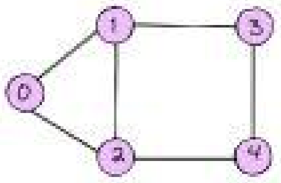
#60daysofcode
Day 2
Lecture - 133

## Graph Representation

## Adjacency Matrix

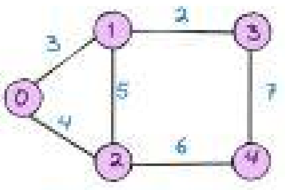


Unweighted Graph

Vertex : {0,1,2,3,4} -> 5
Edge : {(0,1),(1,2),(0,2),(2,4),(3,4),(1,3)}

|   | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 |
| 2 | 1 | 1 | 0 | 0 | 1 |
| 3 | 0 | 1 | 0 | 0 | 1 |
| 4 | 0 | 0 | 1 | 1 | 0 |

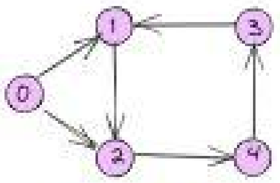
2D Array



Weighted Graph

|   | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 3 | 4 | 0 | 0 |
| 1 | 3 | 0 | 5 | 2 | 0 |
| 2 | 4 | 5 | 0 | 0 | 6 |
| 3 | 0 | 2 | 0 | 0 | 7 |
| 4 | 0 | 0 | 6 | 7 | 0 |

2D Array



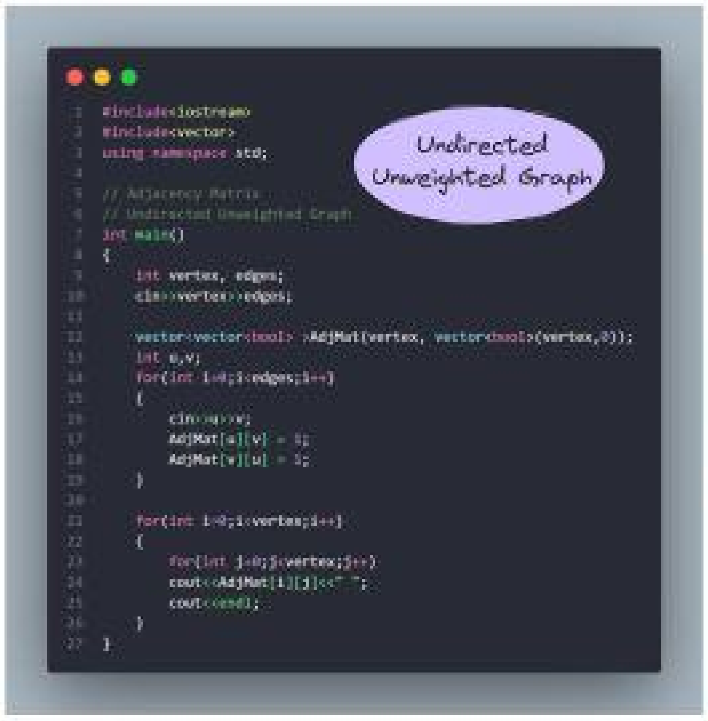
Unweighted Directed Graph

|   | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 | 1 |
| 3 | 0 | 1 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 1 | 0 |

2D Array

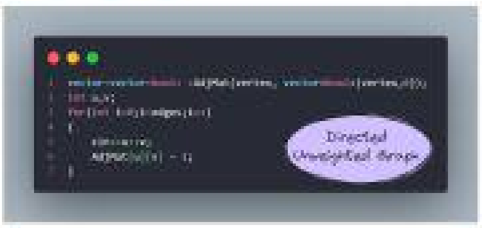
Similarly Weighted Directed Graph

CODE PART -> TC - O(V^2)
SC - O(V^2)



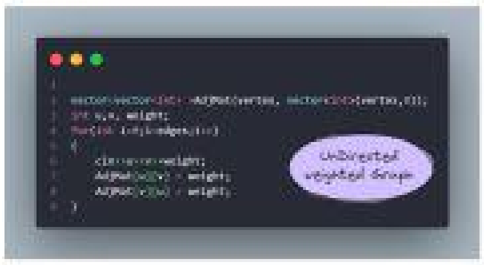
Undirected Unweighted Graph

```
#include<iostream>
#include<vector>
using namespace std;

// Adjacency Matrix
// Undirected Unweighted Graph
int main()
{
    int vertex, edges;
    cin>>vertex>>edges;

    vector<vector<bool> >AdjMat(vertex, vector<bool>(vertex,0));
    int u,v;
    for(int i=0;i<edges;i++)
    {
        cin>>u>>v;
        AdjMat[u][v] = 1;
        AdjMat[v][u] = 1;
    }

    for(int i=0;i<vertex;i++)
    {
        for(int j=0;j<vertex;j++)
        cout<<AdjMat[i][j]<<" ";
        cout<<endl;
    }
}
```



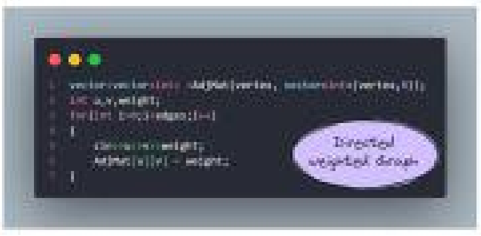
Directed Unweighted Graph

```
vector<vector<bool> >AdjMat(vertex, vector<bool>(vertex,0));
int u,v;
for(int i=0;i<edges;i++)
{
    cin>>u>>v;
    AdjMat[u][v] = 1;
}
```



Undirected Weighted Graph

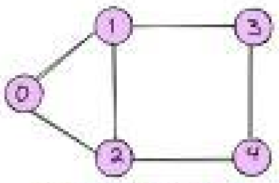
```
vector<vector<int> >AdjMat(vertex, vector<int>(vertex,0));
int u,v, weight;
for(int i=0;i<edges;i++)
{
    cin>>u>>v>>weight;
    AdjMat[u][v] = weight;
    AdjMat[v][u] = weight;
}
```



Directed Weighted Graph

```
vector<vector<int> >AdjMat(vertex, vector<int>(vertex,0));
int u,v,weight;
for(int i=0;i<edges;i++)
{
    cin>>u>>v>>weight;
    AdjMat[u][v] = weight;
}
```

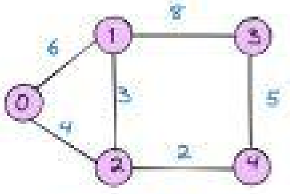
## Adjacency list



Unweighted Graph

0 -> 1 2
1 -> 0 3 2
2 -> 0 4 1
3 -> 1 4
4 -> 2 3

```
vector<int>AdjList[5]
AdjList[0].push-back(1)
AdjList[1].push_back(0)
```

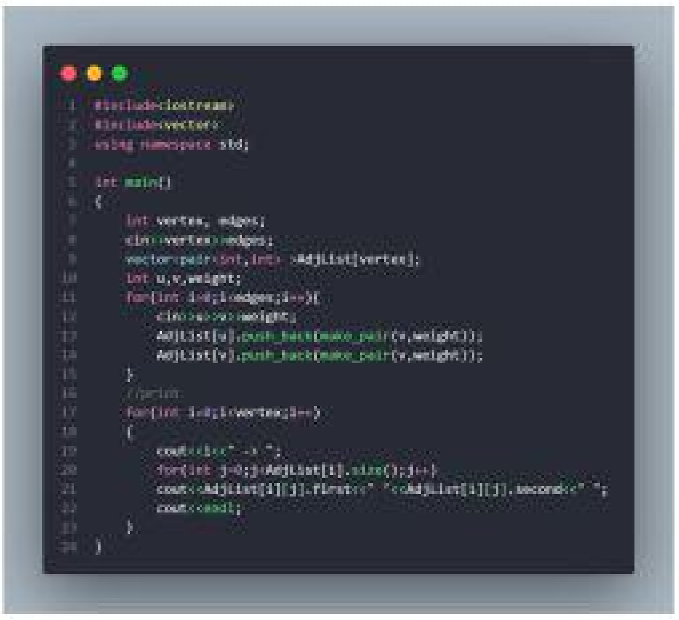


Weighted Graph

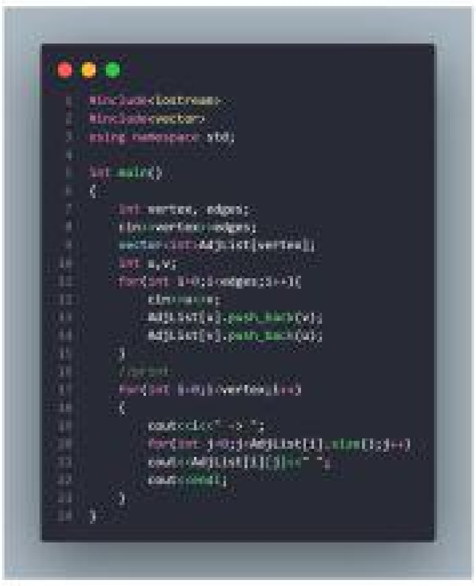
0 -> (1,6) (2,4)
1 -> (0,6) (3,8) (2,3)
2 -> (0,4) (4,2) (1,3)
3 -> (1,8) (4,5)
4 -> (2,2) (3,5)

CODE PART -> TC - O(V+E)
SC - O(V+E) -> Worst Case -> TC - O(V^2)
SC - O(V^2)

E = V^2 - Complete Graph



```
#include<iostream>
#include<vector>
using namespace std;

int main()
{
    int vertex, edges;
    cin>>vertex>>edges;
    vector<pair<int,int> >AdjList[vertex];
    int u,v,weight;
    for(int i=0;i<edges;i++){
        cin>>u>>v>>weight;
        AdjList[u].push_back(make_pair(v,weight));
        AdjList[v].push_back(make_pair(u,weight));
    }
    //print
    for(int i=0;i<vertex;i++)
    {
        cout<<i<<" -> ";
        for(int j=0;j<AdjList[i].size();j++)
        cout<<AdjList[i][j].first<<" "<<AdjList[i][j].second<<" ";
        cout<<endl;
    }
}
```



```
#include<iostream>
#include<vector>
using namespace std;

int main()
{
    int vertex, edges;
    cin>>vertex>>edges;
    vector<int>AdjList[vertex];
    int u,v;
    for(int i=0;i<edges;i++){
        cin>>u>>v;
        AdjList[u].push_back(v);
        AdjList[v].push_back(u);
    }
    //print
    for(int i=0;i<vertex;i++)
    {
        cout<<i<<" -> ";
        for(int j=0;j<AdjList[i].size();j++)
        cout<<AdjList[i][j]<<" ";
        cout<<endl;
    }
}
```

## Which is Better

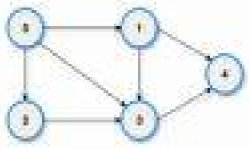
| Parameter | Adj Matrix | Adj List |
|---|---|---|
| Add Edge | O(1) | O(1) |
| Remove Edge | O(1) | O(v) |
| Edge Exist? | O(1) | O(v) |
| SC | O(v^2) | O(v+E) -> O(v^2) |
| When Used | Dense Graph<br>EdgesMore | Sparse Graph<br>No. of Edges Less |

Facebook - Sparse Graph -> Ajacency List

### Key Differences Between Graph and Tree

- **Cycles:** Graphs can contain cycles, while trees cannot.
- **Connectivity:** Graphs can be disconnected (i.e., have multiple components), while trees are always connected.
- **Hierarchy:** Trees have a hierarchical structure, with one vertex designated as the root. Graphs do not have this hierarchical structure.
- **Applications:** Graphs are used in a wide variety of applications, such as social networks, transportation networks, and computer science.

