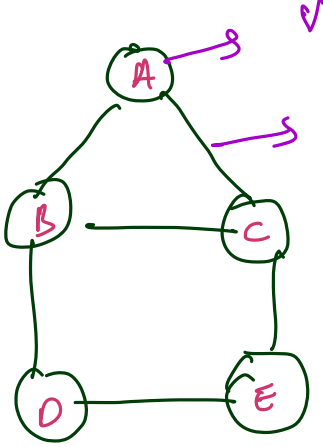
Graphs - 1

Non-Linear

Graphs : A graph consists a finite set of Vertices (Nodes) and edges



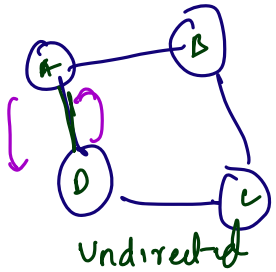
Vertices / Nodes

Which connects any 2 vertices

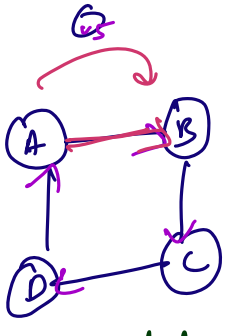
Properties

1) Non - Linear Data Structure
2) Non - Hierarchical D.S [No parent - child relationships]
3) Network like DS : Used to represent network.

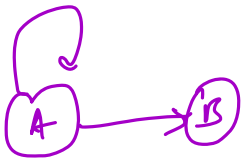
4) Edges
   Undirected & directed



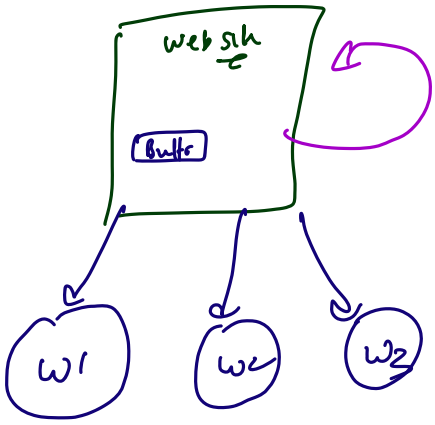
Undirected



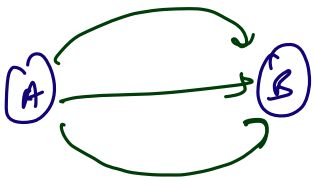
Directed

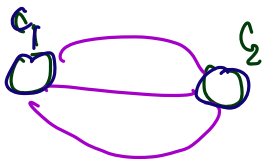


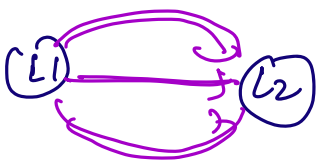
Self loop



Multi - Edge (Parallel)







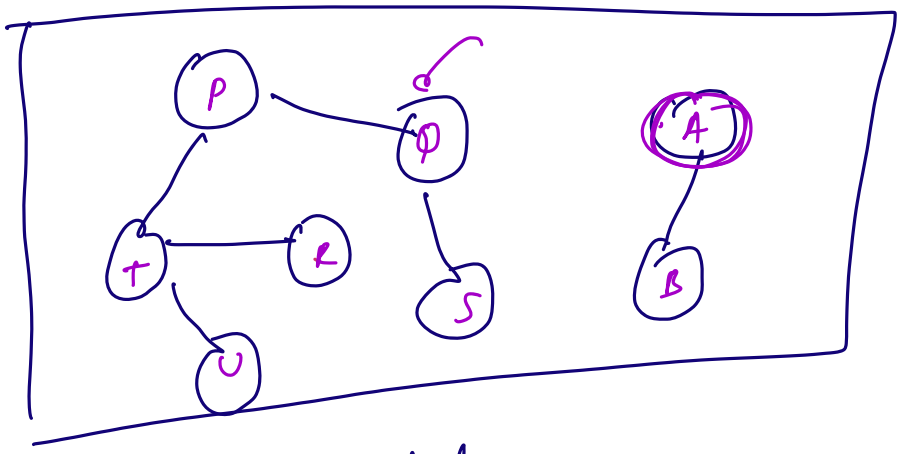
Types of Graphs

1) Simple Graph
   No self loops & parallel edges

2) Multi- Graph:
   A graph loops & parallel edges

3) Undirected / Directed Graph

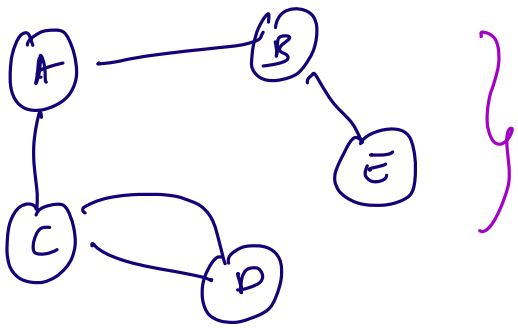
4) Connected & Disconnected graph
   [Undirected graphs]
   If I can reach any node from any other node, it is connected



