GRAPH NOTES

Topics Covered :-

1) Adjacency Matrix & List
2) BFS / DFS (Overview)
3) Detect Cycle in Undirected Graph
   ↳ Using → BFS
   ↳ DFS

4) Bipartite Graph (Overview)
   ↳ Using → BFS
   ↳ DFS

5) Shortest path in Undirected Graph with Unit weights.
6) Shortest path in Directed Acyclic Graph (DAG)
7) Minimum Spanning Tree (MST)
   (i) Prim's Algorithm
   (ii) Kruskal's Algorithm [Disjoint Data Structure]

~~8) Kruskal's Algorithm~~

8) Dijkstra Algorithm
9) Detecting cycle in a Directed Graph
   ↳ Using → DFS
   ↳ BFS [Kahn's Algorithm]

10) Topological Sorting
11) Kosaraju's Algorithm for Strongly Connected Components.
12) Bellman Ford Algorithm for detecting negative cycle in a Graph.

GRAPH

Adjacency Matrix – Space $O(n^2)$

```
int main()
{   int n, e;                    // no. of vertices(n) & edges(e)
    cin >> n >> e

    int adj[n+1][n+1];  //adjacency matrix

    for(int i=0; i<e; i++)
    {
        int u,v;                 //vertices
        cin>>u>>v;
        adj[u][v]=1;             } -> Undirected graph
        adj[v][u]=1;
    }
    return 0;
}
```

Adjacency List – Space $O(n+2e)$

(n: no. of nodes; $2e$: edges b/w nodes.)

```
int main()
{   int n, e;
    cin>>n>>e;

    vector<int> adj[n+1]; // adjacency list.
    for(int i=0; i<e; i++)
    {   int u,v;
        cin>> u >> v;

        adj[u].push_back(v);
        adj[v].push_back(u);          ] Undirected graph.
    }
```

O(n+E)

BFS → Use queue. Time/space → O(n)

DFS → Use stack [if order at each level does not matter]

Time/space → O(n) → Use recursive approach.

↳ O(n+E)

Detect a cycle in Undirected Graph using BFS -

* Iterate through all the vertices of graph.
* Check for all vertices does they have a cycle
  ↳ If any of the vertex has a cycle then the entire graph will be called as cyclic.

↳ Here we will maintain a previous node with each current node

↳ If the upcoming adjacent node in the iteration is not visited yet then push it onto queue with it's prev. node and mark it as visited.

↳ If the adjacent node in the iteration is already visited and it is equal to the previous of current node then no issue move forward.

↳ But if it is not equal to the previous of current node, then it mean that

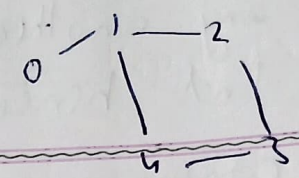
any another node has already visited it And, hence we can say that the graph contains cycle.

~~for~~ Time – O(n)
Space – O(n)

```
bool isCycle(int V, vector<int> adj[])
{   vector<int> visited(V, 0);
    for(int i=0; i<V; i++)
    {   if (isCycleBFS(i, visited, adj))
            return true;
    }
    return false;
}
```

```
bool isCycleBFS(int start, vector<int>& visited, vec<int> adj[])
{
    queue<pair<int,int>> q;        curnode → , prev node
    q.push({start, -1});
    visited[start] = 1;

    while(!q.empty()).
    {   auto it = q.front();   q.pop();
        int curnode = it.first;
        int prevnode = it.second;
        for(auto adjacentnode : adj[curnode])
        {
            if(visited[adjacentnode] == 0)
            {   q.push({adjacentnode, curnode});
                visited[adjacentnode] = 1;
            }
            else if(adjacentnode != prevnode)
                return true;
    }}
    return false;   }
```

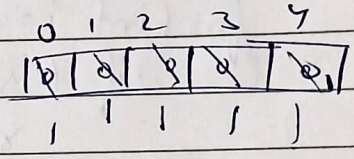
Detect Cycle in undirected graph using DFS :-



* Iterate through all the vertices of graph
- Check for cycle
  - Maintain a previous node
  - Recursively call cycleDfs.
  - If at any point if the adjacent node of the current node is already visited and if it is not the prev node of curr. node then we can say that cycle is present.

cur	prev
0	-1
1	0
2	1
3	2
4	3

0 — 1
1 — 0 2 4
2 — 1 3
3 — 2 4
4 — 1 3



Time - O(n)
Space - O(n)

```
bool cycleDFS (int cur, int prev, vector<int>& visited,
                                  vector<int> adj[])
{
  visited[cur] = 1;
  for(auto adjacent: cur)
  {
    if (visited[adjacent] == 0)
      return cycleDFS(adjacent, cur, visited, adj);
    else if (adjacent != prev)
      return true;
  }
  return false;
}
bool isCycle (int V, vector<int> adj[])
{ vector<int> visited (V, 0);
  for(int i=0; i<V; i++)
  {
    if (visited[i] == 0)
      if (cycleDFS (i, -1, visited, adj)) return true;
  }
  return false;
}
```

cur node, prev node initially

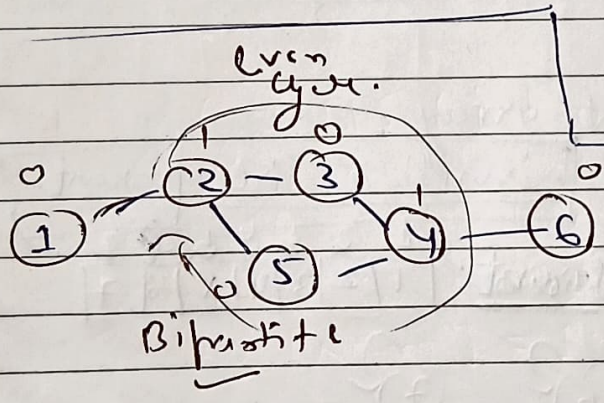
Bipartite Graph (BFS) / DFS —

- A graph that can be coloured using 2 colors such that no two adjacent nodes have same color.