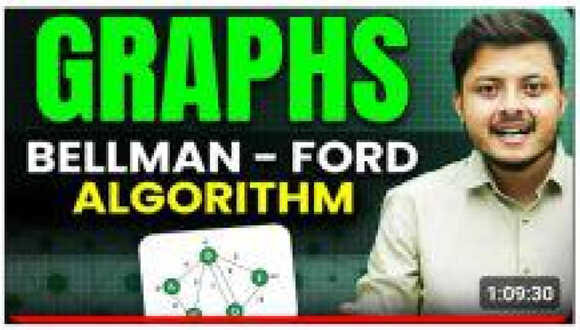
E.g. – Directed graph

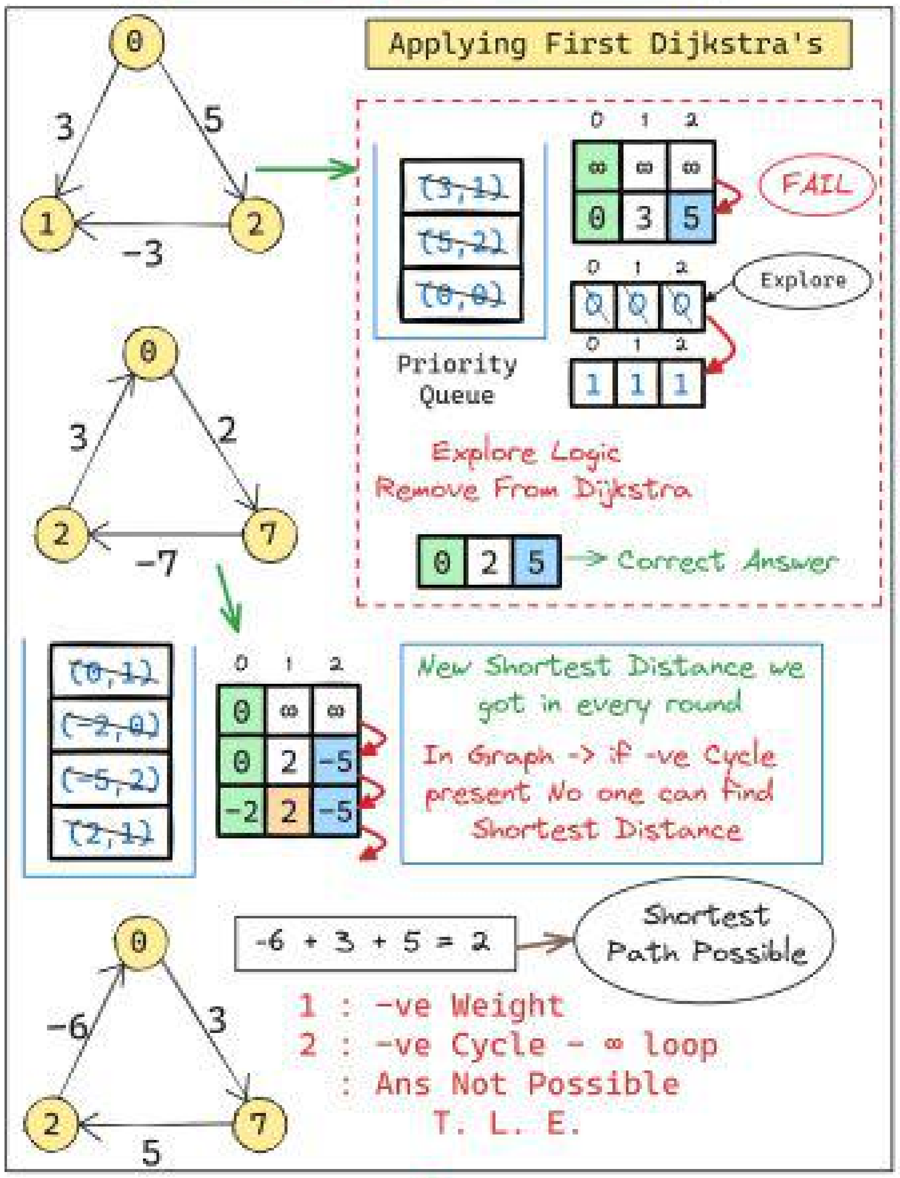


- indegree(0) = 0
- indegree(1) = 1
- indegree(2) = 1
- indegree(3) = 3
- indegree(4) = 2

- outdegree(0) = 3
- outdegree(1) = 2
- outdegree(2) = 1
- outdegree(3) = 1
- outdegree(4) = 0

# #60daysofcode -> Day 14/60
# Lecture - 145





Priority Queue:

| |
|---|
| ~~(3,1)~~ |
| ~~(5,2)~~ |
| ~~(0,0)~~ |

| 0 | 1 | 2 |
|---|---|---|
| ∞ | ∞ | ∞ |
| 0 | 3 | 5 |

| 0 | 1 | 2 |
|---|---|---|
| ~~0~~ | ~~0~~ | ~~0~~ |

| 0 | 1 | 2 |
|---|---|---|
| 1 | 1 | 1 |

Explore Logic
Remove From Dijkstra

| | | |
|---|---|---|
| 0 | 2 | 5 |

-> Correct Answer

| |
|---|
| ~~(0,1)~~ |
| ~~(-2,0)~~ |
| ~~(-5,2)~~ |
| ~~(2,1)~~ |

| 0 | 1 | 2 |
|---|---|---|
| 0 | ∞ | ∞ |
| 0 | 2 | -5 |
| -2 | 2 | -5 |

New Shortest Distance we got in every round

In Graph -> if -ve Cycle present No one can find Shortest Distance

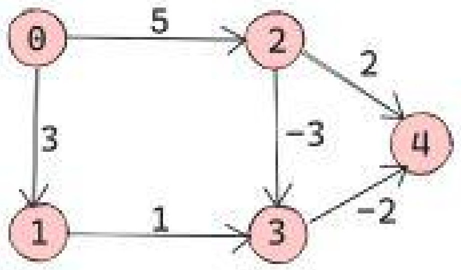
-6 + 3 + 5 = 2

1 : -ve Weight
2 : -ve Cycle - ∞ loop
  : Ans Not Possible
      T. L. E.

## Bellman Ford Algorithm

Help In :
1. Only For Directed Graph
2. If -ve Cycle present in Graph - Ans Not Found
   Else Ans Found

Working :
1. (Relax your all edges) * (v-1) times
   → dist[u] + wt < dist[v]
     dist[v] = dist[u] + wt
2. Relax Your Edges one more to Detect Cycle



| U | V | W |
|---|---|---|
| 2 | 4 | 2 |
| 3 | 4 | -2 |
| 2 | 3 | -3 |
| 0 | 2 | 5 |
| 1 | 3 | 1 |
| 0 | 1 | 3 |

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 0 | ∞ | ∞ | ∞ | ∞ |
| 0 | ∞ | 5 | 2 | 7 |
| 0 | 3 | 5 | 2 | 0 |

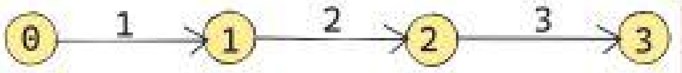
Answer

dist[u] + W < dist[v]
dist[v] = dist[u] + w

(v-1)
Times

dist[2] + 2 < dist[4]
dist[3] + -2 < dist[4]
dist[2] + -3 < dist[3]
dist[0] + 5 < dist[2] ✓
dist[1] + 1 < dist[3]
dist[0] + 3 < dist[1] ✓

## Why This Algorithm Working - Deep Dive 💪



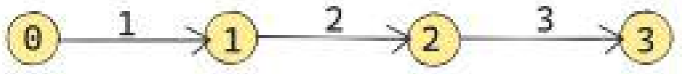
| 0 | 1 | 2 | 3 |
|---|---|---|---|
| 0 | ∞ | ∞ | ∞ |
| 0 | 1 | 3 | 6 |

1 dist[2] + 3 < dist[3]
2 dist[1] + 2 < dist[2]
3 dist[0] + 1 < dist[1]

| U | V | W |
|---|---|---|
| 2 | 3 | 3 |
| 1 | 2 | 2 |
| 0 | 1 | 1 |

3 Time

Vertex → Dijkstra → infinite
Edges → Bellman → Fixed



| 0 | 1 | 2 | 3 |
|---|---|---|---|
| 0 | ∞ | ∞ | ∞ |
| 0 | 1 | 3 | 6 |

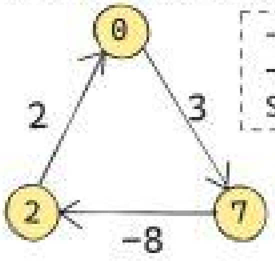
1. dist[0] + 1 < dist[1]
2. dist[1] + 2 < dist[2]
3. dist[2] + 3 < dist[3]

| U | V | W |
|---|---|---|
| 0 | 1 | 1 |
| 1 | 2 | 2 |
| 2 | 3 | 3 |

3 Time

You have relaxed all the edges
and if there is no change in distance
Then This is Answer no need to move
forward and relax edges

CODE PART → Gfg



-8 + 3 + 2 = -3 (-ve)
-ve Cycle Present
Shortest Path Impossible

2 Time + 1 Time(Cycle)

Dist. Changes → Ans Not Possible

```
vector<int> bellman_ford(int V,
vector<vector<int>>& edges, int S) {
        vector<int>dist(V,1e8);
        // 1e8 10 power 8
        dist[S] = 0;
        int e = edges.size();

        for(int i=0;i<V-1;i++){
            // Relax all the edges
            bool flag = 0;
            for(int j=0;j<e;j++){
                int u = edges[j][0];
                int v = edges[j][1];
                int wt = edges[j][2];

                if(dist[u]==1e8)
                continue;
                if(dist[u]+wt<dist[v]){
                    flag = 1;
                    dist[v] = dist[u]+wt;}}
            if(!flag)
            return dist;}

        // To deduct the cycle
        for(int j=0;j<e;j++){
                int u = edges[j][0];
                int v = edges[j][1];
                int wt = edges[j][2];

                if(dist[u]==1e8)
                continue;
                if(dist[u]+wt<dist[v]){
                    // cycle deducted
                    vector<int>ans;
                    ans.push_back(-1);
                    return ans;}}
        return dist;
    }
```