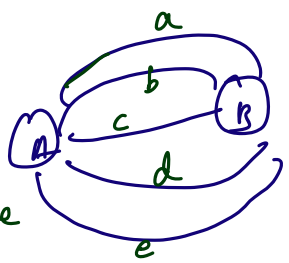
A, Q

Undirected

5) Unweighted / weighted Graph



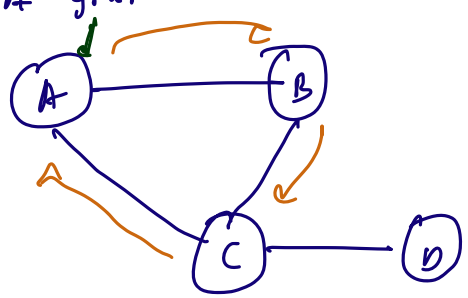
Unweighted



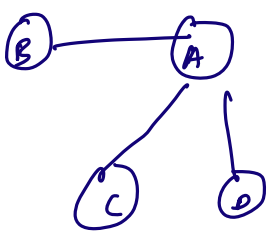
Weight : Cost of moving from one vertex to another

6) Cyclic & Acyclic graphy

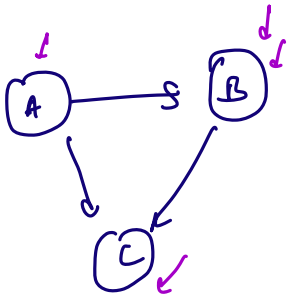
A graph which contains atleast 1 cycle



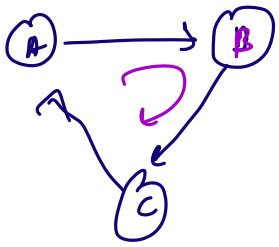
Cyclic Graph



Acyclic Graph



A cyclic

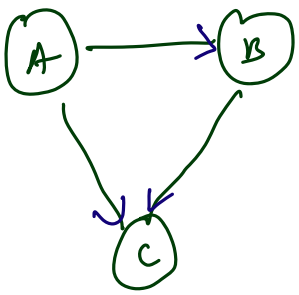


Cyclic

7) Strongly connected graph [Directed Graph]

If there exists a path between every 2 vertices, then it is called SCG

(u, v)



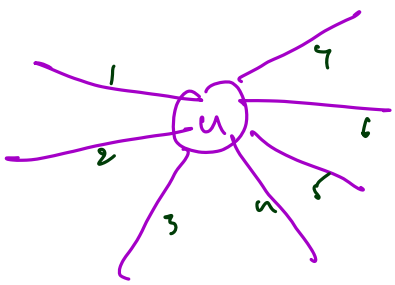
A → C ✓
B → C ✓
C → B ✗
C → A ✗

Not strongly connected

Degree of Vertex

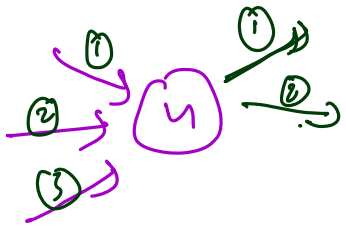
Undirected

No. of edges associated with any Vertex



deg(u) = ?

Directed Graph



indegree(u) = 3

outdegree(u) = 2

Real-life Applications

1) Google Maps (Flight Networks)

Vertices: places / landmark

Edges: roads

2) Social Networking

Vertices: Users / People

Edge: Connection

Facebook: (Friends) → Undirected Graph

A → B ⟺ B → A

Instagram: Directed Graph

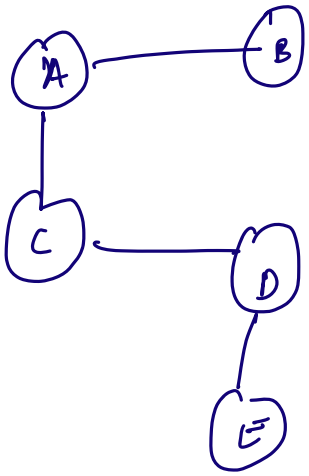
A follows B,

3) Resolving Dependencies:

A → B

Help us with the order of courses I need to take [Topological sort]

Representation of Graph



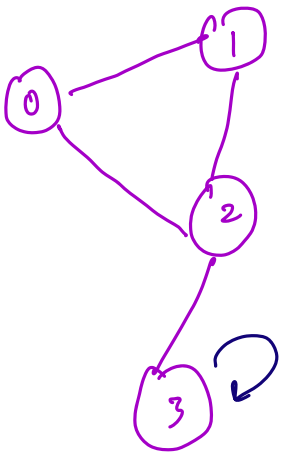
1) Adjacency Matrix

$V = 4$

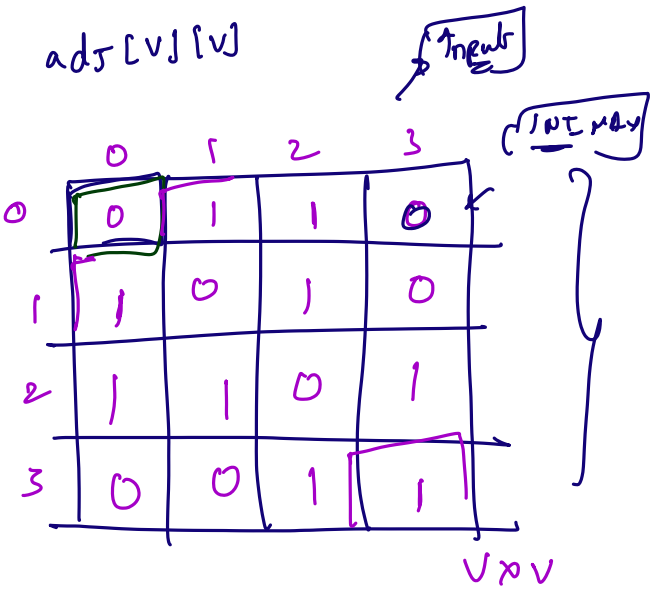
adj[V][V]

adj[0][1] = 1
adj[1][0] = 1

(i, j)