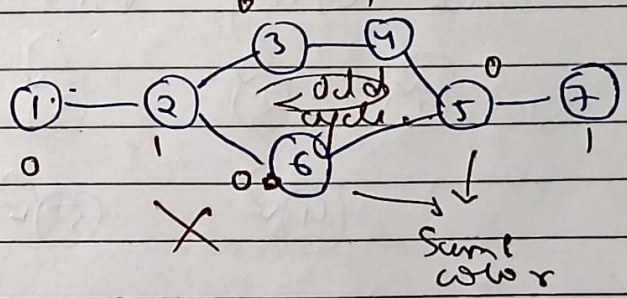
↳ If the graph have an odd length cycle it is not Bipartite
Other-wise it is Bipartite

Steps (by BFS) -

- Apply BFS algorithm & maintain a color array.
- If at any step the color of current node comes out to be same as its adjacent node then the graph will not be a bipartite.



Check for this, for each node of the graph.



Shortest path in Undirected Graph with unit weights.

- Assume all the edges are weighted 1.
- Here we need to find the shortest distance ~~bet~~ from source node (let's take "0") to all other nodes.

- What we'll do is, first we will make a vector of size equal to length of nodes, in which we will store the ~~dist~~ minimum distance from the source node to that node.

- Minimum distance of the source node will be zero

- Maintain a queue and push the source node onto it initially.

- while the queue is not empty : — [BFS]
  - ↳ Pop from queue and store the front element as the current node.
  - ↳ Traverse for all the adjacent nodes of current node.
  - ↳ We know that to reach an adjacent node from current node we have to travel distance = 1. And for travelling till that current node, that distance would be stored in the distance vector [i.e... distance (cur)]

So, [total distance] to reach the adjacent node would be [1 + distance (cur)]

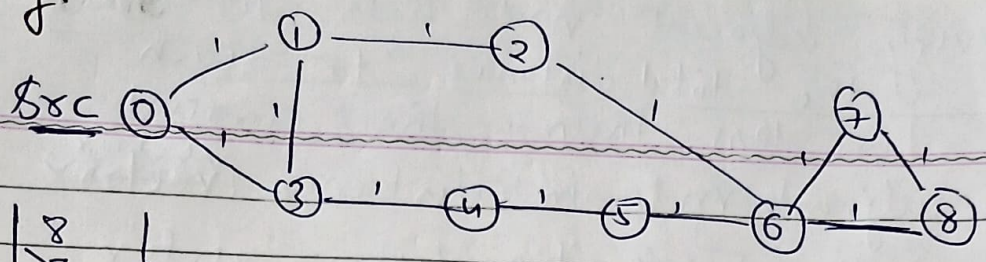
↳ So for every adjacent node we will check that if the total distance to reach that node is less than the current distance of adjacent node in distance vector.

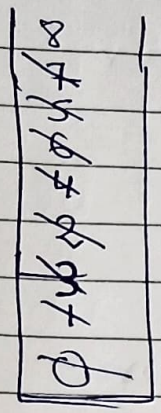
↳ Then we will update the distance vector of adjacent node index to the lower total distance and will push the adjacent node onto the queue.

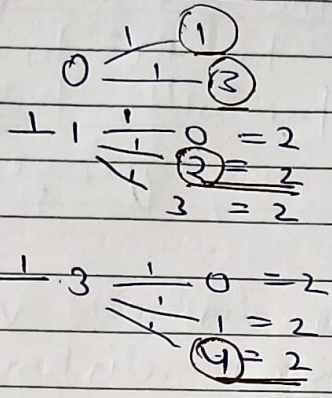
- At end we will get a distance vector in which the shortest distance of all the nodes from the source node will be stored.

```
void ShortestPath (vector<int> adj[], int N, int src)
{  vector<int> distance(N, INT_MAX);  // dist. vector
   distance[src] = 0;   // min dist. of src node = 0

   queue<int> q;
   q.push(src);   // Initially src. node pushed onto queue

   while (!q.empty())   // while queue not empty
   {  int current = q.front();
      q.pop();
      for(auto adjacent: adj[current])  // traverse through all adjacent nodes of current node.
      {
         if( distance[current] + 1 < distance[adjacent])
         {  distance[adjacent] = dist[current] + 1;
            q.push(adjacent);
         }
      }
   }
   for(int x: distance)
   {  cout << x << "\t";  }
}
```

eg:



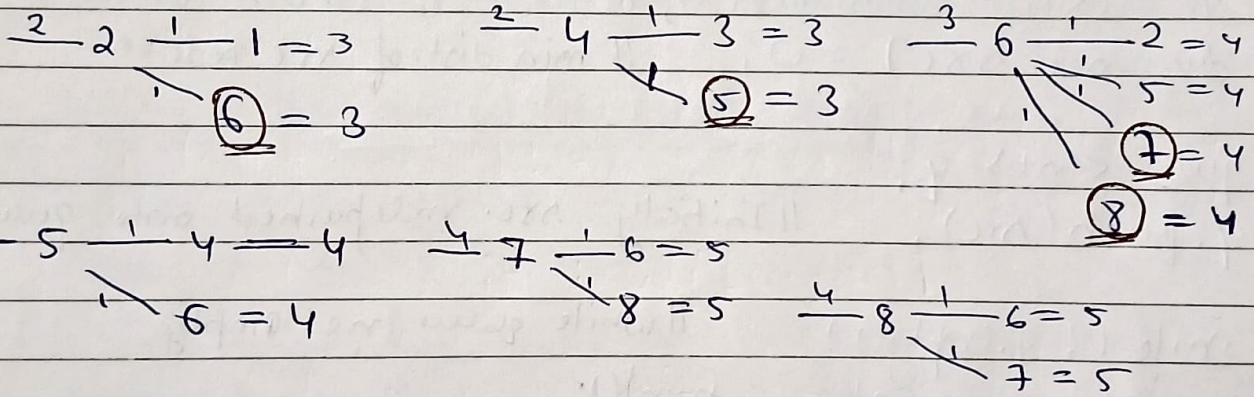




0 → 1, 3
1 → 0, 2, 3
2 → 1, 6
3 → 0, 1, 4
4 → 3, 5
5 → 4, 6
6 → 2, 5, 7, 8
7 → 6, 8
8 → 6, 7