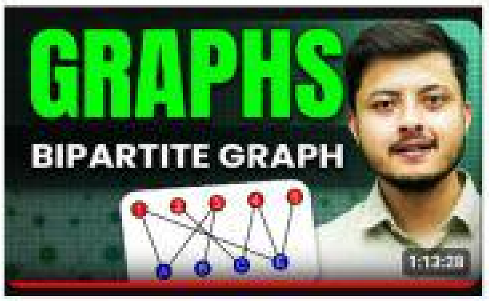
Optimized
Code & Edge
Cases Handle

Time Complexity :
Worst Case → V*E
Best Case → E
Space Complexity:
Space → V

Undirected Graph
-> Dijkstra ✓
-> Bellman ✗

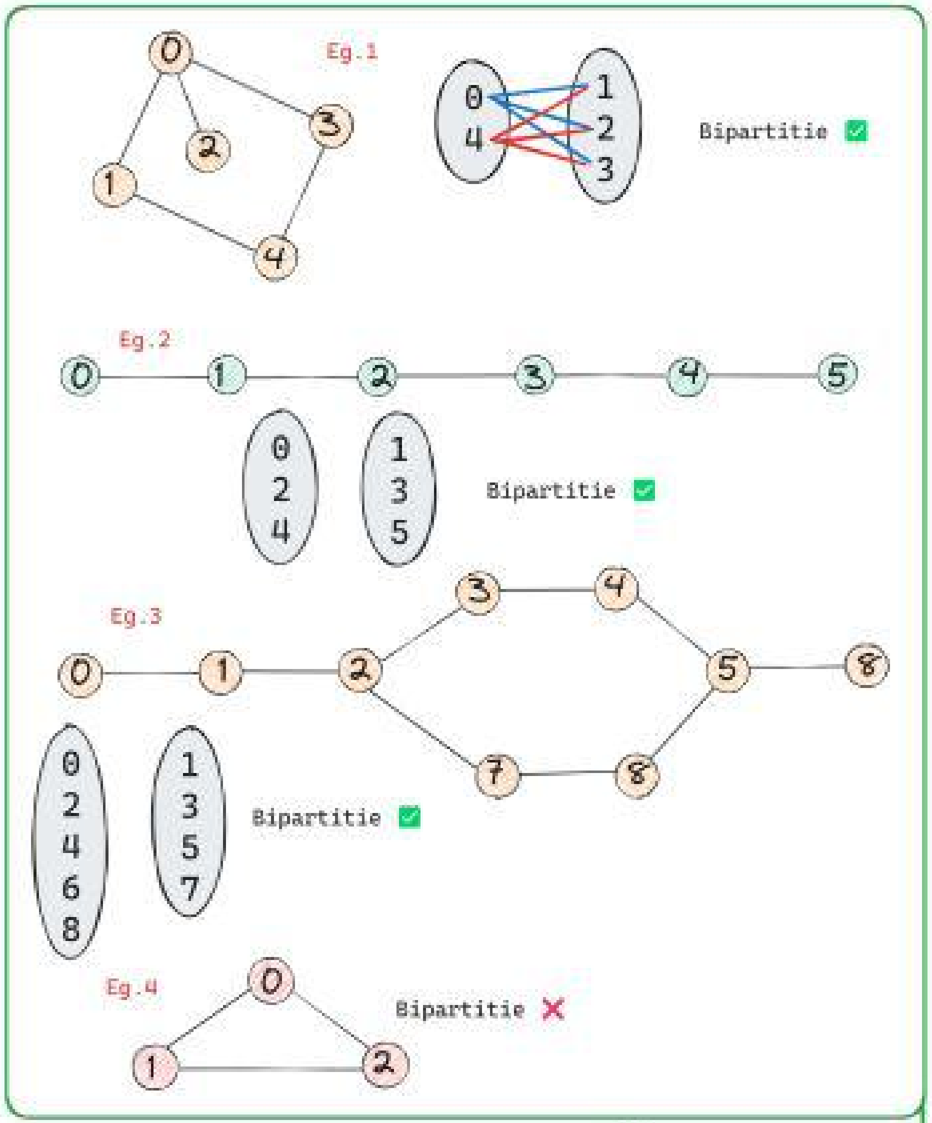
Undirected Graph
with -ve Weight
-> Dijkstra ✗
-> Bellman ✗

#60daysofcode
Day 7
Lecture - 138



# BIPARTITE GRAPH | BFs + DFs

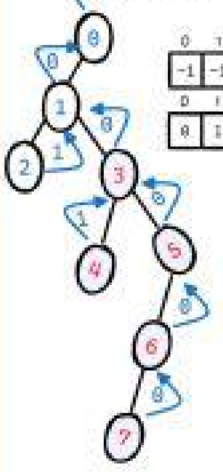
It is a graph in which the vertices can be divided into two disjoint sets, such than no 2 vertices within the same set are adjacent. In other words, it is a graph in which every edge connects a vertex of one set to a vertex of other set.

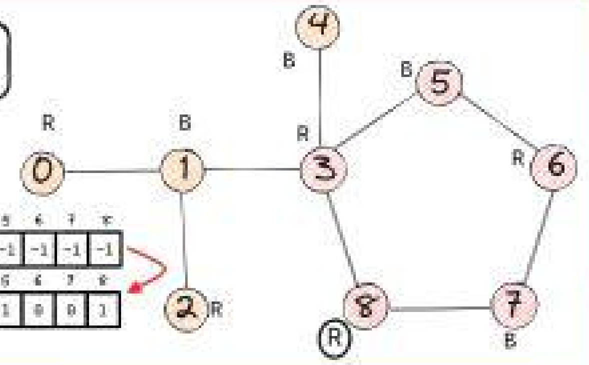


## DFS

R → 0
B → 1

Bipartitie ✗



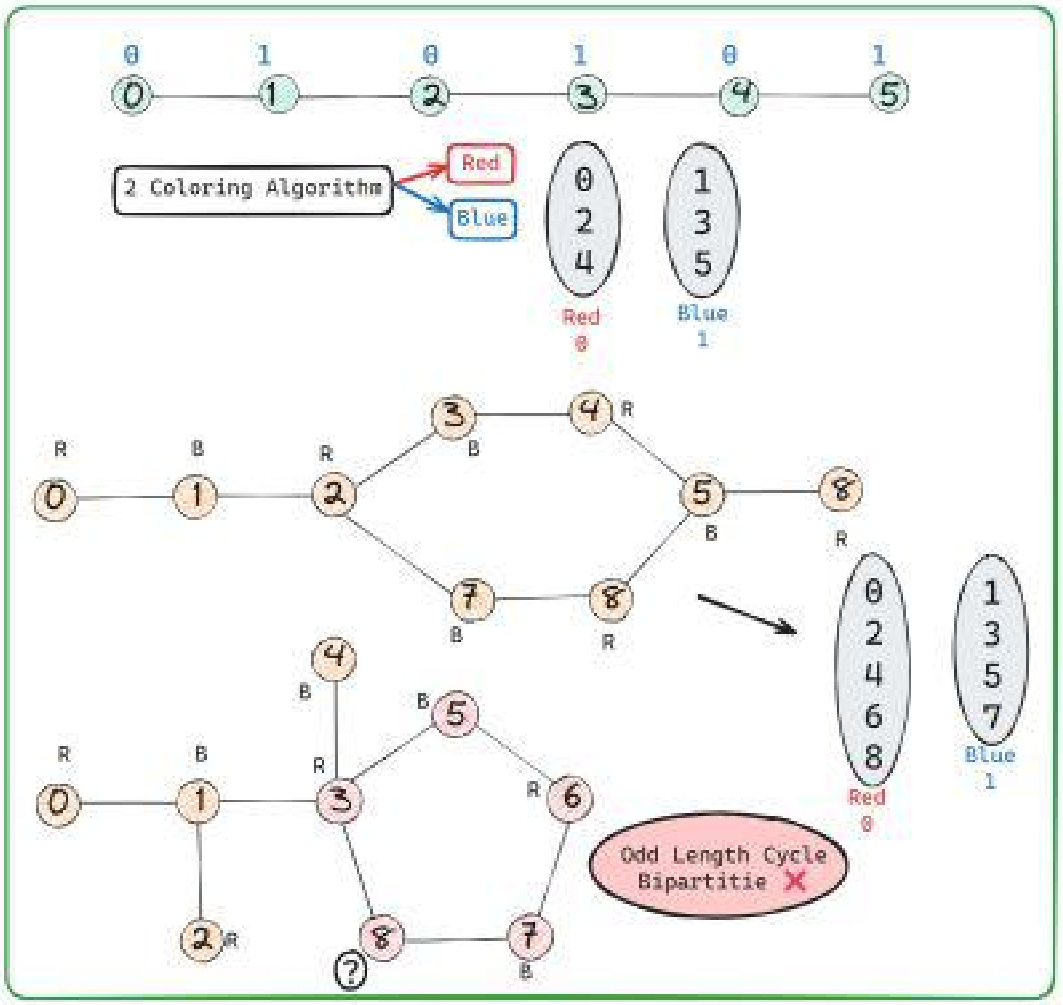


| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |

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

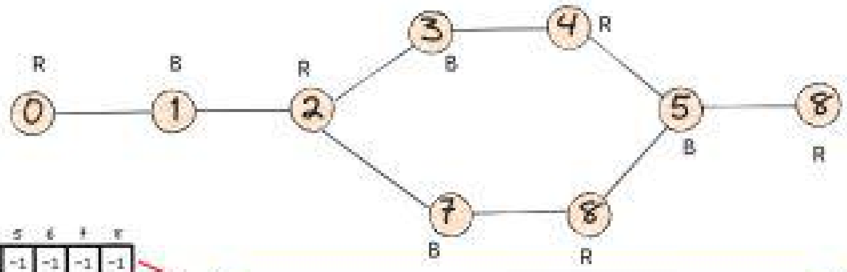
```
bool IsBip(int node, vector<int>adj[],
               vector<int>&color){
   for(int j=0;j<adj[node].size();j++){
        if(color[adj[node][j]] == -1){
   color[adj[node][j]] = (color[node]+1)%2;
        if(!IsBip(adj[node][i], adj, color))
                return 0;
        }
        else{
            if(color[node]==color[adj[node][j]])
                return 0;
         }
    }
    return 1;
}
```

TC - O(V+E)
SC - O(V)



## BFS

R → 0
B → 1



Queue

Color

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

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

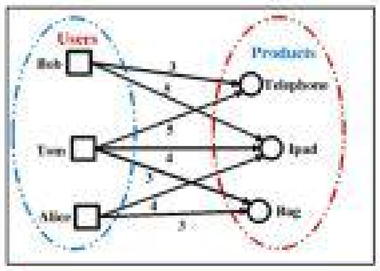
CODE PART →

TC - O(V+E)
SC - O(V)

Steps
1. It will Look at its all neighbour
   - If Color is not assigned,
        - assign a color to them
          (Khud se opposite)
        - Push the neighbour into queue
   - Else → Color is already assigned
        - if neighbour color is same as present
          node we will declare it as not a
          bipartite graph.

