != oldColor || visited[i][j]==1)
        return;
    visited[i][j]=1;
    image[i][j] = newColor;
    dfsUtil(i-1, j, x, y, image, visited, oldColor, newColor);
    dfsUtil(i+1, j, ...
    dfsUtil(i, j-1, ...
    dfsUtil(i, j+1, ....
}
```

#⑤ Make wired Connections

① So connect each node - find out number of ~~connected~~ components

② Connections = components - 1.

Code

```
int makeConnected(int n, vector<vector<int>>& connections){
    vector<int> adj[n];
    int m = connections.size();
    if(m<n-1)      // Connections available < (n-1)
        return -1;      // Not possible
    for(auto i: connections){   // Make adj list.
        adj[i[0]].push_back(i[1]);
        adj[i[1]].push_back(i[0]);
    }

    vector<int> visited(n,0);
    int countConnected = 0;
    for(int i=0; i<n; i++){
        if(!visited[i]){
            countConnected++;
            dfsUtil(i, adj, visited);
        }
    }
    return countConnected;
}
void dfsUtil(int src, vector<int> adj[], &visited){
    visited[src] = 1;
    for(auto x: adj[src]){
        if(!visited[x])
            dfsUtil(x, adj, visited);
    }
}
```

#⑥ Dijkstra Algorithm

① Shortest path from given src to all vertices
② No -ve wgt in graph.

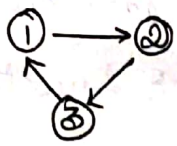
Use Set; set stores data based on asc. sorting of data.

③ If smaller dist found
  ** If vertex not in set; insert directly
  else
  remove and insert.

Code

```
vector<int> djikstra(int V, vector<vector<int>> adj[], int S){
    vector<int> distance(V, INT_MAX);
    distance[S] = 0;
    set<pair<int,int>> s;          // Set of pairs has values
    s.insert({0, S});              sorted acc. to first value
    while(!s.empty()){
        auto it = s.begin();       // First value has shortest
        pair<int,int> num = *it;   dist.
        int dist = num.first;
        int v = num.second;
        s.erase(it)
                                   // x[0] = vertex
        for(auto x: adj[v]){       // x[1] = distance
            if(dist + x[1] < distance[x[0]]){
                auto ii = s.find({distance[x[0]], x[0]})
                if (ii==s.end){    // Not found → Insert.
                ① || s.insert({dist + x[1], x[0]});
                    distance[x[0]] = dist + x[1];
                } else {           // Not found → Remove & insert
                    s.erase(ii);
                    Insert & update ①
                }
            }
        }
    }
    return distance;
}
```

#⑦ Detect Cycle in Directed Graph (DFS)

I/P - 

O/P:- True

① Maintain an order array along with vis.
② If node already vis and present in order array it means it was already in path. Cycle found

Code

```
bool isCyclic(int V, vector<int> adj[]){
    vector<int> order(V,0);
    vector<int> visited(V,0);
    for(int i=0; i<V; i++){
        if(!visited[i]){
            bool check = dfsUtil(i, adj, order, visited);
            if(check == true)
                return true;
        }
    }
    return false;
}

bool dfsUtil(int src, vector<int> adj[], & visited, & order){
    visited[src]=1;
    order[src]=1;
    for(auto x : adj[src]){
        if(!visited[x]){
            bool check = dfsUtil(x, adj, visited, order);
            if(check == true)
                return true;
        }
        if(visited[x] && order[x] == 1)   // Visited and already present in order so cycle
            return true;
    }
    order[src]=0;   // Back Track.
    return false;
}
```

#⑧ Detect Cycle in Undirected Graph (DFS)

① Maintain a parent variable for each node.

② If a node is visited but it is not parent; then cycle detected.

Code