src

0	1	2	3	4	5	6	7	8
0	1	2	1	2	3	3	4	4

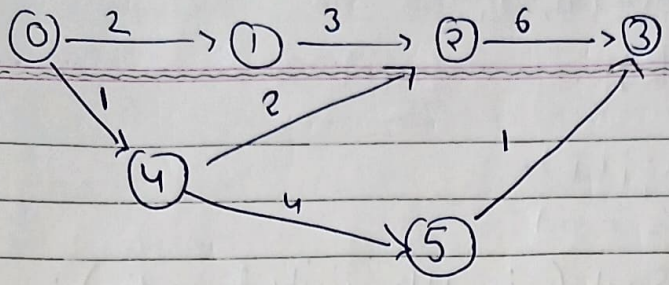


∴ Shortest distance from source (0) :—

Distance :

0	1	2	3	4	5	6	7	8
0	1	2	1	2	3	3	4	4

Shortest path in Directed Acyclic Graph (DAG)



- Generate the topo sort for the given graph
  - Initialize a distance vector of size equal to the no. of nodes present in the graph.
  - Initialize the shortest distance for all the nodes to 'INT_MAX' and for the source node it will be '0'

For the stack returned from the topo sort - do,

- While stack is not empty -
  - Pop from stack and store the item as the current node
  - Traverse through all adjacent nodes of current node.
  - ~~If the total~~ For each adjacent node check, If the total distance covered to reach the adjacent node [dist[current] + dist[cur->adj]] is less than the present distance of that adjacent node in the distance vector.

Check, if(dist[cur] !=

Then update the distance vector for that adjacent node.

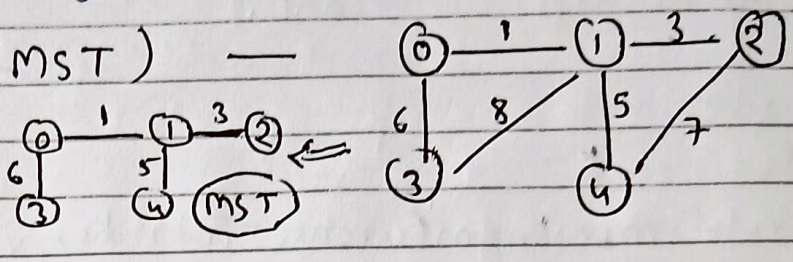
- At end print the distance vector.

```
void shortestPath(int src, int n, vector<pair<int,int>> adj[])
{
    vector<int> visited(n,0);
    stack<int> stk;

    for(int i=0; i<n; i++)
    {   if(visited[i]==0)
            findTopoSort(i, stk, visited, adj);
    }

    // Recieved stack
    // Initialize distance vector.
    vector<int> dist(n, INT_MAX);
    dist[src] = 0;   // min^n dist. of src node = 0

    while(!stk.empty())
    {
        int current = stk.top();
        stk.pop();

        if(dist[current] != INT_MAX) // Cond^n checked w.r.t
        {                            unreachable vertices.
            for(auto it : adj[current])
            {
                if(dist[current] + it.second < dist[it.first])
                {   dist[it.first]
                    = dist[current] + it.second; } // update
            }
        }
    }

// Print distance vector -
for(int i=0; i<n; i++)
{   cout.
(dist[i]==INT_MAX) ? cout << "Unreachable It" : cout <<
                                                dist[i];
```

Minimum Spanning Tree –

- A tree having 'n' nodes and (n-1) edges such that every node can be reachable from every other node in that tree in minimum cost, such tree is known as minimum spanning tree.

① Prim's Algorithm (MST) –



Approach 1:

- Initialize three vectors – key, visited, parent of size n (no. of vertices)

	key	visited	parent
Type:	int	bool	int
Initial value of indices:	∞ i.e INT_MAX	false	-1

- First we will choose one vertex as the starting node. Let's say '0'

So, key[0] = 0 // Because dist. to be travelled to reach starting node will be 0.

- Loop for i = 0 to n
  - ↳ Identify index having min key value & visited[index] = false // means not added to MST yet
  - ↳ Assume this node as current node
  - ↳ Add this to MST ... i.e ... visited[current] = true;
  - ↳ ~~Traverse through all adjacent nodes of current node.~~
  - ↳ If cost to reach adj. node from cur. node is lesser than key[adjacent]

↳ Then → Update : Key[adjacent]
&
→ parent[adjacent] = current.

End of loop.

- Traverse the key vector & find the totalCost.
  ↳ Return totalCost.