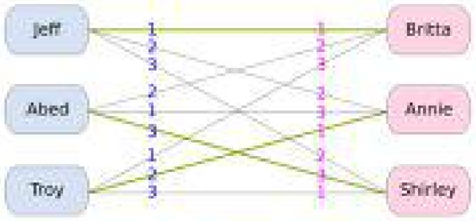
## Real Life Example

1. Recommendation System



u1 → p1 p2 : p3 recommend
u2 → p1 p2 p3
u3 → p2 p3 : p1 recommend

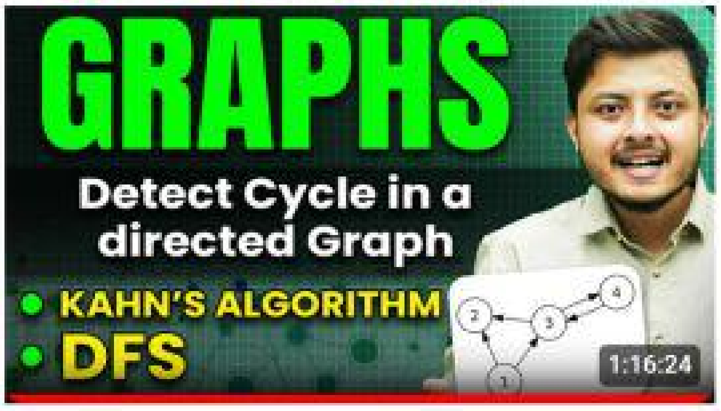
2. Stable Marriage Problem



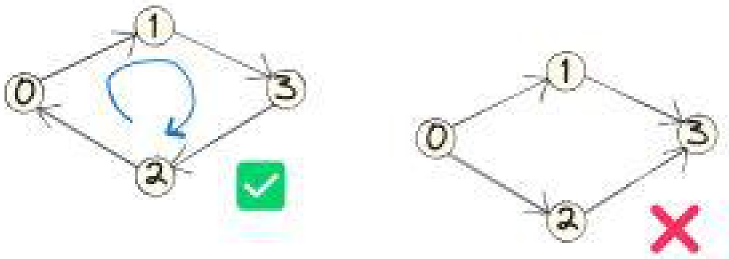
On the whole, bipartite graphs prove themselves in different fields, especially in matching problems, recommendation systems, social networks, and so on.

Bipartite graphs that capture relationships between various entities in an effacious way become the essential tool for making decision and analysis, thus, they are essential in diverse real-life applications.

#60daysofcode
Day 6
Lecture - 137

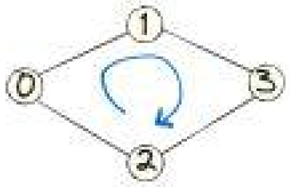


# Cycle Detection in an directed Graph (Bfs+Dfs)

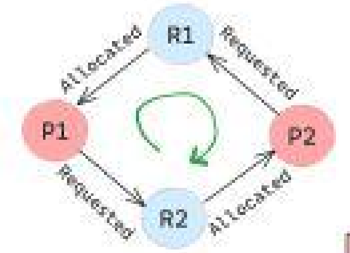


Revision of Undirected

if There is node, if its neighbour is already visited and it's not a parent. Cycle ✓



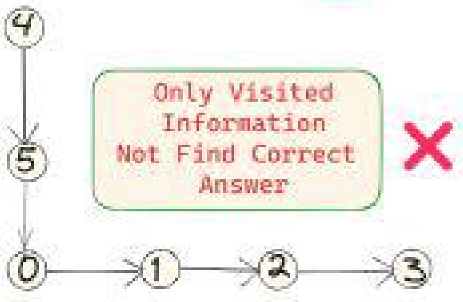
Real Life Example



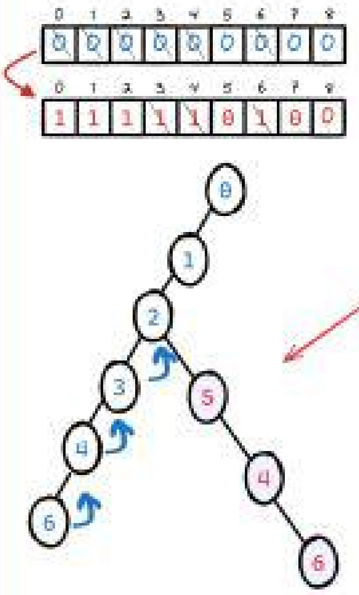
Deadlock
→ Mutual Exclusion
→ Circular Wait
→ Hold And Wait
→ No Preemption

Banker's Algorithm Also Used to Detect Deadlock

Detect And Recover

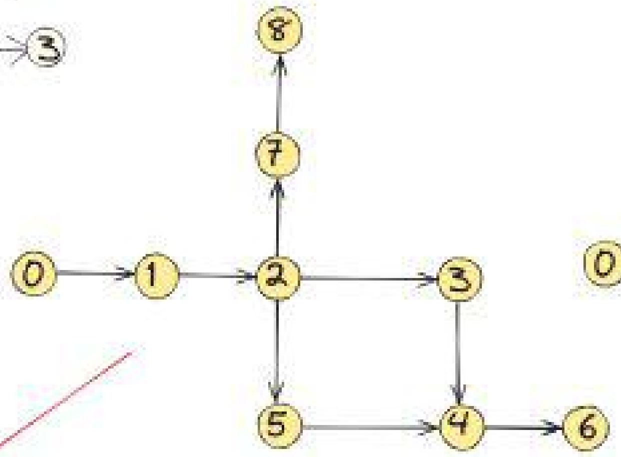


Only Visited Information Not Find Correct Answer



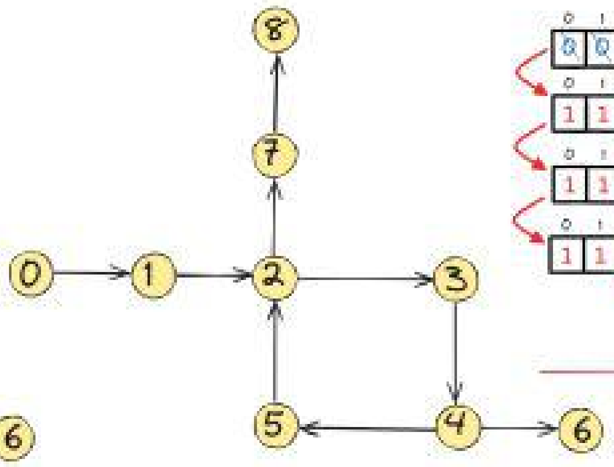
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

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

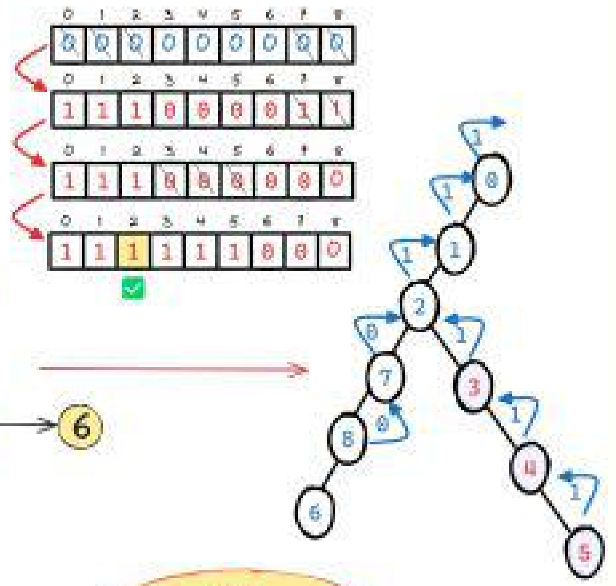


Visited Only Information → Cycle ✓

If a Node Appeared more than once in a path, then we will call it Cycle → Cycle ✗



Cycle ✓



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

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

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

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

✓

Not Optimized Code

```
bool DetectCycle(int Node,vector<int>adj[i],
    vector<bool>&path,vector<bool>&visited ){
    path[node] = 1, visited[Node] = 1;
    for(int j=0;j<adj[node].size();j++){
        if(path[adj[node][j]]
            return 1;

    if(DetectCycle(adj[node][j], adj, path,visited))
            return 1;
        }
    path[node] = 0;
        return 0;
}
```

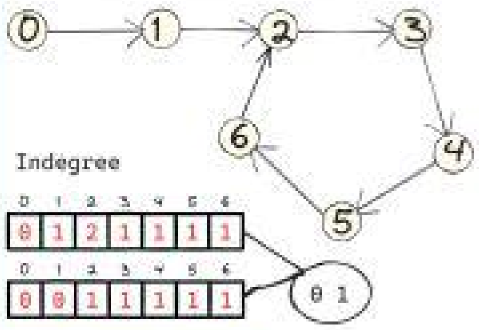
Optimized

```
if(visited[adj[node][j]])
    continue;
```

TC - O(V+E)
SC - O(V)

CODE PART → GfG

## BFs(Khan's Algorithm)



| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 1 | 1 | 1 | 1 |

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 1 | 1 | 1 |

DAG : Count = = V
DCG : Count ! = V

HW : Topological Sort - DFs ??