```
bool isCycle(int V, vector<int> adj[]){
    vector<int> visited(V, 0);
    for(int i=0; i<V; i++){
        if(!visited[i]){                    parent of initial
            bool check = dfsUtil(i, -1, adj, visited);
            if(check == true)
                return true;
        }
    }
    return false;
}

bool dfsUtil(int src, int par, vector<int> adj[], & visited
    visited[src] = 1;
    for(auto x : adj[src]){
        if(!visited[x]){
            bool check = dfsUtil(x, src, adj, visited);
            if(check == true)
                return true;
        }
        //Visited and x != parent.
        else if(visited[x] && par != x)
            return true;
    }
    return false;
}
```

#9 Topological Sort (Kahn's Algo)

① Sorting based on the number of indegrees.

Code

```
vector<int> topoSort(int V, vector<int> adj[]){
    vector<int> res;
    vector<int> inDeg(V,0);

    queue<int> q
    for(int i=0; i<V; i++){
        for(auto x: adj[i]){
            inDeg[x]++;
        }
    }
    //Push nodes whose inDegree == 0
    for(int i=0; i<V; i++){
        if(inDeg[i] == 0)
            q.push(i);
    }

    while(!q.empty()){
        int temp = q.front();
        res.push_back(temp);
        q.pop();
        for(auto x: adj[temp]){
            inDeg[x]--;
            if(inDeg[x] == 0)
                q.push(x);
        }
    }
    return res;
}
```

#⑩ Course Schedule - I / II

Check whether all courses can be taken.
Print the courses to be taken in order (II)

Code

```
bool canFinish(int numCourses, vector<vector<int>>
                                    &prerequisites)
    vector<int> adj[numCourses];
    vector<int> in(numCourses, 0);
    for(auto x: prerequisites){
        adj[x[1]].push_back(x[0]);
        in[x[0]]++;
    }

    queue<int> q;                    // II - vector<int> res;
    int coursesProcessed = 0;
    for(int i=0; i< numCourses; i++){
        if(in[i] == 0){
            coursesProcessed++;
            q.push(i);
        }
    }

    while(!q.empty()){
        int temp = q.front(); // II - res.push_back(temp)
        q.pop;
        for(auto x: adj[temp]){
            in[x]--;
            if(in[x]--){
                coursesProcessed++;
                q.push(x);          // II - if(numCourses != coursesProcessed)
            }                                  return {};
        }                                  else
    }                                      return res;
    return (coursesProcessed == numProcessed);
}
```

Prim's Algorithm Minimum Spanning Tree

(11)

① Select the min cost edge connected to existing edges.

Code

```
int spanningTree(int V, vector<vector<int>> adj[]){
    vector <bool> inMST(V, false);
    int ans = 0;

    priority queue < pair>  minHeap;
    minHeap.push({0, 0});    // distance followed by
                                vertex.
    while(!minHeap.empty()){
        auto best = minHeap.top();
        minHeap.pop();
        int dist = best.first;
        int toVertex = best.second;

        if(inMST[toVertex])   // Already present skip
            continue;

        else{
            inMST[toVertex] = true;
            ans += dist;
            for(auto v: adj[toVertex]){
                if(!inMST[v[0]])
                    minHeap.push({v[1], v[0]})
            }
        }
    }
    return ans;
}
```

Similary Same Problem -

Commutable Islands

(InterviewBit)

#12 Bellman Ford Algorithm

① Works well when there are negative weights present.

② However if -ve wgt cycle is present will not work

Dist. b/w source vertex to all other vertices.

Code

```
vector<int> dist_BF(int V, vector<vector<int>> adj, int S){
    vector<int> dis(V, 100000000);
    dis[S] = 0;        // Source Vertex.