Code :—

```
int minimum(vector<int> Key, vector<bool> visited)
{   int min = INT_MAX; j = 0;
    for(int i=0; i< Key.size(); i++)
    {   if( Key[i] <min  && visited[Key] ==false)
        {       min = Key[i];
                j = i;
        }
    }
    if(min == INT_MAX)
        return -1;
    return j;
}
int spanningTree(int n, vector<vector<int>> adj[])
{
    vector<int> key(n, INT_MAX);
    vector<int> parent(n, -1); //will store the parent of each index
    vector<bool> visited (n, false);

    Key[0] = 0;    // Starting vertex.

    for(int count=0; count<n; count++)
    {
        int current = minimum(Key, visited);
        visited[current] = true;
```

```
// traverse through adjacent nodes.
for( auto it: adj[current])
{
    int v = it[0];          //i.e adjacent
    int weight = it[1];
    if (weight < key[v] && key[v] == false)
    {
        key[v] = weight
        parent[v] = current;
    }
}
}
```

// traverse key vector and calculate totalCost of MST -

```
int totalCost = 0;
for( int i=0; i<n; i++)
{   totalCost += key[i];   }
return totalCost;
}
```

Time Complexity -

$O(n * n + E)$ — For traversing through the adjacent nodes. (edges)

- n ↑ For outer for loop
- n ↖ For identifing the index with min. key value & visited[key]=false i.e minimum() fun?

Space Complexity -

$$O(n + n + n) = O(3n) \simeq O(n)$$

* It is not an efficient solution. Since, it's taking $O(n^2+E)$ time complexity.

∴ ∴ ⟶ Approach 2

Approach 2 – [More Efficient]

* Use the priority queue (min_heap) to store a pair<int, int> = <Key[i], i>

* Since, we know that the min heap will be sorted according to the first element of the pair.

* So, we will automatically get min index from the top of the queue. And we don't need the minimum() fun^n for finding the min. index

```
int spanningTree(int n, vector<vector<int>> adj[])
{
    vector<int> Key(n, INT_MAX);
    vector<bool> visited(n, false);
    vector<int> parent(n, -1);

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

    min.push({0, 0});   // initially starting vertex '0'
                        // having Key = 0 will be pushed onto min.
                 // Key  index
    Key[0] = 0; // Starting node

    while (!min.empty())
    {
        auto it = min.top();
        min.pop();

        int current = it.second;
        visited[current] = true;
```

```
for (auto it : adj[current])
{   int adjacent = it[0];
    int weight = it[1];
    if ( weight < Key[adjacent] && visited[adjacent]==false)
    {
        Key[adjacent] = weight;
        parent[adjacent] = current;
        min.push({ Key[adjacent], adjacent });
    }
  }
}
// Calculate totalCost -
int totalCost = 0;
for(int x: Key)
{   totalCost += x;   }
return totalCost;
}
```

Time Complexity :-

$O(n \log(n))$

→ inserting / deleting n-pairs from the priority-queue.

Space Complexity -

$O(n)$

[for single insert/delete operation time complexity is $O(\log n)$

Disjoint Set

- This datastructure is commonly used in Kruskal's Algo. in detecting a cycle.
- Used in EP.

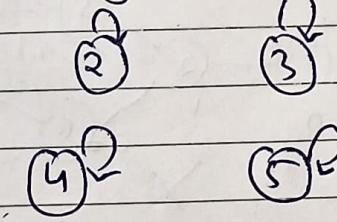
There are multiple components in a graph and if we are asked to tell whether any two nodes/vertices belongs to the same component or not there we use Disjoint Set.

Two Operations → findParent()
→ Union()

* Implementation of Disjoint Set is done by Union By Rank & Path Compression.

Step -

(1) Initially we have a parent array, and every node has himself as the parent.

Initially :- 

	1	2	3	4	5	6	7
Parent:	1	2	3	4	5	6	7

(2) Maintains a rank array → stores the rank of all the nodes
↳ Initial ranks will be 0

	1	2	3	4	5	6	7
Rank array →	0	0	0	0	0	0	0

(3) Do Union operations

~~Union(1,2) :-~~
~~if par(1) = 1 → par(2) = 2~~
~~Ran~~

Union (1, 2) :-

(i) Parent(1) = 1 , Par(2) = 2

Rank(1) = 0 , Rank(2) = 0

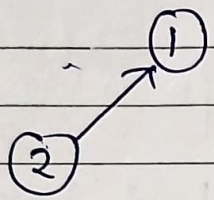
Node → Parent → Rank
1 → 1 → 0
2 → 2 → 0

Both rank 0–0

So, either attach 2 to 1 and increase the rank of 1 by 1. (whom you are attaching)

or attach 1 to 2 and increase rank of 2 by 1

So,



	1	2	3	4	5	6	7
Parent:	1	1	3	4	5	6	7
Rank :	1	0	0	0	0	0	0

Union (2,3) :-

~~Parent (2) = 1~~ ~~Parent (3) = 3~~

~~Rank (2) = 0~~ ~~Rank (3) = 0~~

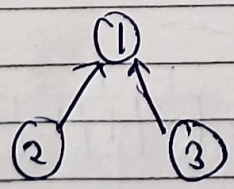
Node → Parent → Rank
2 → 1 → 1
3 → 3 → 0

Connect these 2

↳ Whoever has the smaller rank attach it to the other one.

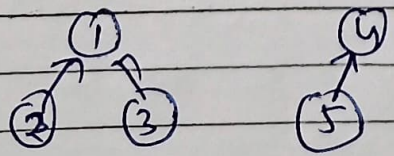
* Whenever you are attaching two similar rank nodes then only increase the rank of the parent node by 1.
Otherwise don't not increase the rank.