Adj[i] =



Adj[i][j] = 1 if there is an edge from i → j

= 0 if there is no edge

What if edges are weighted?

Instead of 0/1, store the weights of edges

Graph is directed? u → v

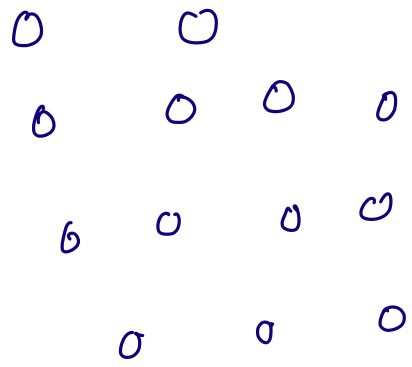
adj[u][v] = 1

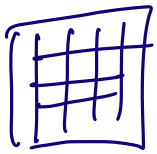
Cons

→ Space : $O(V^2)$

→ Can we represent/store parallel edges?

Sparse





(1000) x 1000

1000 vertices

. [Hashmap]

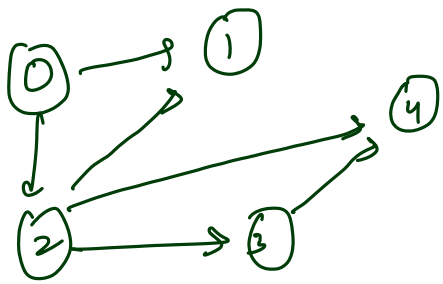
2) Adjacency List :

Array of vectors

vectors of vectors

V = 5

Adj List



Directed

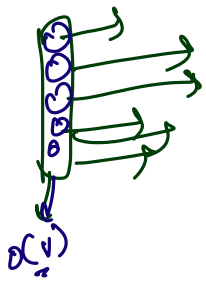
0 : → { 1, 2 }

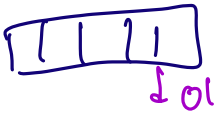
1 : { }

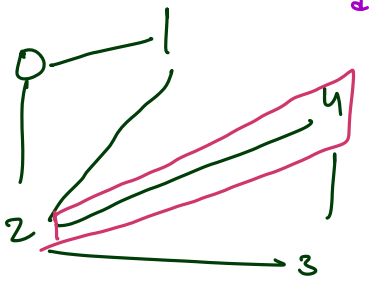
2 : { 1, 3, 4 }

3 : { 4 }

4 : { }

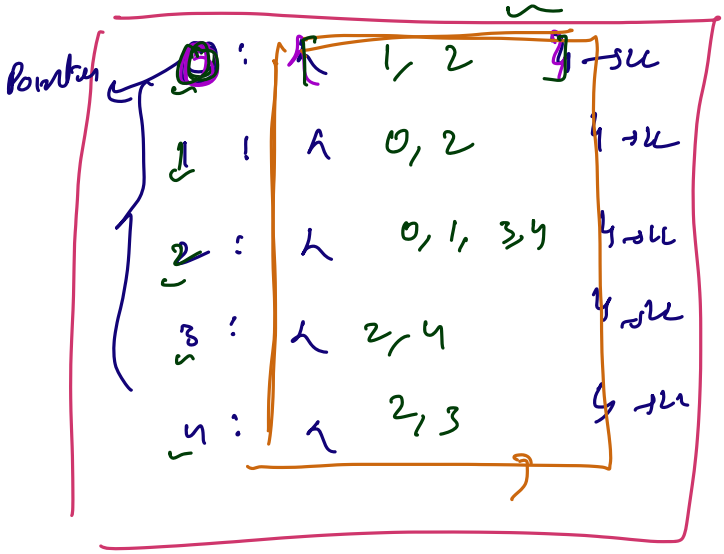






Un Directed graph

Linked List → vector of vectors



Space Complexity :

vector(u).push(v)
vector(v).push(u)

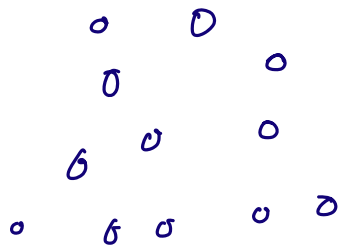
O(V+E)

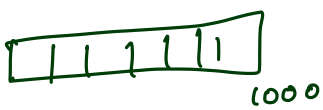
Edges

(2.E + V)

⇒ O(V+E)

O(V + V²) ⇒ O(V²)





1000 vertices, 0 edges

Matrix: 10^6

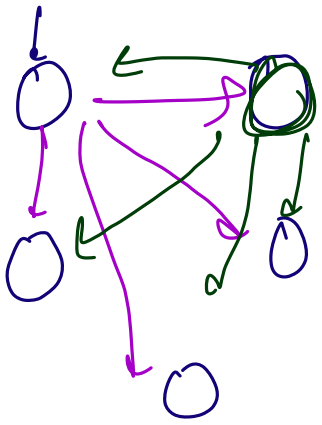
Cost : 10^3

Worst Case :

- Vertices, No of edges ?