CODE PART → GfG

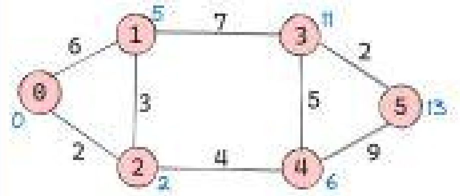
```
bool isCyclic(int V, vector<int> adj[]){
    vector<int>InDeg(V,0);
    for(int i=0;i<V;i++){
    {
        for(int j=0;j<adj[i].size();j++)
        InDeg[adj[i][j]]++;
    }
        queue<int>q;
for(int i=0;i<V;i++)
    if(!InDeg[i])
       q.push(i);
int count = 0;
```

```
while(!q.empty()){
    int node = q.front();
     q.pop();
     count++;
for(int j = 0;j<adj[node].size();j++)
{
     InDeg[adj[node][j]]--;
     if(!InDeg[adj[node][j]])
        q.push(adj[node][j]);
    }
}

return count ≠ V;
```

TC - O(V+E)
SC - O(V)

# Dijikstra's Algorithm

Single Source Shortest Path Alogrithm



| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 0 | ∞ | ∞ | ∞ | ∞ | ∞ |
| 0 | 6 | 2 | ∞ | ∞ | ∞ |
| 0 | 5 | 2 | ∞ | 6 | ∞ |
| 0 | 5 | 2 | 12 | 6 | ∞ |
| 0 | 5 | 2 | 11 | 6 | 15 |
| 0 | 5 | 2 | 11 | 6 | 13 |

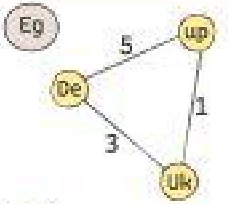
Vector<vector<int>>adj[N];

Explore

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| ~~0~~ | ~~0~~ | ~~0~~ | ~~0~~ | ~~0~~ | ~~0~~ |
| 1 | 1 | 1 | 1 | 1 | 1 |

```
0 → {1,6}
    {2,2}
1 → {0,6}
    {3,7}
    {2,3}
2 → {0,2}
    {1,3}
    {4,4}
3 → {1,7}
    {5,2}
    {4,5}
4 → {2,4}
    {3,5}
    {5,9}
5 → {3,2}
    {4,9}
```

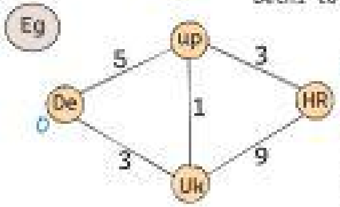
Eg



| Del | Uk | UP |
|---|---|---|
| 0 | 3 | 4 |

Shortest Distance B/w Delhi to UP

Eg



| Del | Uk | UP | Hr |
|---|---|---|---|
| 0 | 3 | 4 | 7 |

Shortest Distance B/w Delhi to Haryana

#60daysofcode
Day 12/60
Lecture - 143

| M 1 Vs M 2 | Method 1 | Method 2 |
|---|---|---|
| 1 | Tc → V^2 | TC → Elogv |
| 2 | Dense Graph | Sparse Graph<br>E = V |
| 3 | Minimum used | Maximum Time Used |
| 4 | Rarely Implement in Real Life | Real Life Applications Google Maps |

Dijkstra's Not Works with -ve Weight

TC → V+(V-1) = V*V = O(V^2)
SC → O(V)

1. Select the Vertex which is not Explored yet & it's distance is minimum among all the unexplored vertices.
2. Relax the edges:
   - → Look at your all the unexplored Neighbour
   - → if(dist[node]+weight<dist[neighbour]) dist[negihbour] = dist[node] + weight

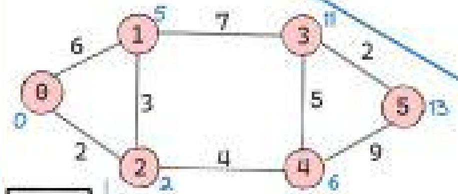
CODE PART

OPTIMIZATION
Decrease TC

Psuedo Code

```
1. Initialize Priority Queue with (0,S)
    dist[S] = 0;
pair<int,int>
2. While(priorityQueue is not empty)
    {
        Information of node
if node is explored → Ignored
else explore[node] = 1
3. Relax the edges:
   → Look at your all the unexplored
     Neighbour
   → if(dist[node]+weight<dist[neighbour])
       dist[negihbour] = dist[node] + weight
       q.push(dist, neighbour);
```

METHOD - 2



Priority Queue
Min Heap
pair<int,int>
dist
Node

| ~~(13,2)~~ |
|---|
| ~~(13,5)~~ |
| ~~(11,3)~~ |
| ~~(15,5)~~ |
| ~~(6,4)~~ |
| ~~(6,1)~~ |
| ~~(5,1)~~ |
| ~~(2,2)~~ |

Priority Queue MinHeap

Used To Select Min Element

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 0 | ∞ | ∞ | ∞ | ∞ | ∞ |
| 0 | 6 | 2 | ∞ | ∞ | ∞ |
| 0 | 5 | 2 | ∞ | 6 | ∞ |
| 0 | 5 | 2 | 12 | 6 | ∞ |
| 0 | 5 | 2 | 11 | 6 | 15 |
| 0 | 5 | 2 | 11 | 6 | 13 |

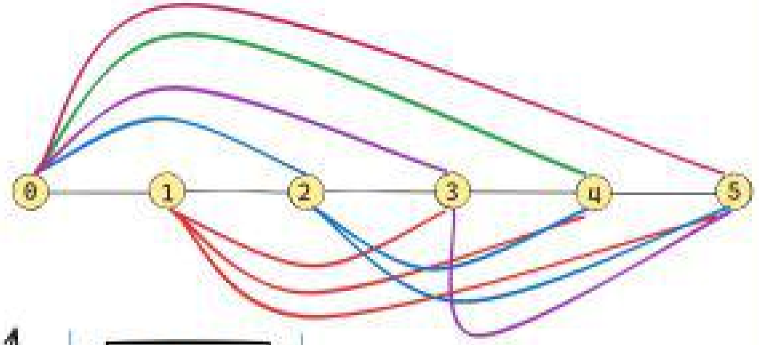
Explore

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| ~~0~~ | ~~0~~ | ~~0~~ | ~~0~~ | ~~0~~ | ~~0~~ |
| 1 | 1 | 1 | 1 | 1 | 1 |

if Complete Graph TC → Elogv

TC → ElogE (push) + ElogE(pop) = ElogE
SC → V(E) + V(D) + MaxsizeofPQ(E) = O(V+E)

Check By Complete Graph



| 5 5 5 5 5 |
|---|
| 4 4 4 4 |
| 3 3 3 |
| 2 2 |
| 1 |

Priority Queue MinHeap

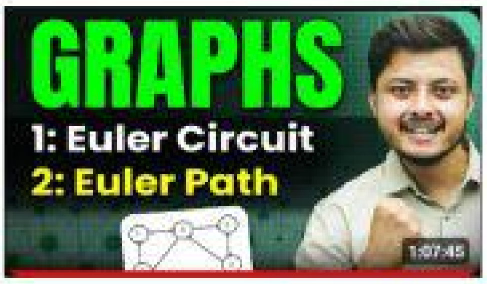
(n-2) + (n-3) + (n-4) + .......

n(c,2) = Edges

$$\boxed{\frac{(n-2)\ (n-1)}{2}} = \boxed{\frac{(n)\ (n-1)}{2}}$$

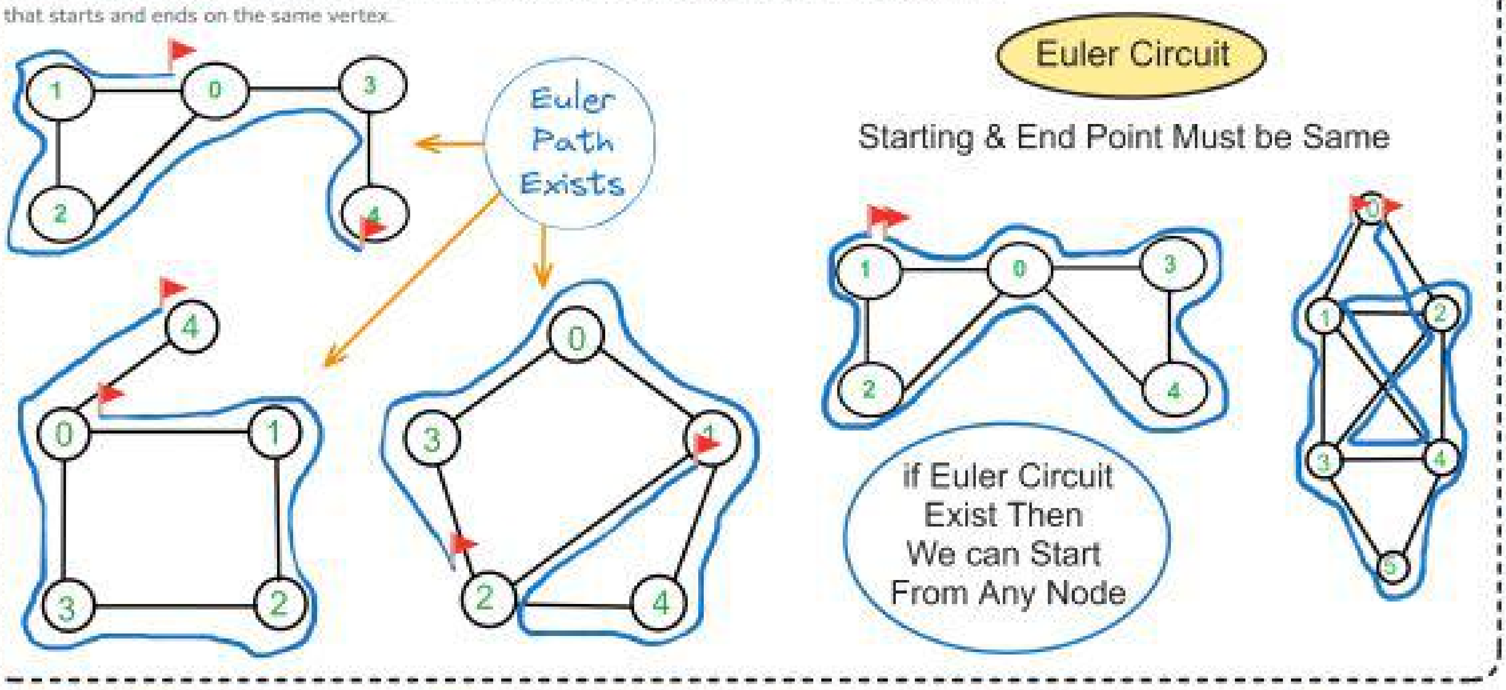
Maximum Size = Number of Edges

# #60daysofcode -> Day 17/60
# Lecture - 148



## Euler Path And Euler Circuit

Eulerian Path is a path in a graph that visits every edge exactly once. Eulerian Circuit is an Eulerian Path that starts and ends on the same vertex.