    // Relax for (n-1) times.
    for(int i=0; i<V-1; i++){
        for(int j=0; j<adj.size(); j++){
            int u = adj[j][0];    // Initial Vertex
            int v = adj[j][1];    // Final Vertex
            int w = adj[j][2];    // wgt.
            if(dis[u] != 1e8 && dis[u]+w < dis[v])
                dis[v] = dis[u]+w;
        }
    }
    return dis;
}
```

Finding -ve wgt cycle;

After (n-1) relaxations; relax for 1 more time

If dis of any vertex changes; then there is a -ve wgt cycle present

#(13) Implement Floyd-Warshall Algorithm

The algorithm is used to find shortest dist b/w every pair of vertices in a given edge weighted directed graph.

Mathematical relationship:-

$$A^{k}[i,j] = \min\{ A^{k-1}[i,j] ; A^{k-1}[i,k] + A^{k-1}[k,j] \}$$

$k = 0$ to (n) [no. of vertices];

Generally if a path does not exist b/w i and j; it is shown by ∞.

Code

```
void floydWarshall(vector<vector<int>> &mat){
    int n= matrix.size();
    for(int k=0; k<n; k++){
        for(int i=0; i<n; i++){
            for(int j=0; j<n; j++){
                //Make sure int. paths are not ∞.
                if(mat[i][k]!=INT_MAX && mat[k][j]!=∞){
                    mat[i][j] = min( mat[i][j],
                                     mat[i][k] + mat[k][j] );
                }
            }
        }
    }
}
```

If inst. of ∞; paths that are not connected by are presented by -1

Transform all '-1' to INT_MAX; perform calculation. Replace INT_MAX by -1 in final answer.

#⑭ Travelling Salesmans Problem

Given a set of cities and dist. b/w every pair of cities, problem is to find the shortest path that visits every city exactly once and returns to the starting point. (Find min. wgt path in Hamiltonian G

Code (O(n-1)!)

```
int total_cost(vector<vector<int>> cost){
    int V = cost.size();
    int start = 0;

    vector<int> vertex;
    for(int i=0; i<V; i++){
        if(i != start)
            vertex.push_back(i);
    }

    int minPath = INT_MAX;

    do{
        int currPath = 0;
        int k = start;
        for(int i=0; i<vertex.size(); i++){
            currPath += cost[k][vertex[i]];
            k = vertex[i];
        }
        // Include last node to src (to return to starting pt)
        currPath += cost[k][start];
        minPath = min(minPath, currPath);
    }
    while(next_permutation(vertex.begin(), vertex.
                                // Do for all possible
    return minPath;             permutation of path
}
```

Calculate cost for this path from start

#15 Count Number of Islands

I/P:- $\begin{bmatrix} "1" & "1" & "1" & "1" & "0" \\ "1" & "1" & "0" & "1" & "0" \\ "1" & "1" & "0" & "0" & "0" \\ "0" & "0" & "0" & "0" & "0" \end{bmatrix}$

O/P:- 1

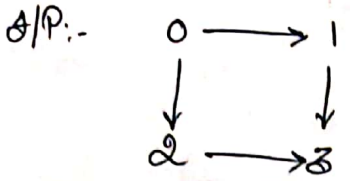
① Find no. of connected components in matrix using DFS.

Code

```
int numIslands(vector<vector<char>>& grid){
    int n = grid.size();
    int m = grid[0].size();
    vector<vector<bool>> vis(n, vector<bool>(m, false));
    int countIslands = 0;
    for(int i=0; i<n; i++){
        for(int j=0; j<m; j++){
            if(!visited[i][j] && grid[i][j]=="1"){
                countIslands++;
                dfs(i,j,n,m,vis,grid);      // max area = max(maxarea, dfs());
            }
        }
    }
    return countIslands;
}
void dfs(int i, int j, int n, int m, & vis, & grid){   // Return int
    if(i<0 || i>=n || j<0 || j>=m) return;
    if(visited[i][j] || grid[i][j] != "1") return;
    visited[i][j] = true;
    dfs(i+1,j,n,m,vis,grid);
    dfs(i-1,j,n,m,vis,grid);
    dfs(i,j+1,n,m,vis,grid);
    dfs(i,j-1,n,m,vis,grid);
}
```