Directed Graph

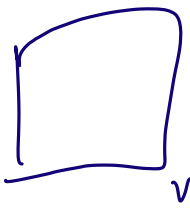
V vertices



$(V-1) + (V-1)$

V vertices

$= V(V-1)$ edges



V × V

→ If the graph is sparse (No. of edges is of the order O(V) → Adjacency list is better

→ In dense graphs, both of the occupy the space.

Edges

V → No of vertices = 5

E → No. of edges = 10

Edges :

1	2
2	4
3	1
4	5
1	3
4	2
3	2

(u, v)

u → v

Matrix :

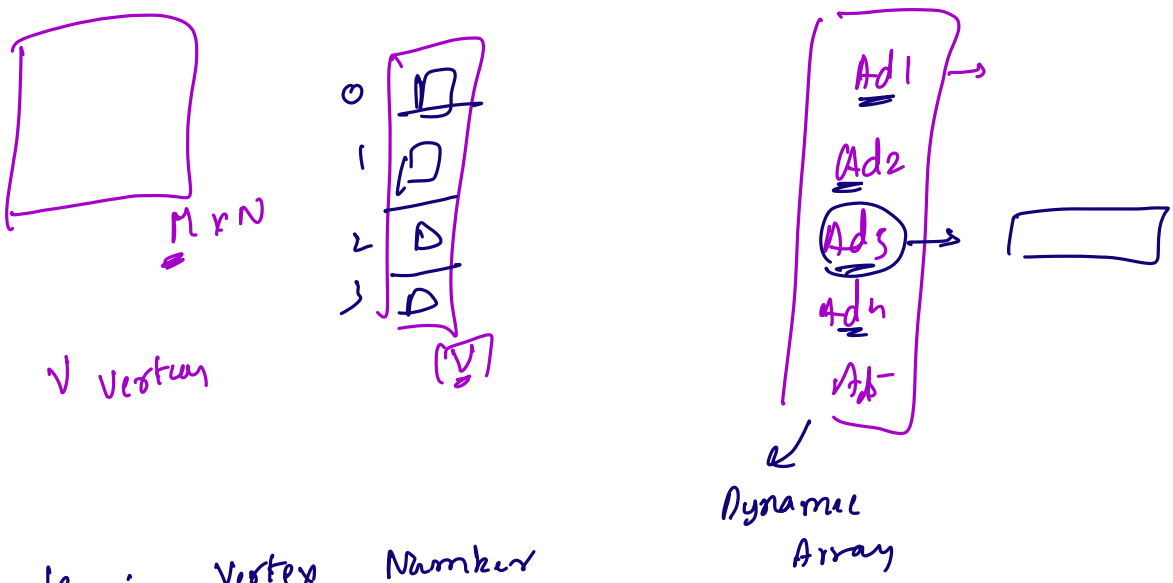
```
for (i=0; i < E; i++) {
    u = edges[i][0];
    v = edges[i][1];
    mat[u][v] = 1
}
```

Adjacency List :

// Undirected

u — v

```
vector <vector<int>> adj;
for (i=0; i< Edges.size(); i++) {
    u = Edges[i][0];
    v = Edges[i][1];
    adj[u].push(v);
    adj[v].push(u);
}
```



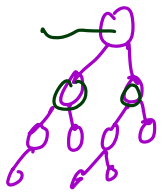
Key : Vertex Number

Value : Vector (Dynamic Array)

Traversal:

Visiting all the vertices and edges exactly one

The order of visiting the vertices is different in different algorithms

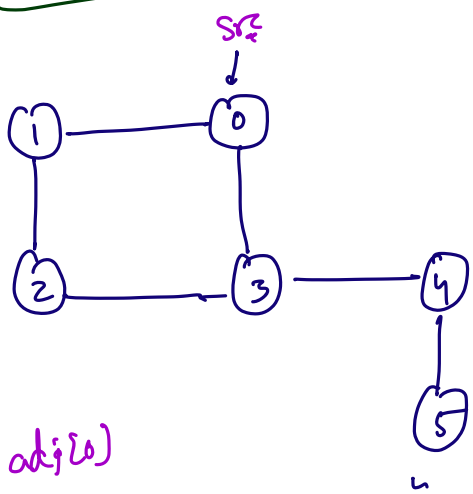


Trees

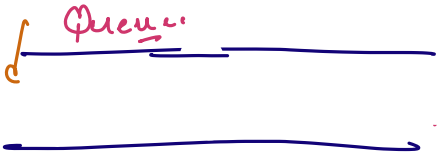
(DFS) ← Preorder, Postorder, Inorder : Recursion (Stack)