∴



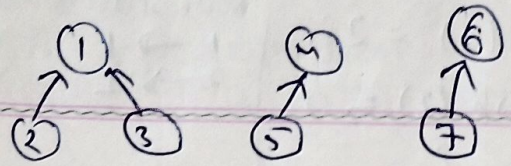
	1	2	3	4	5	6	7
Parent:	1	1	1	4	5	6	7
Rank :	1	0	0	0	0	0	0

Union (4,5) :-



	1	2	3	4	5	6	7
Par:	1	1	1	4	4	6	7
Rank:	1	0	0	1	0	0	0

Union (6, 7) :-



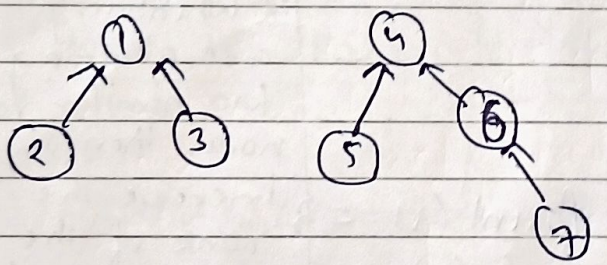
	1	2	3	4	5	6	7
Parent:	1	1	1	4	4	6	6
Rank:	1	0	0	1	0	1	0

Union (5, 6) :-

Node	Parent	Rank
5	4	1
6	6	1

Attach these 2 nodes

↑ Attach 6 to 4



	1	2	3	4	5	6	7
Parent:	1	1	1	4	4	4	6
Rank:	1	0	0	2	0	1	0

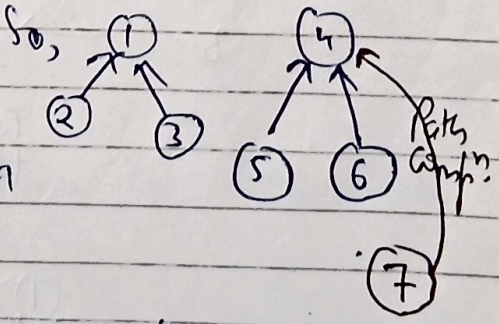
Union (3, 7) :-

Node	Parent	Rank
3	1	1
7 → ~~6~~	4	2

We know that the ultimate parent of 7 is 4

So, why not we perform path compression and attach 7 directly to 4

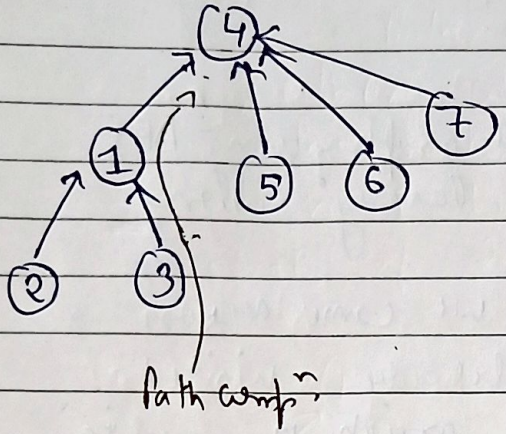
∴ It reduces the complexity.



	1	2	3	4	5	6	7
Parent:	1	1	1	4	4	4	4
Rank:	1	0	0	2	0	1	0

Node	→ Parent	→ Rank
3	→ 1	→ 1
7	→ 4	→ 2

Rank of 4 is greater than that of 1
∴ Attach 1 to 4 so that the length of tree won't increase.



All operations ~~Combindly~~ -

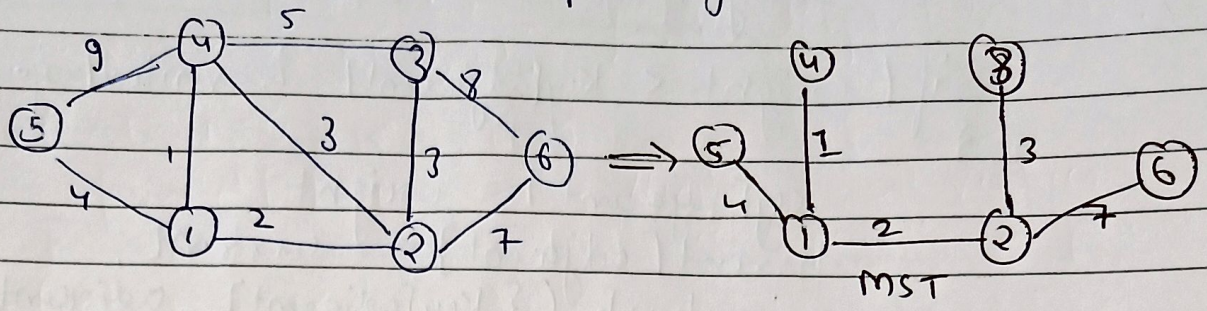
Time Complexity → $O(4\alpha) \simeq O(4)$ → Constant
Mathematically proven.

Space Complexity → $O(n)$
- Rank array
- Parent array.

Krushkal's Algo.
Prim's Algo.

(2) Kruskal's Algorithm —

- Kruskal's algo. uses the **Disjoint set** datastructure to get the minimum spanning tree.



(I) At first create a user-defined datatype using 'struct'. It will store 3 variables namely the source, destination and the cost i.e weight (to reach from src. to dest.)

```
struct node {
    int u, v, w;
    node (int first, int second, int weight)
    {
        u = first;
        v = second;
        w = weight;
    }
}
```

(II) Create a vector named edges of type <struct node> and store all the edges of the graph along with their weight onto it.