```
// int area = 1
area += dfs(i+1,j...
"   +=
"   +=
"   +=
return area.
```

#⑯ All Paths from Source to Target in DAG



Code

```
vector<vector<int>> paths;
vector<vector<int>> allPaths(vector<vector<int>>& graph){
    vector<int> p;
    dfs(0, graph, p);
    return paths;
}

void dfs(int v, &graph, vector<int> p){
    p.push_back(v);
    if(v== graph.size()-1){
        paths.push_back(p);
        p.clear();          // Essential.
    }
    for(int i=0; i< graph[v].size(); i++){
        dfs(graph[v][i], graph, p);
    }
}
```

17) Alien Dictionary (N words and K starting alphabets find order in alien dictionary)

I/P:- {"caa", "aaa", "aab"}

O/P:- "c", "a", "b"

① Idea is to use Topo Sort.

Code

```
string findOrder(string dict[], int N, int K){
    vector<int> adj[K], inDeg(K,0);
    for(int i=0; i<N-1; i++){
        string w1 = dict[i];
        string w2 = dict[i+1];
        for(int j=0; j<min(w1.length(), w2.length()); j++){
            if(w1[j] != w2[j]){
                adj[w1[j]-'a'].push_back(w2[j]-'a');
                inDegree[w2[j]-'a']++;   // Create connected graph.
                break;                   // => Put as integer
            }
        }
    }
    queue<int> q;
    for(int i=0; i<K; i++){
        if(inDegree[i]==0) q.push(i);
    }
    string res="";
    while(!q.empty()){
        int temp = q.front();
        char chr = temp+'a';
        q.pop();
        res = res + chr;

        for(auto x: adj[temp]){
            inDegree[x]--;
            if(inDegree[x]==0)
                q.push(x);
        }
    }
    return res;
}
```

#18) Flower Planting with No Adjacent

Coloring problem (Leet Code 1042)

I/P:- n = 3
paths = [[1, 2], [2, 3], [3, 1]]

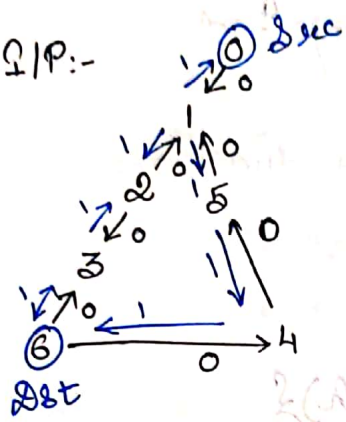
O/P:- [1, 2, 3].

Code

```
vector<int> gardenNoAdj(int n, vector<vector<int>>
                                & paths) {
    vector<int> adj[n+1];
    vector<int> ans(n);
    for(auto i: paths){
        adj[i[0]].push_back(i[1]);
        adj[i[1]].push_back(i[0]);
    }

    for(int i=1; i<=n; i++) {
        // Initially all clrs are available
        vector<int> color(5, 0);
        for(auto a: adj[i]){
            // Ans vect stores color of node which i paint
            // Visit adj vertices and mark clr as "1" ie,
               not available
            color[ans[a-1]] = 1;
        }

        for(int j=1; j<=4; j++){
            if(color[j-1] == 0) {   // If any clr is avail
                ans[i-1] = j;       color ith node with
                } break;            it
        }
    }
    return ans;
}
```

#19) Minimum Edges to Reverse to make path from src to destination

I/P:-



O/P:- 2

Algorithm

① Make a reverse edge corresponding to each edge.

② Give a wgt of 0 to original edge
Give a wgt of 1 to reverse edge
(as shown in i/p figure)

③ Now applying Dijkstra shortest path algo, we find no. of reversals req.

[If we move towards org edge 0 cost
Moving in reverse edge incurs cost of 1
(which signify reversals)]

#80) Covid Spread (Rotten Oranges)

0: Empty Ward
1: Uninfected Patient
2: Infected

I/P: 2 1 0 2 1
1 0 1 2 1
1 0 0 2 1

O/P:- 2 time units

① BFS in 4 directions.

Code