(BFS) ← Level Order Traversal : Queue DS

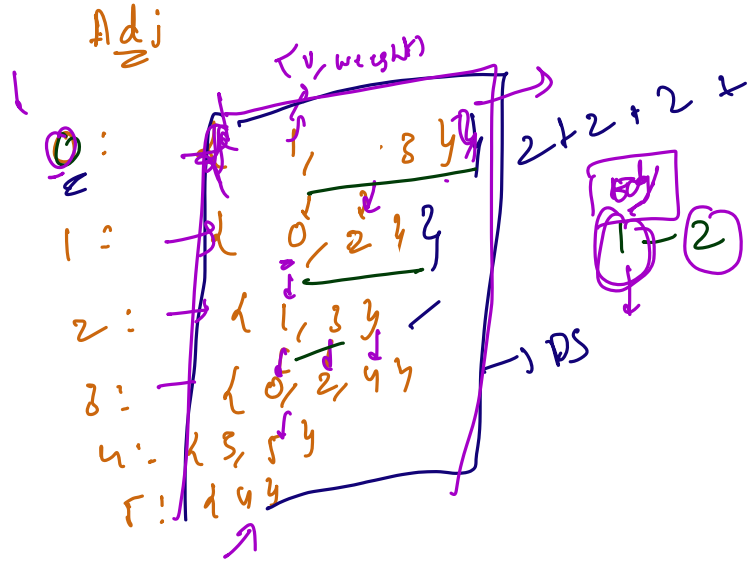
Breadth First Search Traversal



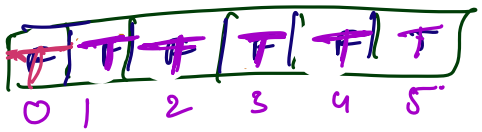
adj[0]



Output: 0 1 3 2 4 5



Visited :



```
// Generate the adjacency list
vector<int> adj[N];

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

void bfs(int src){
      bool visited[N];
      queue<int> q;
      q.push(src);
      visited[src] = true;
      while( !q.empty()){
          temp = q.front();
          cout << temp;
          q.pop();
          for(auto i : adj[temp]){
              if(!visited[i]){
                  visited[i] = 1;
                  q.push(i);
              }
          }
      }
return;
}
```

V = 5 → {0, 1, 2, 3, 4}

adj[0]
[1]
[2]
[3]
[4]

Diff

Random vertex

= {false}

adj List → 2 Push Operations

BFS

Iterate over adj list

Iterator

```
// Generate the adjacency list
vector<int> adj[N];

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

void bfs(int src){
     bool visited[N];
     queue<int> q;
     q.push(src);

     while( !q.empty()){
          temp = q.front();
          cout << temp;
          q.pop();
          for(auto i : adj[temp]){
               if(!visited[i]){
                    visited[i] = 1;
                    q.push(i);
               }
          }
     }
return;
}
```

√ time

→ Constant

Neighbours of v_i

√

v_1 v_2 . . . . . v_v

$C + N_1$ $C + N_2$ . . . $C + N_n$

$N_1 + N_2 + N_3 + N_4 + \ldots N_n$

Backward

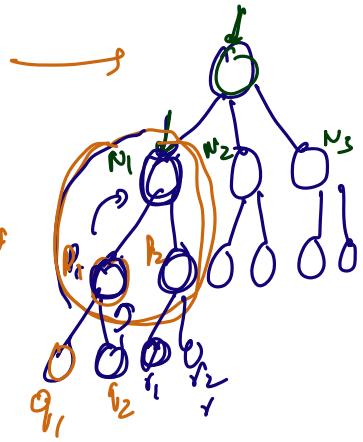
E or $2E$ → undirected graph

$\Rightarrow C \cdot V +$ (Constant)

$C \cdot V + 2E \Rightarrow O(V+E)$

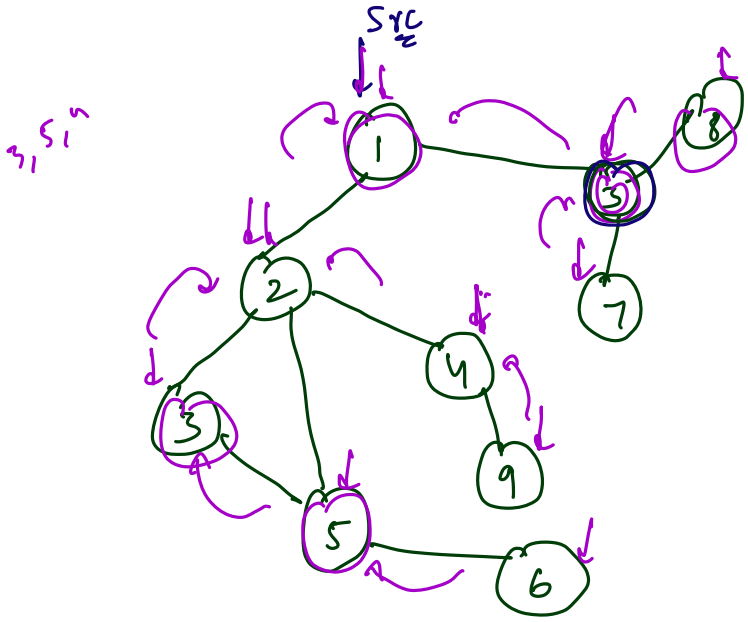
Depth First Search (DFS)

Exploring as far as possible along each branch before we back track.