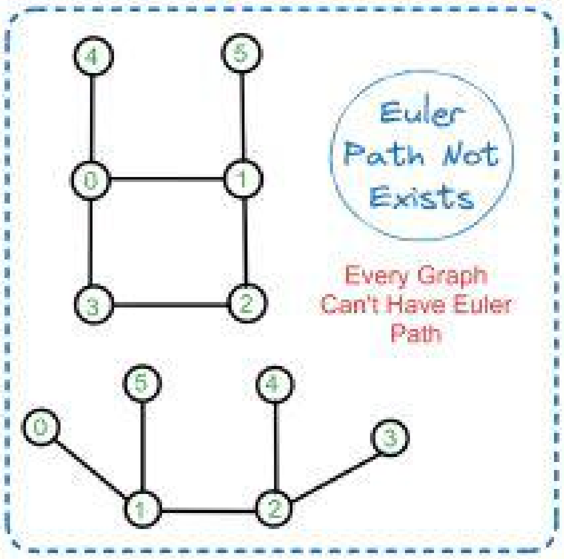


Euler Path Exists

Euler Circuit

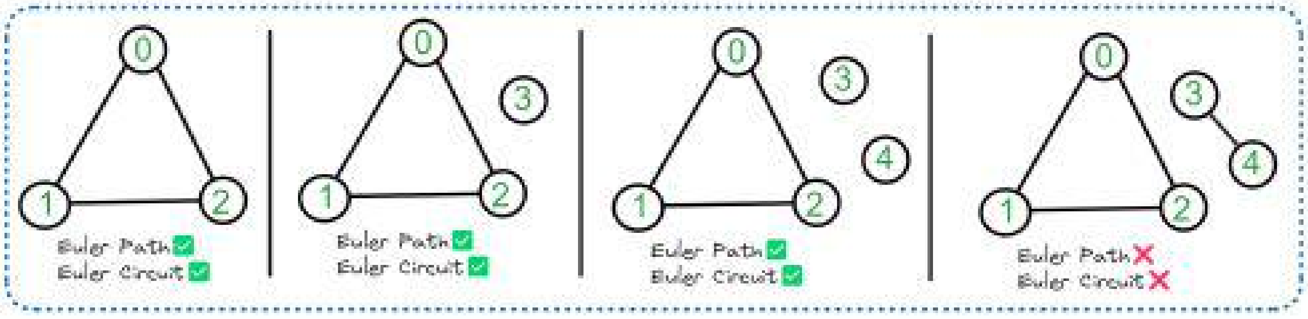
Starting & End Point Must be Same

if Euler Circuit Exist Then We can Start From Any Node



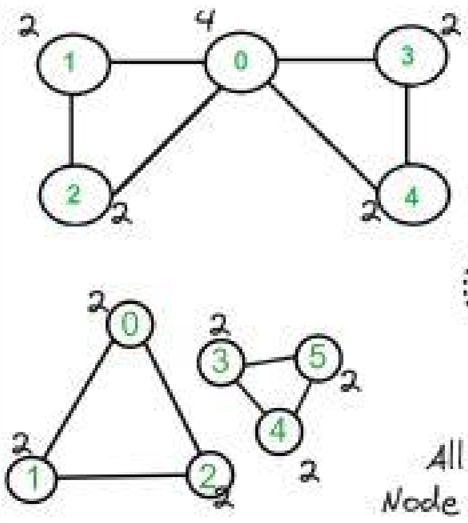
Euler Path Not Exists

Every Graph Can't Have Euler Path



## Whether Euler Circuit/Path Exist or Not

1. All Edges Should be visited exactly once,
2. start == end

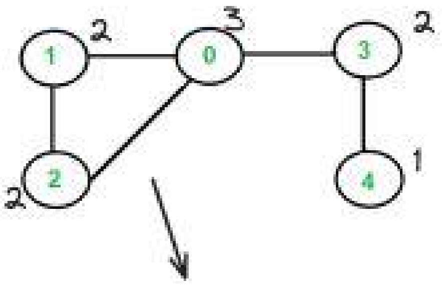


1. Every Node Degree -> Even

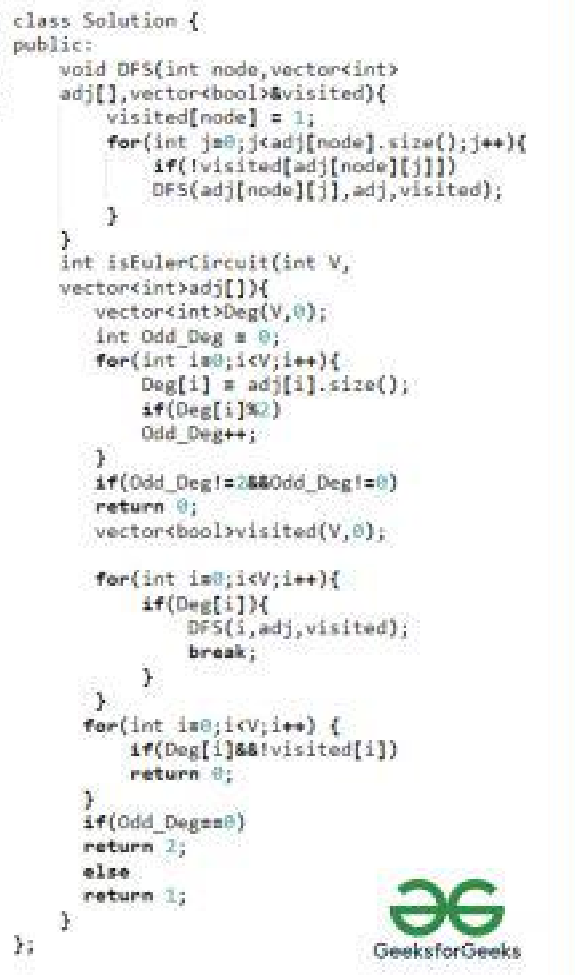
2. All Edges Should be part of 1 component

or

All Non Zero Degree Node Should be connected



1. Zero or two node can have odd Degree And Remaining Node Should Have Even Degree
2. All Non-Zero degree Node Should be Connected



```
class Solution {
public:
    void DFS(int node,vector<int>
    adj[],vector<bool>&visited){
        visited[node] = 1;
        for(int j=0;j<adj[node].size();j++){
            if(!visited[adj[node][j]])
            DFS(adj[node][j],adj,visited);
        }
    }
    int isEulerCircuit(int V,
    vector<int>adj[]){
       vector<int>Deg(V,0);
       int Odd_Deg = 0;
       for(int i=0;i<V;i++){
           Deg[i] = adj[i].size();
           if(Deg[i]%2)
           Odd_Deg++;
       }
       if(Odd_Deg!=2&&Odd_Deg!=0)
       return 0;
       vector<bool>visited(V,0);

       for(int i=0;i<V;i++){
           if(Deg[i]){
               DFS(i,adj,visited);
               break;
           }
       }
      for(int i=0;i<V;i++) {
          if(Deg[i]&&!visited[i])
          return 0;
      }
      if(Odd_Deg==0)
      return 2;
      else
      return 1;
    }
};
```

GeeksforGeeks

```
Time Complexity  : O(V+E)
Space Complexity  : O(V)
```

Complete Easy Implementation of Euler Path And Euler Circuit

## PSUEDO CODE

1. Find Degree of Each Node
2. If Deg of any Node is odd, Not a EC
3. Even Degree
4. Apply DFs, From Any Non-Zero Deg Node

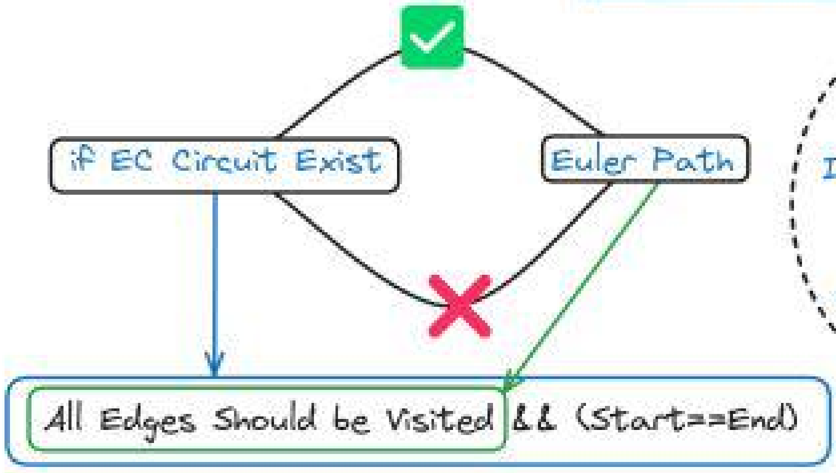
| 0 | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | Visited |

| 0 | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| 4 | 2 | 2 | 2 | 2 | 2 | Degree |

Deg Exists But Not Visited -> EC ✗

Deg Exists and Visited -> EC ✓

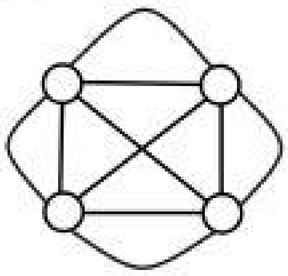
if Deg 0 -> Ignore (Don't Need to check Traverse)



if EC Circuit Exist

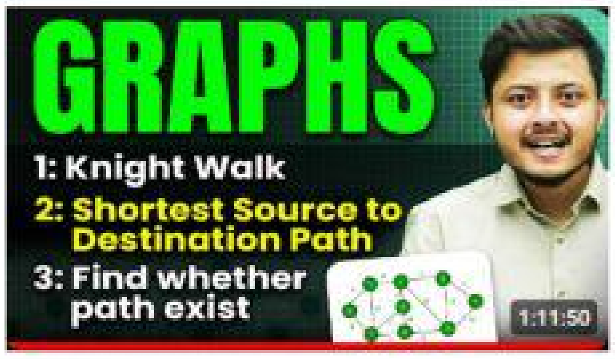
Euler Path

All Edges Should be Visited && (Start==End)



We Can't Draw This Without lifting Pen -> Because it has 4 Odd Degree Nodes (5,5,5,5) -> Eulerian Path Not Possible

# #60daysofcode -> Day 16/60
# Lecture - 147


GRAPHS

1: Knight Walk

2: Shortest Source to Destination Path

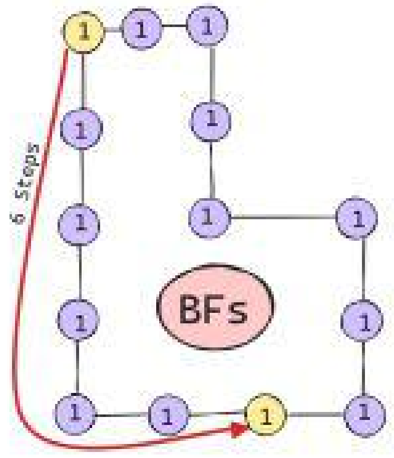
3: Find whether path exist

1:11:50

## Shortest Source to destination Path

|   | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 2 | 1 | 0 | 1 | 1 |
| 3 | 1 | 0 | 0 | 1 |
| 4 | 1 | 1 | 1 | 1 |

Binary Matrix Given



|   | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | $\cancel{0}^1$ | $\cancel{0}^1$ | $\cancel{0}^1$ | 0 |
| 1 | $\cancel{0}^1$ | 0 | $\cancel{0}^1$ | 0 |
| 2 | $\cancel{0}^1$ | 0 | $\cancel{0}^1$ | $\cancel{0}^1$ |
| 3 | $\cancel{0}^1$ | 0 | 0 | 0 |
| 4 | $\cancel{0}^1$ | $\cancel{0}^1$ | $\cancel{0}^1$ | 0 |