```
int covidSpread(vector<vector<int>> h){
    int n = h.size();
    int m = h[0].size();

    vector<vector<int>> infectionTime(n, vector<int>(m,
                                          it
    queue<pair<pair<int,int>,int>> q;

    for(int i=0; i<n; i++){
        for(int j=0; j<m; j++){
            if(hospital[i][j]==2){
                q.push({{i,j},0});
                infectionTime[i][j]=0;
            }
        }
    }

                // Traverse in 4 directions.
    vector<pair<int,int>> direction
        = {{1,0},{0,-1},{-1,0},{0,1}};

    while(!q.empty()){
        auto curr = q.front();
        q.pop();
```

```
        for(auto it : direction){
            int i = curr.first.first + it.first;
            int j = curr.first.second + it.second;

            // Search within bounds
            if(i<0 || j<0 || i>=n || j>=m || h[i][j] != 1
                || infectionTime[i][j] != -1)
                continue;

            q.push({{i, j}, curr.second+1});
            infectionTime[i][j] = curr.second+1;
        }
    }

    int maxTime=0; flag=0;

    for(int i=0; i<n; i++){
        for(int j=0; j<m; j++){
            if(hospital[i][j] != 1)
                continue;
            if(hospital[i][j]==1 && infectionTime==-1)
                return -1;          // all patients not infected
            maxTime = max(maxTime, infectionTime[i][j]);
        }
    }

    return maxTime;
}
```

#21) Check if Graph is Bipartite

Two Teams ?
Interview Bit

• Graph is 2 colorable.

```
int flag = 0;
bool isBipartite(vector<int> & adj){
    vector<int> clr(n, -1);
    flag = 0;
    for(int i=0; i<n; i++){
        if(color[i] == -1){
            dfs(i, 0, 1, adj, color);
        }
    }
    return (flag)? 0 : 1;
}

void dfs(int src, int par, int clr, adj, & color){
    color[src] = clr;
    for(auto it: adj[src]){
        if(color[it] == -1)
            dfs(it, src, 3 - clr, adj, color);
        else if(it != par && clr == color[it]){
            flag = 1
            break;
        }
    }
}
```

#22) Journey to the Moon (Hackerrank)

Each pair of nodes given is from same country. Determine total number of pairs from diff. countries you can choose from.

I/P:- [1,2] [2,3] [0]. O/P:- 3

(0,1); (0,2); (0,3).

① Total no. of pairs possible = $^nC_2$.

② pairs from 1 country = $^nC_2$.

∴ $^nC_2 - {}^nC_2 - {}^nC_2 \ldots\ldots$

Code

```
long long journeyToMoon(int n, vector<vector<int>> astronaut) {
    if(n == 0)
        return 0;
    vector<int> g[n];
    for(auto i: astronaut){
        g[i[0]].push_back(i[1]);
        g[i[1]].push_back(i[0]);
    }
    vector<long long> numberNodes; // no. of nodes in each connected comp.
    vector<bool> visited(n, false);
    for(int i=0; i<n; i++){
        if(!visited[i]){
            long long countNodes = 0;
            dfs(i, g, visited, countNodes);
            numberNodes.push_back(countNodes);
        }
    }
}
```

```
// nC2 total pairs
long long res = n;
res *= n-1;
res /= 2;