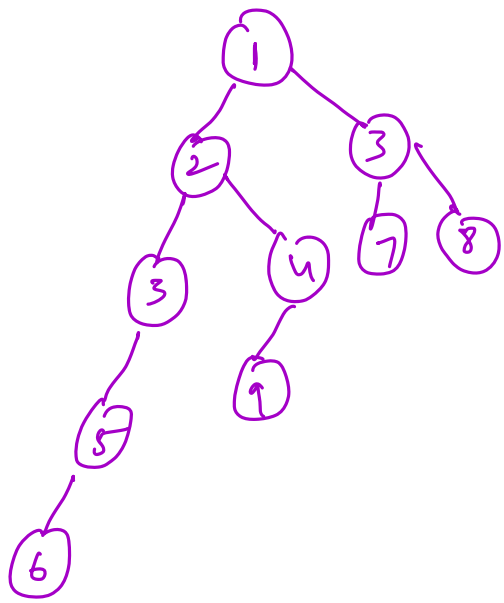


visited[] - fn

Recursion



Graph

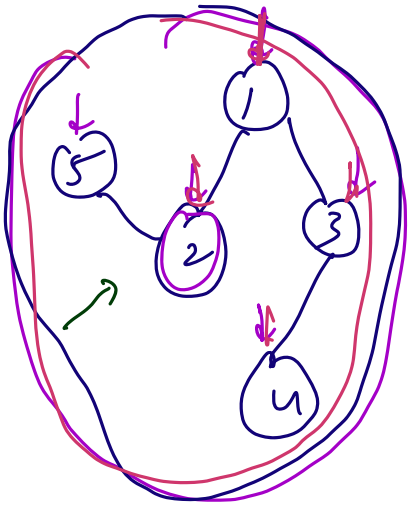


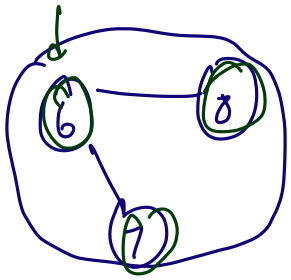
DFS Tree

Output: 1 2 3 5 6 4 9 3 7 8

```
void dfs(int src) {
    visited[src] = true;
    print(src);
    for(i in adj[src]) {
        if(!visited[i]);
            dfs(i);
    }
}
```

→ DFS(random node)



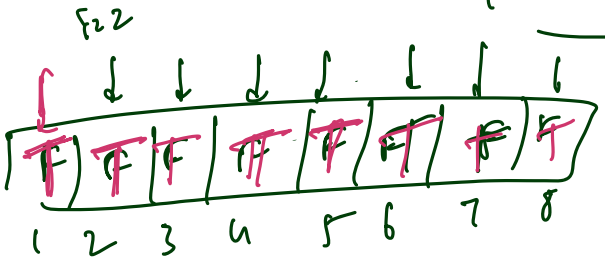


DFS(1)

Disconnected Graph

```
for(i=0; i<V; i++) {
    if( visited[i] == false ) {
        dfs(i);
        count++;
    }
}
```

DFS

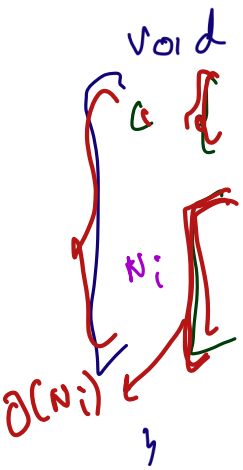


T.C: O(V+E)

```
void dfs( int src) {
    visited[src] = true;
    print( src);
    for( i in adj[src] ) {
        if( !visited[i] )
            dfs(i);
    }
}
```

over all my neighbours

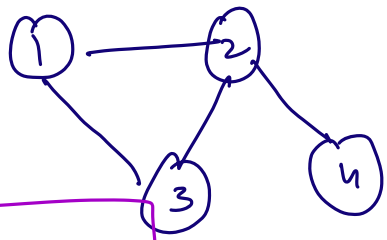
→ ith one



How many calls to DFS function?

V calls

O(V+E)