Visited Array
initially 0

```
pair<int,pair<int,int>>p
    (row,col,step)
```

Time Complexity : O(n*m)
Space Complexity : min(n,m)

## Edges Cases Handled, Optimized, Without Using Visited Array Complete Code :

```
class Solution {
  public:
    int row[4] = {1,-1,0,0};
    int col[4] = {0,0,1,-1};

    bool valid(int i,int j,int n,int m){
        return i≥0&&j≥0&&i<n&&j<m;}
    int shortestDistance(int N, int M,
    vector<vector<int>> A, int X, int Y) {
        //Edge Cases
        if(X==0&&Y==0)
        return 0;
        if(!A[0][0])
        return -1;

        //row,col,step
        queue<pair<int,int>>q;
        q.push({0,0});
        int step = 0;
        A[0][0] = 0;

        while(!q.empty()){
            int count = q.size();
            while(count--){
            int i = q.front().first;
            int j = q.front().second;
            q.pop();
            //up down left right
            for(int k=0;k<4;k++){
                int new_i = i+row[k];
                int new_j = j+col[k];
      if(valid(new_i,new_j,N,M)&&A[new_i][new_j])
                {
                    if(new_i==X&&new_j==Y)
                    //Check for the destination
                    return step+1;

                    A[new_i][new_j] = 0;
                    q.push({new_i,new_j});
                }}}step++;
            }
        return -1;
    }};
```

## Knight Walk

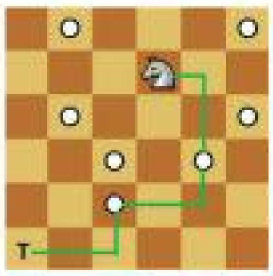
|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 |  | 🟥🐎 | 🟦🐎 | 🟥🐎 |  | 🟥🐎 | 🟦🐎 | 🟥🐎 |
| 1 | 🟥🐎 |  | 🟥🐎 |  | 🟨🐎 |  | 🟥🐎 |  |
| 2 | 🟩🐎 | 🟥🐎 | 🟦🐎 | 🟥🐎 |  | 🟥🐎 | 🟦🐎 | 🟥🐎 |
| 3 | 🟥🐎 |  |  | 🟦🐎 | 🟥🐎 | 🟦🐎 |  |  |
| 4 |  | 🟥🐎 |  | 🟥🐎 |  | 🟥🐎 |  | 🟥🐎 |
| 5 |  |  | 🟥🐎 | 🟩🐎 | 🟥🐎 |  | 🟥🐎 |  |
| 6 | 🟩🐎 |  | 🟩🐎 |  |  |  |  |  |
| 7 |  | 🟩🎯 |  |  |  |  |  |  |

N = 8

Best Method → BFs

```
int row[8] = [2,2,-2,-2,1,-1,1,-1]
int col[8] = [1,-1,1,-1,2,2,-2,-2]
```

- 🟡 Yellow → source
- 🔵 Blue→ 1 Step
- 🔴 Red→ 2 Step
- 🟢 Green→ 3 Step → Answer



Complete Optimized Edge Cases Handled Code in Gfg

## Find Whether Path Exist

|   | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 3 | 0 | 3 | 3 | 0 |
| 1 | 3 | 0 | 2 | 0 | 3 |
| 2 | 3 | 3 | 3 | 3 | 3 |
| 3 | 0 | 3 | 0 | 0 | 3 |
| 4 | 0 | 3 | 1 | 0 | 3 |

1 → Source
2 → Destination
3 → Blank
0 → Wall

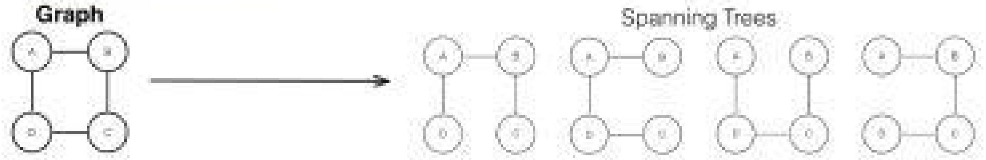
YES OR NO
BFs ✅ → Homework
DFs ✅ → Homework

Traversed → Make 0

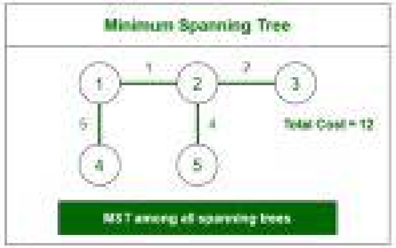
#60daysofcode -> Day 19/60
Lecture - 150

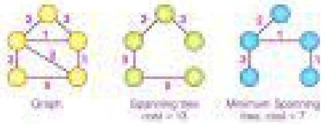
# Minimum Spanning Tree | Prim's Algorithm

*A spanning tree is a subset of Graph G, such that all the vertices are connected using minimum possible number of edges. Hence, a spanning tree does not have cycles and a graph may have more than one spanning tree.*

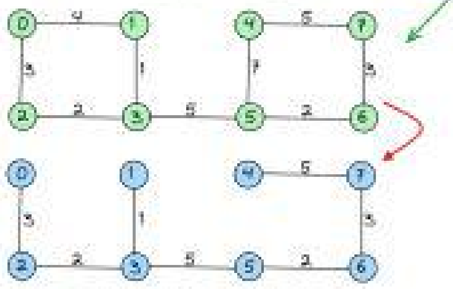


*A **minimum spanning tree (MST)** is defined as a spanning tree that has the minimum weight among all the possible spanning trees.*

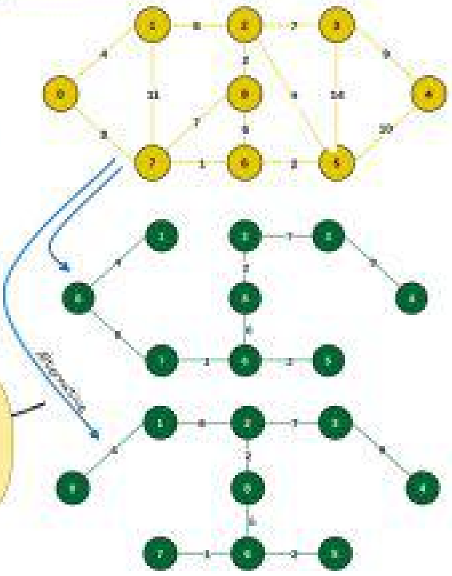




Prim's Algorithm
Or
Greedy Algorithm



Minimum Cost : 3+2+1+5+2+3+5 ⇒ 21



Min Heap Priority Queue

(Weight, node, parent)
Edge

| IsMST | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| initially 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

| Parent | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| | -1 | 0 | 1 | 2 | 3 | 6 | 8 | 6 | 2 |

Cost = 0 2 4 10 14 22 24 28 30 39
Answer

Draw MST = ?

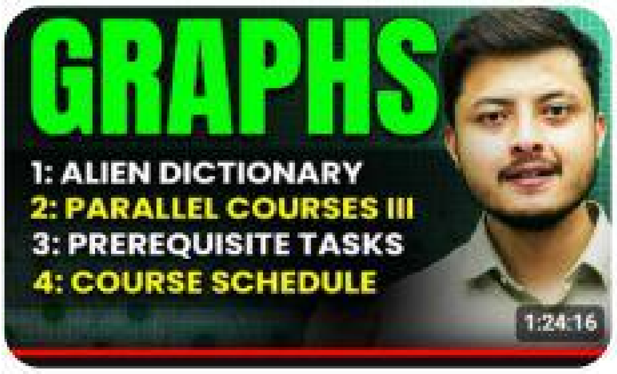
COMPLETE CODE

```cpp
class Solution{
 public:
 //Function to find sum of weights of edges of the Minimum Spanning Tree.
    int spanningTree(int V, vector<vector<int>> adj[]){
        priority_queue<pair<int,pair<int,int>>,
        vector<pair<int,pair<int,int>>>,
        greater<pair<int,pair<int,int>>>>pq;
        vector<bool>IsMST(V,0);

        vector<int>parent(V); //Temporary
        int cost = 0;
        pq.push({0,{0,-1}});

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

            if(!IsMST[node]){
                IsMST[node] = 1;
                cost += wt;
                parent[node] = pqr;

                for(int j=0;j<adj[node].size();j++){
                    if(!IsMST[adj[node][j][0]])
                    {
                        pq.push({adj[node][j][1], {adj[node][j][0], node}});
                    }}}}
          return cost;}};
```

Time Complexity : E*logE+ElogE
⇒ ElogE / ElogV
Space Complexity : V+V+E ⇒ V+E

1. Prerequisite Tasks
2. Course Schedule
3. Parallel Courses 3
4. Alien Dictionary

#60daysofcode
Day 9/60
Lecture - 140



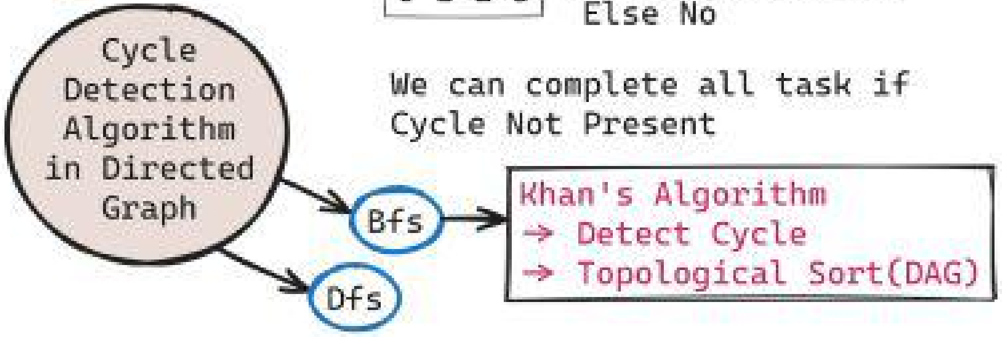
## 1 → Prerequisite Tasks

N = 4
P = 3

Pre = {{1,0}, {2,1}, {3,2}}

0 1 2 3    Ans Possible Yes
Else No

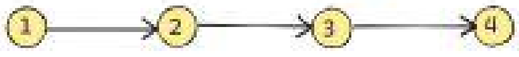
We can complete all task if
Cycle Not Present

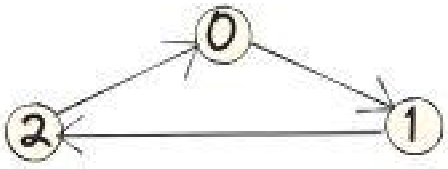

Cycle Detection Algorithm in Directed Graph

Bfs

Dfs

Khan's Algorithm
→ Detect Cycle
→ Topological Sort(DAG)

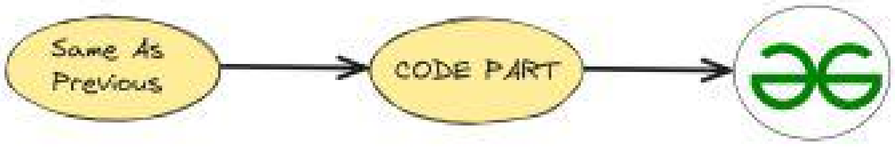




| 0 | 1 | 2 |
|---|---|---|
| 1 | 1 | 1 |

Size = 0 ≠ 3
All Task Not Completed

## 2 → Course Schedule


Same As Previous → CODE PART →

## 3 → Alien Dictionary

Given
K = 4

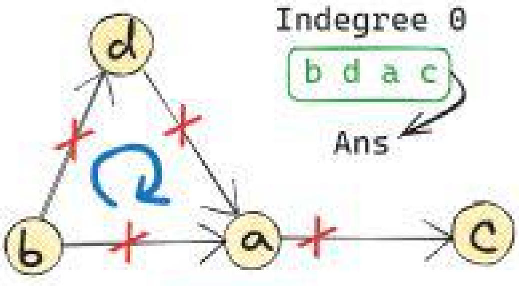
| Alien | Human |
|---|---|
| b a a | a b c a |
| a b c d | a b c d |
| a b c a | b a a |
| c a b | c a b |
| c a d | c a d |

Order of Character ?
b → a
d → a
a → c
b → d

Topological Sort

Order → a b c d..