// Remove pairs from individual countries
for(int i=0; i<numberNodes.size(); i++){
    long long x = numberNodes[i];
    x *= (x-1);
    x /= 2;
    res = res - x;
}
long return res;
}
```

// Imp. to follow given steps to avoid overflow

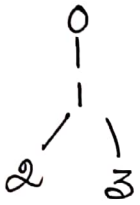
```
// DFS method to count no. of nodes in each connected component.
void dfs(int src, vector<int> g[], &visited,
                                long long &countNodes){
    visited[src] = 1;
    countNodes++;
    for(auto i: g[src]){
        if(!visited[i])
            dfs(i, g, visited, countNodes);
    }
}
```

#23) Find minimum height of a tree

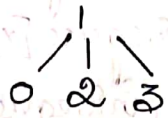
① If you find out height from ~~tt~~ nodes leaf nodes it will be higher. (leafnode : indegree = 1)

② Always try and pick nodes that are non leaf nodes.

I/P:- [[1,0], [1,2], [1,3]]



height = 2



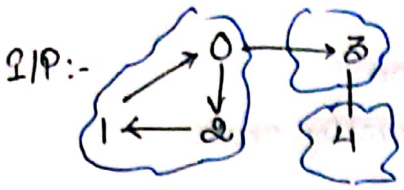
height = 1.

Code

```
vector<int> findMinHgt (int n, vector<vector<int>> & edges){
    if(n==1)
      return {0};
    vector<int> graph[n];
    vector<int> inDegree(n,0);
    for(auto i: edges){
       graph[i[0]].push_back(i[1]);
       graph[i[1]].push_back(i[0]);
       inDegree[i[0]]++;
       inDegree[i[1]]++;
    }
    queue<int> q;
    for(int i=0; i<n; i++){
       if(inDegree[i]==1)
          q.push(i);
    }
```

```
while(n>2){
    int size = q.size();
    n = n - size;
    for(int i=0; i<size; i++){
        int temp = q.front();
        q.pop();
        for(auto x: graph[temp]){
            inDegree[x]--;
            if(inDegree[x]==1)
                q.push(x);
        }
    }
}

vector<int> res;
while(!q.empty()){
    res.push_back(q.front());
    q.pop();
}
return res;
}
```

#24) Kosaraju algorithm to find strongly connected Components



O/P:- 3

Algo

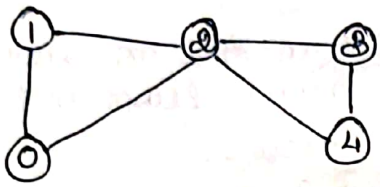
1. Input graph
2. Make copy of rev graph
3. Maintain a order vector for DFS on graph
4. Run DFS on reverse graph from end of order vector.

Code

```
vector<int> order;
int kosaraju(int V, vector<int> adj[V]){
    order.clear();
    vector<int> revadj[V];
    for(auto it int i=0; i<V; i++){          // Reverse graph.
        for(auto x: adj[i]){
            revAdj[x].pb(i);
    } }

    vector<bool> vis(V, false);
    for(int i=0; i<V; i++){
        if(!vis[i])
            dfs(i, vis, adj);
    }

    vector<bool> revis(V, false);
    int count = 0;
    for(int i= order.size()-1; i>=0; i--){
        if(!revis[order[i]]){
            dfs(order[i], revis, revadj);   // Pass to a simple dfs funct".
            count++;
        }
    } return count;
}
void dfs(int src, & vis, adj[]){
    vis[src] = true;
    for(auto x: adj[src])
        if(!vis[x])
            dfs(x, vis, adj);
}
    order.push_back(src);
```

#25) Euler's Path & Circuit

A trail that starts & ends at the same vertex covering every edge ~~at least~~ exactly once is a Euler's circuit



Euler Circuit

0 → 1 → 2 → 3 → 4 → 2 → 0

(Vertex can be repeated)

① There should be only 1 component
② Start & end vertex same
③ Each edge visited exactly once.

**
All vertices must have a even-degree to be a euler graph (a graph with euler circuit)

Even degree because (1 node coming in vertex
1 node going out "
to form cycle.

Semi-Eulerian Path

i) Every edge should be visited once
ii) Starting vertex ≠ End vertex.

Exactly 2 vertices must have ~~even~~ odd degree (start & end vertex)

All other vertices must have even degree.

#26) Snakes and Ladders

① BFS use

② Find the coordinates

row = n - ((curr - 1)/n) - 1;

col = (curr - 1)%n → if(even) → n-1-col
                    → else → col.

Code