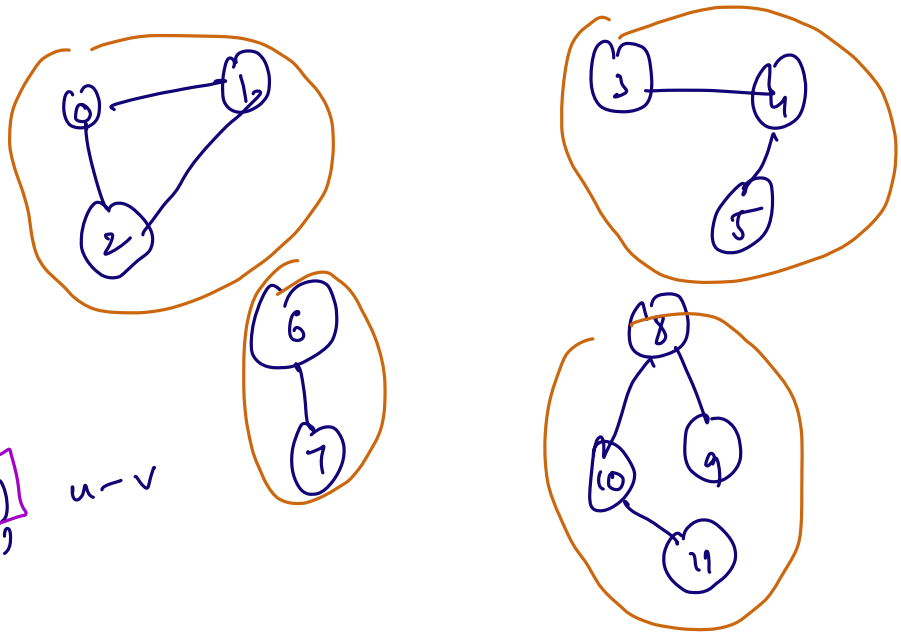


V = 4

C + N1 + C + N2 + C + N3 + . . .

O(V+E)

Question: No. of Connected components



adj[u].push(v)
adj[v].push(u)

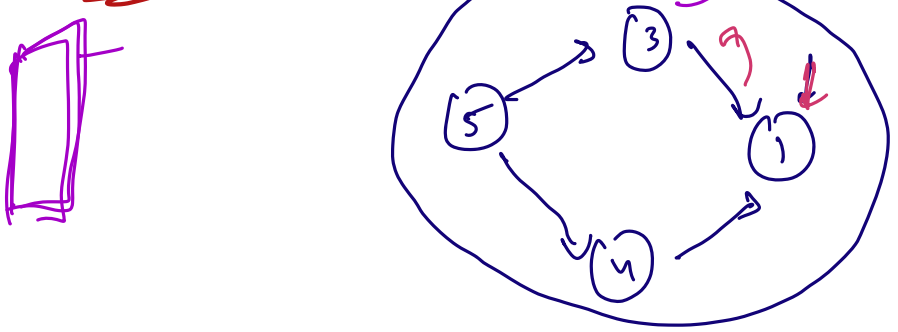
u - v

Ans = 4

Disconnected graph

0	1	2	3	4	5	6	7	8	9	10	11
T	T	T	T	T	T	T	T	T	T	T	T

Question: Path in a Directed Graph.



(5, 1) ✓ True

(3, 1) ✓ False

(4, 1) True

(u, v)
u = src
v = dest.

→ Start DFS / BFS and return true if we reach the destination

T.C: O(V+E)

S.C:

BFS / DFS

E : V

1 → 2 → 3 → 4 → 5

u = 1 v = 5

BFS

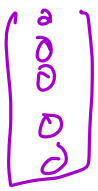
1) Visited Array O(V)
2) Queue O(V)
3) Adj list: O(V+E)

All pair shortest : $O(V^2)$

O(V+E)
No. of vertex

DFS

1) Visited Array O(V)
2) Stack space : O(V)
3) Adj Lst: O(V+E)

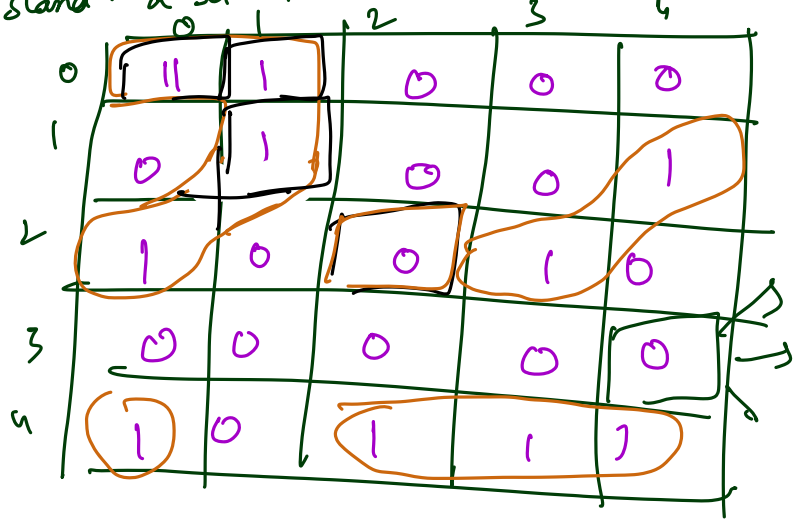


Question: No. of Islands

Given a matrix of 0's [water] and 1's [Land]

Island: a set of connected 1's

cell [i] [j]

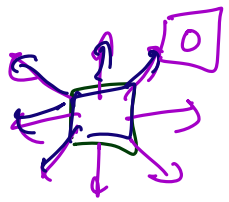


4 Islands

Out of bound

Vertex : Each cell with value 1

Edge : All 8 neighbours which are 1's



Cell (i,j) is a vertex if A[i][j] == 1

```
for(i=0; i<n; i++) {
    for(j=0; j<m; j++) {
        if( A[i][j] == 1 && !visited[i][j] ){
            DFS(i, j);
            count;
        }
    }
}
```