→ for sorting use "comparable" function

(III) Sort the "edges" vector in increasing order according to their wright

(IV) In order to use Disjoint Set, create two vectors parent and rank of size n (no. of nodes)

type => parent: int, rank: int

initially -> parent[i] = i, rank[i] = 0

(v) Make two functions findParent() & Union() of Disjoint set.

```
int findParent(int node, vector<int>& parent)
{   if(node == parent[node])
        return node;
    else
        return findParent(parent[node], parent);   // recursive call
}

void Union(int u, int v, vector<int>& rank, vector<int>& parent)   // u, v -> Vertices
{
    u = findParent(u, parent);   // find parent of u & v.
    v = findParent(v, parent);   // Since we will connect the parent node
    if (rank[u] > rank[v])
        parent[v] = u;
    else if (rank[u] < rank[v])
        parent[u] = v;
    else                         // both have same rank
    {   parent[v] = u;           // Connect one to another
        rank[u]++;               // And append rank of parent by 1.
    }
}
```

(VI) Traverse through edges vector and at each iteration check if the vertices of each edge-node does not have same parent, then ~~only~~ union them in order to form a tree.
Otherwise,
if they have same parent then we will not connect them since it will form a cycle.

```
int spanningTree(int n, vector<vector<int>> adj[])
{
    vector<node> edges;    //edges vector of "node" type.
    vector<int> parent(n);
    vector<int> rank(n,0);   } -> for disjoint set DS.

    for(int i=0; i<n; i++)
    {
        parent[i] = i;      } //initially all vertices
                               are seperate components

    for(int i=0; i<n; i++)
    {
                               //Storing all the edges
        int first = i;         of the graph along
        int second =           with their weight onto
        for(auto it: adj[i])   the "edges" vector.
        {   int first = i
            int second = it[0]
            int weight = it[1]
            edges.pushback(node(first, second, weight));
        }
    }
    //Sorting 'edges' vector acc. to weight of edges.
    sort(edges.begin(), edges.end(), compare);

    //3. Traverse through the 'edges' vector and
       calculate the totalCost of MST.*/
    int totalCost = 0;
    for(auto it: edges)
    {
        if(findParent(it.u, parent) != findParent(it.v, parent))
        {
            Union(it.u, it.v, parent, rank);
            totalCost += it.w;
        }
    }
    return totalCost;
}
```

Time Complexity :-

Let, e → no. of edges
n → no of nodes.

$O(e * \log(e)) + O(e * O(4\alpha))$

- $O(e * \log(e))$: For sorting the "edges" vector according to weight of edges.
- $O(e * O(4\alpha))$: For traversing across the "edges" vector
- $O(4\alpha)$: for using the Disjoint Set functions inside the traversal.

$$TC = O(e \log(e))$$

Space → $O(e * 3) + O(n) + O(n)$

- $O(e * 3)$: "edges" vector.
- $O(n)$: parent vector
- $O(n)$: rank vector

$$SC \simeq O(n)$$

Dijkstra Algorithm -

- Similar to the shortest path in undirected graph with unit weights problem.

↳ There we've used queue because the weight of all the edges was 1 unit

↳ But here the weight of edges is not same so at each iteration we would have to choose the edge with minimum weight Hence, we've used priority queue (MIN HEAP) to store weights along with the adjacent vertex.

```
vector<int> dijkstra (int n, vector<vector<int>> adj[],
                                        int src)
{ priority_queue<pair<int,int>, vector<->, greater<>> min;
  vector<int> distance(n, INT_MAX)
  distance[src] = 0;
  min.push({ distance[src] , src });

  while( ! min.empty() )
  {   auto it = min.top();
      min.pop();
      int current = it.second;
      int dist    = it.first;
      for( auto it: adj[current] )
      {     int adjacent = it[0];
            int nextdist = it[1];
        if( dist + nextdist < distance[adjacent])
        {     distance[adjacent] = dist + nextdist;
              min.push({distance[adjacent], adjacent}); }
  } }
  return distance;
}
```

TC = O(n log n)
SC = O(n)

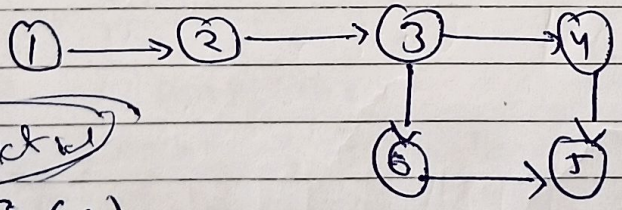
Detect cycle in a directed graph :— [DFS]

* We can't apply the DFS technique which we applied in undirected graph to find cycle in directed graph.

In detecting cycle in undirected graph using DFS
↳ • We maintains a "visited" array along the recursion.
• If at any recursive step we come across a vertex which is already visited then we says that the graph is cyclic

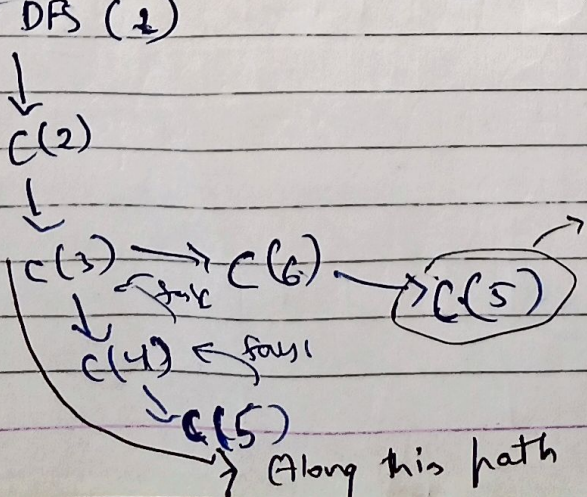
But this could not hold true in case of directed graph because even though the visited index of a vertex is set equal to '1' but it might be possible that, that vertex was visited in any other recursive loop and we can't consider that visit for the current recursive loop.

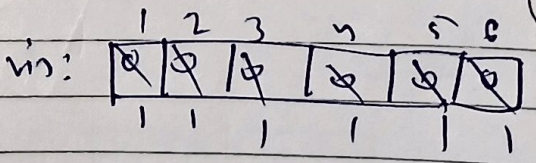
eg:



because

Undirected cycle DFS (1)





Here our algo. will check is vertex 5 already visited, and it will get true answer.
But it's not correct.