```
int snakesAndLadders(vector<vector<int>>& board){
    int n = board.size();
    int steps = 0;
    queue<int> q;
    vector<vector<bool>> visited(n, vector<bool>(n, false));
    q.push(1);
    visited[n-1][0] = true;
    while(!q.empty()){
        int size = q.size();
        for(int i=0; i<size; i++){
            int x = q.front();
            q.pop();
            if(x == n*n) return steps;
            for(k=1; k<=6; k++){ // 6 faces of dice
                if(k+x > n*n)
                    break;
                vector<int> pos = findCoordinates(k+x, n)
                int r = pos[0]; int c = pos[1];
                if(visited[r][c]== true) continue;
                visited[r][c] = true;
                if(board[r][c]==-1) q.push(k+x);
                else
                    q.push(board[r][c])
            }
        }
        steps++;
    }
    return -1;
}
```

```
vector<int> findCoordinates (int curr, int n){
    int r = n - ((curr-1)/n) - 1;
    int c = (curr-1)%n;

    if(r%2 == n%2){
        return {r, n-1-c};
    }else{
        return {r, c}
    }
}
```

(27) # Walls and Gates (LeetCode)

Fill each empty room with dist. to nearest gate

- (-1) → a wall/obstacle. [If not given replace with -1].
- 0 → A gate [If not given replace].
- INT_MAX → An empty room [If not given replace].

Code

```
void wallsAndGates(vector<vector<int>> & rooms){
    for(int i=0; i<rooms.size(); i++){
        for(int j=0; j<rooms[i].size(); j++){
            if(rooms[i][j]==0)
                dfs(i, j, 0, rooms);
        }
    }
}

void dfs(int i, int j, int dist, & rooms){
    if(i<0 || i>=rooms.size() || j<0 ||
                j>=rooms[0].size)
        return;
    if(rooms[i][j] < dist || rooms[i][j]==-1)
        return;

    rooms[i][j] = dist;

    dfs(i+1, j, dist+1, rooms);
    dfs(i-1, j, dist+1, rooms);
    dfs(i, j+1, dist+1, rooms);
    dfs(i+1, j-1, dist+1, rooms);
}
```

#28) Word Search (LeetCode - 79)

Given a grid of chars; return true if the word exists

I/P:- | A | B | C | E |
| S | F | C | E |
| A | D | E | E |

word = "ABCCED" O/P:- True

① DFS Traversal with slight backtrack

Code

```
bool exist(vector<vector<char>>&board, string word){
    for(int i=0; i<n; i++){
        for(int j=0; j<m; j++){
            if(board[i][j] == word[0]){
                if(dfs(i, j, n, m, board, 0, word))
                    return true;
            }
        }
    }
    return false;
}

bool dfs(int i, int j, int n, int m, &board, int pos,
         string word){
    if(i<0 || j<0 || i>=n || j>=m
       || board[i][j] != word[pos])
        return false;
    if(pos == word.length() - 1)
        return true;
    char tmp = board[i][j];
    board[i][j] = '-';          // Prevent going in loop.
    bool found = dfs(i+1, j, n, m, board, pos+1, word)
              || dfs(i-1, j, ...) || dfs(i, j+1, ...)
              || dfs(i, j-1, ...);
    board[i][j] = tmp;
    return found;
}
```

#29) Evaluate Division (LeetCode - 399)

I/P:- Eqns = [["a","b"],["b","c"]]
Values = [2.0, 3.0]. ← represents a_i / b_i.
Queries = [["a","c"],["b","a"],["a","e"],["a","a"]
["x","x"]]

O/P:- [6.00, 0.50, -1.00, 1.00, -1.00]

① Implemented using BFS

Code