---

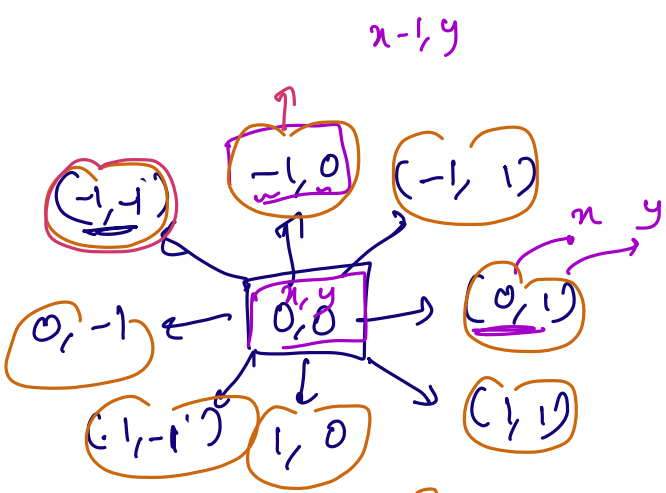
8 if-else conditions

```
if
if
if
.
.
.
```

(x-1, y+0)

x, y

(x-1, y-1)

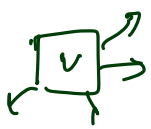



```
rows = [-1, -1, -1, 0, 1, 1, 1, 0]
cols = [-1, 0, 1, 1, 1, 0, -1, -1]
```

8

```
for(i=0; i<8; i++) {
    u = x + rows[i];
    v = y + cols[i];
}
```

```
rows = {
cols =
void dfs(int x, int y) {
    visited[x][y] = true;
    for(i=0; i<8; i++) {
        u = x + rows[i];
        v = y + cols[i];
        if(u>=0 && v>=0 && u<r && v<c && !visited[u][v] && A[u][v]==1)
            DFS(u,v)
    }
}
```

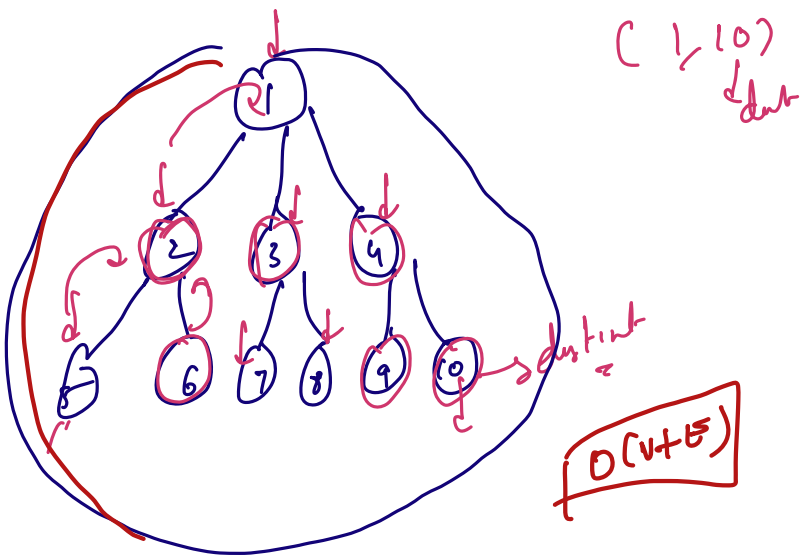


T.C: $O(V+E)$

$V \times C$ $\quad$ $V \times C \times 8$

T.C: $O(r \times c)$

S.C: $O(r \times c)$



DFS

```
dfs(int src){
      visited[src] = 1;
      cout << src;
      for(auto i : adj[src]){
            if(!visited[i])
                  dfs(i);
      }
}

for(i = 0; i < n; i++){
      if(!visited[i])
            dfs(i);
}
```

BFS

```
// Generate the adjacency list
vector<int> adj[N];

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

void bfs(int src){
      bool visited[N];
      queue<int> q;
      q.push(src);
      visited[src] = true;

      while( !q.empty()){
            temp = q.front();
            cout << temp;
            q.pop();
            for(auto i : adj[temp]){
                  if(!visited[i]){
                        visited[i] = 1;
                        q.push(i);
                  }
            }
      }
return;
}
```

No. of islands

```
void dfs(int i, int j){
      visited[i][j] = 1;
      for(int p = 0; p < 8; p++){
            x = i + rows[p];
            y = j + cols[p];
            if(x >= 0 && y >= 0 && x < r && y < c && A[x][y] == 1 &&
!visited[x][y])
            dfs(x, y);
      }
}

int count = 0;
for(int i = 0; i < r; i++){
      for(int j = 0; j < c; j++){
            if(A[i][j] == 1 && visited[i][j] == 0){
                  dfs(i, j);
                  count++;
            }
      }
}
```

Path in a Directed graph

```
int v; // target node global variable
int vst[N] ={0}; //array to store if node is visited or not
bool dfs(int s){
      if(s==v){
            return true;
      }
      vst[s]=1;//marking node visited
      for(int v:adj[s]){
            if(!vst[v])//if node not visited , go to that node
                  if(dfs(v))//If we found target node return
true
                        return true;
      }
      return false; //if target node not found return false
}

int main(){
      dfs(u);
}
```