∴ Maintain a revisited vector, so, that if any vertex is revisited in the same recursive loop/iteration then it's visited & revisited value will be both equal to 1.

↳ And if it fails then concurrently update the revisited vector ~~index~~ index value for that vertex to '0' and return false.

Using DFS :—

```
bool cyclicDFS(int i, vector<int> adj[], vector<int> & vis
               vector<int> &, reVis)
{   vis[i] = 1;
    reVis[i] = 1;
    for(int adjacent : adj[i])
    {   if(vis[adjacent] == 0)
        {   if(cyclicDFS(adjacent, adj, vis, reVis))
                return true;
        }
        if(reVis[adjacent] == 1)   ← That vertex is revisited in the same recursive iteration.
            return true;
    }
    reVis[i] = 0;   // Otherwise update to 0 back.
    return false;
}
bool isCyclic(int V, vector<int> adj[])
{   vector<int> vis(V,0);
    vector<int> reVis(V,0);
    for(i=0; i<V; i++)
    {   if(vis[i] == 0 && cyclicDFS(i, adj, vis, reVis))
            return true;
    }
    return false;
}
```

Time - O(N+E)
Space - O(N)+O(N)
Auxiliary stack space - O(N)

Topological Sorting [DFS] –

- Linear ordering of vertices such that if there is an edge [u → v], then u appears before v in that ordering.

* Only possible in Directed Acyclic Graph.

Steps :-

1) Maintain a "visited" vector & a "stack"
2) Iterate over all the vertices one by one
   - ↳ If a vertex is not visited
   - ↳ Call topological sort for this vertex
   - ↳ Recursively call topological sort for adjacent vertices of the current vertex.
   - ↳ At the end of the recursive iteration push that current node onto the stack and return back to the previous recursive step and there also push onto stack the current vertex of that step.

Repeat this until that recursive call is finished / ended.

Time → $O(N + E)$ (N → nodes (total), E → edges (total))

Space → $O(N) + O(N)$
- ↳ vector
- ↳ stack

Aux. Stack Space → $O(N)$

```
void topologicalDFS(int cur, vector<int> adj[],
                    vector<int> &visited, stack<int>& stk)
{
    visited[cur] = 1;
    for(int adjacent : adj[cur])
    {
        if(visited[adjacent] == 0)
            topologicalDFS(adjacent, adj, visited);
    }
    stk.push(cur);
}
vector<int> topoSort(int V, vector<int> adj[])
{   vector<int> visited(V, 0);  stack<int> stk;
    for(int i=0; i<V; i++)
    {   if(visited[i] == 0)
            topologicalDFS(i, adj, visited, stk);
    }
    visited.clear();
    while(!stk.empty())
    {   visited.push_back(stk.top());
        stk.pop();
    }
    return visited;
}
```

Topological Sorting [BFS] :-
(Kahn's Algorithm)

1) Form an indegree vector which stores the indegree of all the vertices of the graph.
2) Maintain a queue which stores only those vertices which has 0 indegree.
3) Initially, vertices having 0 indegree will be pushed/inserted onto the queue.

4) ~~Iterate over~~
While queue is not empty
↳ Pop from queue and store as current node in vector
↳ Iterate over the adjacent nodes of the current node and decrease their indegree value by 1.
↳ If any of those node's indegree becomes 0 push it onto the stack

5) Return vector.

```
vector<int> topoSort(int V, vector<int> adj[])
{   vector<int> inDegree(V,0);
    vector<int> ans;
    for(i=0; i<V; i++)
    {   for(int adjacent: adj[i])
            inDegree[adjacent]++;
    }
    queue<int> q;
    for(i=0; i<V; i++)
    {   if(inDegree[i]==0)
            q.push(i);
    }
    while(!q.empty())
    {   int cur = q.front();
        q.pop();
        vec.push_back(cur);
        for(int adjacent: adj[cur])
        {   inDegree[adjacent]--;
            if(inDegree[adjacent]==0)
            {   q.push(adjacent); }
        }
    }
    return vec;
}
```

Detect cycle in a ~~Graph~~ Directed Graph using BFS [Kahn's algo] -

We know that Kahn's Algo. is used in Topological Sorting to produce a vector of size "N" as output if the graph is DAG.

But if we use Kahn's Algo. for Directed Cyclic graph then it is sure that this algo will not return a vector of same size as "N"

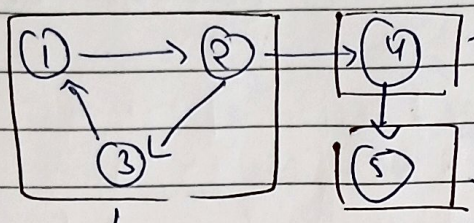
So, at end we will compare the ~~length~~ size of vector produced by the Kahn's Algo. with the no. of nodes, if it ~~Compare~~ comes out same then we can say the the graph is DAG else

the given graph is ~~a~~ Cyclic.

Kosaraju's Algorithm for Strongly Connected Components (SCC)

- Strongly Connected Components are the sub-parts of the graph in which if we start from a particular node then we can reach every other node present in that component.

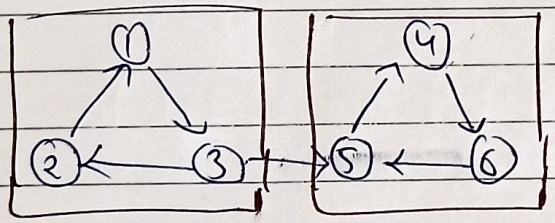
eg:



Here in this component we can reach each and every node from any other node present within component.