```
vector<double> calc(vector<vector<string>>& equations, vector<double>& values,
                    vector<vector<string>> &queries){
    int en = equations.size();
    int qn = queries.size();
    unordered_map<string, vector<pair<string, double>>> hashmap;
    for(int i=0; i<en; i++){
        hashmap[equations[i][0]].push_back
                {equations[i][1], values[i]};
        hashmap[equations[i][1]].push_back
                {equations[i][0], 1.0/values[i]};
    }
    vector<double> res;
    // For each query.
    for(int i=0; i<qn; i++){
        string src = queries[i][0];
        string des = queries[i][1];
        // src node is not found
        if(hashmap.find(src) == hashmap.end()){
            res.push_back(-1.0);
            continue;
        }
```

```
queue<pair<string, double>> q;
q.push({src, 1.0});
int flag = 0;
unordered_set<string> visited;
visited.insert(src);

while(!q.empty()){
    auto temp = q.front();
    q.pop();

    string node = temp.first;
    double value = temp.second;

    if(node == des){ // Reached destination node.
        res.push_back(value);
        flag = 1;
        break;
    }

    for(auto p: hashmap[node]){ // Do only iff not visited
        if(visited.find(p.first) == visited.end()){
            q.push({p.first, val * p.second});
            visited.insert(p.first);
        }
    }
}
if(!flag)
    res.push_back(-1.0);
}
return res;
}
```

#30) Swim in Rising Water (LeetCode) 778

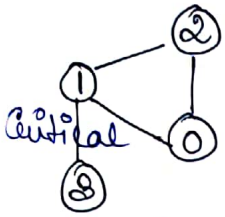
I/P:- $\begin{bmatrix} 0 & 2 \\ 1 & 3 \end{bmatrix}$ O/P:- 3.

Use → Binary Search + Graph(DFS)

Code

```
int swimInWater(vector<vector<int>>& grid){
    int n= grid.size(); int m=grid[0].size();
    int l=0, r= n*m;   // Define timeline
    while(l<r){
        int mid = l+(r-l)/2;
        vector<vector<int>> visited (n, vector<int>(m,0));
        dfsisOK(0,0,n,m,mid,grid,visited)
        if(visited[n-1][m-1])   // If entire matrix can be travelled within "mid" time it can travelled for t>mid.
            r=mid;
        else
            l=mid+1;
    }
    return r;
}
// Simple DFS function.
void dfsisOK(int i, int j, int n, int m, int mid, &grid, &visited){
    if(i<0 || j<0 || i>=n || j>=m){
        return;
    if(visited[i][j] || grid[i][j]>mid)   // If need to minimise time to search in lower bound time interval.
        return;
    visited[i][j]=1;
    dfsisOK(i-1,j,n,m,mid,grid,visited);
    dfsisOK(i+1,j,n,m,mid,grid,visited);
    dfsisOK(i,j-1,n,m,mid,grid,visited);
    dfsisOK(i,j+1,n,m,mid,grid,visited);
}
```

#) Critical Connections in a Network (LeetCode - 1192)



O/P:- {1, 3}.

Condition to be a Bridge
low[v] > disc[u] (i.e, there is no back edge)

Code

```
vector<vector<int>> criticalConnections(int, v<v<int>> & connect){
    vector<int> adj[n];
    //Make adj list with 0 based indexing.
    vector<int> disc(n,-1), low(n,-1), parent(n,-1);
    vector<vector<int>> res;
    for(int i=0; i<n; i++){
        if(disc[i] == -1)
            dfs(i, adj, disc, low, parent, res);
    }
    return res;
}

//Sanjan's Algorithm
void dfs(int i, adj[], & disc, & low, & parent, & res){
    static int time = 0;
    disc[u] = low[u] = time;
    time++;
    for(auto v: adj[u]){
        if(disc[v] == -1){
            parent[v] = u;
            dfs(v, adj, disc, low, parent, res);
            low[u] = min(low[u], low[v])
            if(low[v] > disc[u])
                res.pb({u, v});
        }
        else if(parent[u] != v){
            low[u] = min(low[u], disc[v]);
        }
    }
}
```