Total → 3 components

eg:



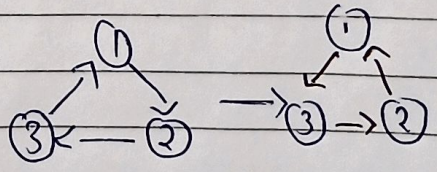
(2)

Algorithm -

I) Sort all the nodes in order of the finishing time → Topological Sort

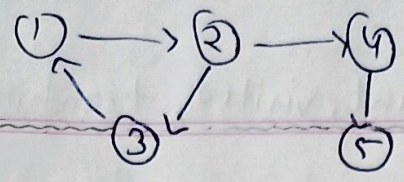
II) Transpose the graph.

i.e

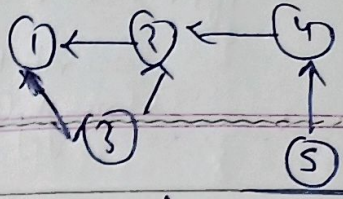


↳ So, that we will remain only in a single component while traversing, and don't end up going to the other parts of graph

eg:



Transpose



Here, from (2) we can go to two parts of the graph either (3) or (4)

But, here, from (2) we again reached to (1), i.e. within the same part only. And that's the advantage.

III) Now, we will do DFS according to the finishing time... which we have obtained through topological sort in terms of a Stack in which elements are inserted in order of ~~their~~ their finishing time.

```
int kosaraju (int V, vector<int> adj[])
{   vector<int> visited(V,0);
    stack<int> stk;
    Do topo sort ⟶ obtain stack

    vector<int> transpose[v];
    for(int i=0; i<V; i++)                    // transpose the graph
    {   for( int gadjacent: adj[i])
        {   transpose[adjacent].push_back(i); }
    }

// DFS traversal and obtaining count.
int count=0;  // count of SCC
visited.assign(V,0);   // re-initialize visited vector.
while(!stk.empty())
{
    int current = stk.top();
    stk.pop();
```

```
        if(visited[current] == 0)
        {
            count++;
            StronglyConnectedDFS(current, visited, transpose);
        }
    }
    return count;
}

void StronglyConnectedDFS(int cur, vector<int>& visited,
                          vector<int> transpose[]);
{   visited[cur] = 1;
    for(int adjacent : transpose[cur])
    {   if(visited[adjacent] == 0)
        {   StronglyConnectedDFS(adjacent, vis., transp.);
        }                          ↳ Recursive call
    }
}
```

Time Complexity --

$O(N+E) + O(N+E) + O(N+E) = O(N+E)$

for toposort — for transpose — for DFS traversal

Space Complexity --

$O(N) + O(N+E) + O(N) = O(N+E)$

for visited vector — for transpose graph — for stack

Bellman Ford Algorithm :—

- We know that Dijiktra's Algo. gives us the shortest path from the source node to all the other nodes.

But, Dijiktra's Algo. doesn't works for negative edges

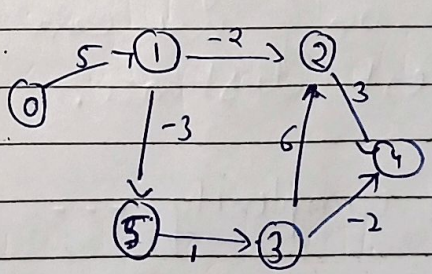
↳ Imagine we have -ve edges in graph and we had applied dijiktra's algo. then what will happen is that the distance will keep on decreasing and there will be a INFINITE LOOP.

- Bellman Ford Algo. is also capable of finding shortest path. But apart from this it is also capable of detecting negative cycle.

- In order to implement Bellman Ford Algo. for undirected graphs, first we would have to convert the Undirected Graph into Directed Graph.

* Inshort, Bellman Ford works for Directed Graph.

Algorithm :—



(I) Relax all the edges (N-1) times.

means => if (dist[u] + wt < dist[v])
then dist[v] = dist[u] + wt;

(u,v) → wt

↳ maintain a dist[] array.

Source

Initially: dist[] →

0	∞	∞	∞	∞	∞
0	1	2	3	4	5

So, here the algorithm states that when you do relaxation (n-1) time, you will get an array (dist) having the shortest distances from source to all the other nodes.

→ for all the edges

* But if we relax the edges for one more time, and if the distances reduces one more time then it can be concluded that the graph contains NEGATIVE CYCLE.

Why (n-1) times relaxation :— INTUTION

↳ At each relaxation one of the node of the graph will get updation in its distance array.

↳ So, we have (n-1) nodes to be updated. (since ... the source node distance is already taken as 0)

↳ Hence, we need (n-1) times relaxation of all the edges.

↳ Then only we can calculate the shortest distance.

Time Complexity --- $O(E) * O(n-1)$

E → Traversal of 'E' edges

$n-1$ → $(n-1)$ times.

Space Complexity --- $O(n)$

↓

distance vector

```
int isNegativeWeightCycle(int n, vector<vector<int>> edges)
{
    vector<int> distance(n, INT_MAX);   // Vector to store shortest path.
    distance[0] = 0;                    // shortest path to the source node will be 0.
    for(int i=1; i<=n-1; i++)           // (n-1) times relaxations on
    {
        for(auto it:edges)              // all the edges
        {
            int u = it[0];
            int v = it[1];
            int weight = it[2];
            if( distance[u] != INT_MAX && dist[u]+wt < dist[v] )
            {
                dist[distance[v] = distance[u] + weight; }
        }
    }

    for(auto it : edges)                // Relaxing 1 more time
    {   int u = it[0];
        int v = it[1];
        int weight = it[2];

        /* While relaxing one more time, if the shortest path for a particular ~~edge~~ node reduces one more time... then it means there is a -ve cycle and this weight will keep on reducing infinitely */

        if( dist[u] + weight < dist[v] )
        {   return 1; } // i.e ... negative cycle exists
    }

    return 0;   // ..-ve cycle doesn't exist
}
```

THANK

YOU