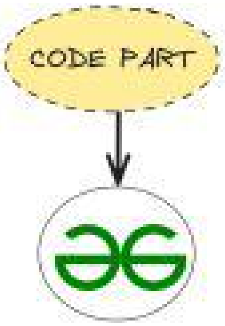
1. Adjacency List Create
2. Khan's Algorithm



Indegree 0

b d a c

Ans


CODE PART

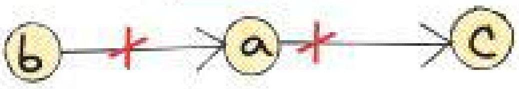
Alien Dict.

c a a
a a a
a a b

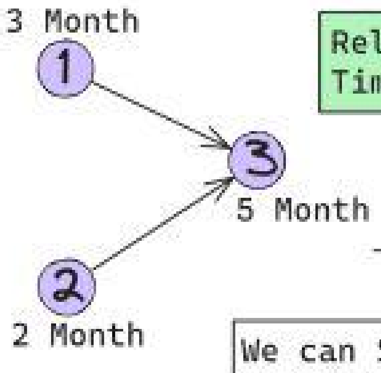
Given
K = 3

Ans

b a c



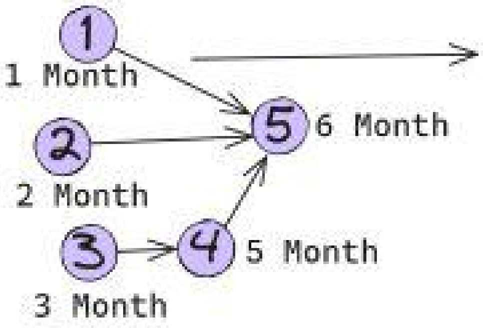
## 4 → Parallel Courses 3



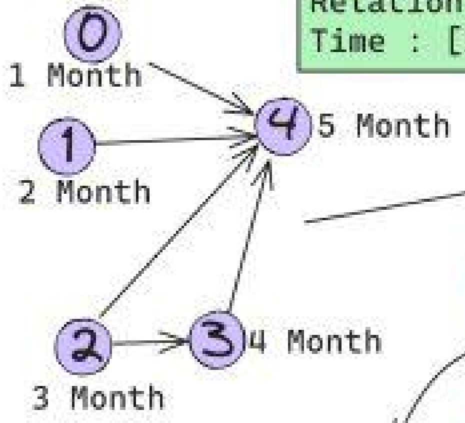
Relation : [[1,3][2,3]]
Time : [3,2,5]

→ 10 Month Required

We can Start
More than One
Course Together



→ 14 Month Required



Relation : [[1,5],[2,5],[3,5],[3,4],[4,5]]
Time : [1,2,3,4,5]

→ 12 Month Required

Queue

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

Topological Sort

Khan's Algorithm
→ Detect Cycle
→ Topological Sort(DAG)

Maximum Time To Complete Previous courses = Course Time + Present Time

Indegree

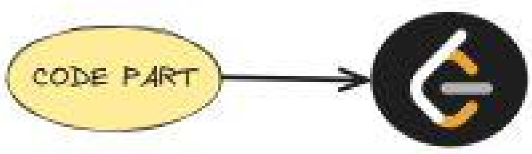
| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 0 | 0 | 0 | 1 | ~~4~~ |
| 0 | 0 | 0 | 1 | ~~3~~ |
| 0 | 0 | 0 | ~~1~~ | ~~2~~ |
| 0 | 0 | 0 | 0 | ~~1~~ |
| 0 | 0 | 0 | 0 | 0 |

Course Time

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | ~~0~~ |
| 0 | 0 | 0 | 0 | ~~1~~ |
| 0 | 0 | 0 | ~~0~~ | ~~2~~ |
| 0 | 0 | 0 | 3 | ~~3~~ |
| 0 | 0 | 0 | 3 | 7 |

| + 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| 1 | 2 | 3 | 7 | 12 |

12 Month


CODE PART →

1. Covid Spread
2. find the number of Islands
3. Replace O's with X's

#60daysofcode
Day 8
Lecture - 139

## Covid Spread

Hospital Rooms

|   | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | $1^2$ | 2 | $1^2$ | 1 | 0 | $1^2$ |
| 1 | $1^2$ | $1^2$ | 0 | $1^2$ | $1^2$ | 2 |
| 2 | 0 | $1^2$ | 0 | 2 | $1^2$ | $1^2$ |
| 3 | $1^2$ | $1^2$ | 0 | $1^2$ | 0 | $1^2$ |
| 4 | $1^2$ | 0 | $1^2$ | 2 | 0 | $1^2$ |

0 → Empty
1 → Normal Patient
2 → Covid Patient

5 Unit of Time - Spreaded

BFS

|   | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 2 | $^{2}1$ | 0 | 2 | $^{2}1$ |
| 1 | $_{2}1$ | 0 | $1^2$ | 2 | $^{2}1$ |
| 2 | $_{2}1$ | 0 | 0 | 2 | $_{2}1$ |

Timer = ~~1~~ ~~2~~ 3

3 - 1 = 2 Ans

TC - O(V+E)
SC - O(V)

Queue ~~(0,0)~~ ~~(0,3)~~ ~~(1,3)~~ ~~(2,3)~~

Queue ~~(0,1)~~ ~~(1,0)~~ ~~(0,4)~~ ~~(1,2)~~ ~~(1,4)~~ ~~(2,4)~~

Queue ~~(2,0)~~

CODE PART

## Find the Number of Island

|   | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 0 | 1 | 1 |
| 1 | 0 | 1 | 1 | 0 | 0 | 1 |
| 2 | 1 | 0 | 0 | 0 | 0 | 0 |
| 3 | 1 | 0 | 1 | 0 | 1 | 1 |
| 4 | 1 | 1 | 0 | 0 | 1 | 1 |

0 → Water
1 → land

3 Island Ans

BFS

Array for Visited

|   | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 0 | 1 | 1 |
| 1 | 0 | 1 | 1 | 0 | 0 | 1 |
| 2 | 1 | 0 | 0 | 0 | 0 | 0 |
| 3 | 1 | 0 | 1 | 0 | 1 | 1 |
| 4 | 1 | 1 | 0 | 0 | 1 | 1 |

Queue ~~(0,0)~~ ~~(0,1)~~ ~~(1,1)~~ ~~(0,2)~~ ~~(1,2)~~ ~~(2,0)~~ ~~(3,0)~~ ~~(4,0)~~ ~~(4,1)~~

Queue ~~(0,4)~~ ~~(0,5)~~ ~~(1,5)~~

Queue ~~(3,4)~~ ~~(3,5)~~ ~~(4,4)~~ ~~(4,5)~~

|   | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | ~~1~~0 | ~~1~~0 | ~~1~~0 | 0 | ~~1~~0 | ~~1~~0 |
| 1 | 0 | ~~1~~0 | ~~1~~0 | 0 | 0 | ~~1~~0 |
| 2 | ~~1~~0 | 0 | 0 | 0 | 0 | 0 |
| 3 | ~~1~~0 | 0 | ~~1~~0 | 0 | ~~1~~0 | ~~1~~0 |
| 4 | ~~1~~0 | ~~1~~0 | 0 | 0 | ~~1~~0 | ~~1~~0 |

TC - O(V+E)
SC - O(V)

CODE PART

Solution Without Using Without Visited Array

1. Find 1
→ Count OF Island Increase
→ push in Queue (Row & Col)
→ Adjacent 1, Make them Zero

Queue ~~(0,0)~~ ~~(0,1)~~ ~~(1,1)~~ ~~(0,2)~~ ~~(1,2)~~ ~~(2,0)~~ ~~(3,0)~~ ~~(4,0)~~ ~~(4,1)~~

Queue ~~(0,4)~~ ~~(0,5)~~ ~~(1,5)~~

Queue ~~(3,4)~~ ~~(3,5)~~ ~~(4,4)~~ ~~(4,5)~~

## Replace O's With X's

| | | | |
|---|---|---|---|
| X | X | X | X |
| X | O | X | X |
| X | O | O | X |
| X | O | X | X |
| X | X | O | O |

Set of O's Surrounded by X Replaced with X

Set m ek bhi O's boundary pe hoga, to woh kabhi bhi replace nahi honge X se

| | | | |
|---|---|---|---|
| X | O | X | X |
| X | O | X | X |
| X | O | O | X |
| X | O | X | X |
| X | X | O | O |

Boundary O's can't be replaced With X

1. Traverse Starts from Boundary
→ If O's Find - Put in Array That not need to change
→ Apply BF's

|   | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 |  | 1 |  |  |
| 1 |  | 1 |  |  |
| 2 |  | 1 | 1 |  |
| 3 |  | 1 |  |  |
| 4 |  |  | 1 | 1 |

Boundary O's → 1

TC - O(V+E)
SC - O(V)

CODE PART

Without Using Extra Array

| | | | | | | |
|---|---|---|---|---|---|---|
| X | X | X | ~~O~~T | X | X | X |
| ~~O~~T | ~~O~~T | ~~O~~T | X | O | O | X |
| X | X | ~~O~~T | X | O | O | X |
| X | X | ~~O~~T | X | O | X | X |
| X | ~~O~~T | X | X | O | X | ~~O~~T |
| ~~O~~T | ~~O~~T | ~~O~~T | X | X | X | ~~O~~T |

Boundary O's → T
Other O's → X

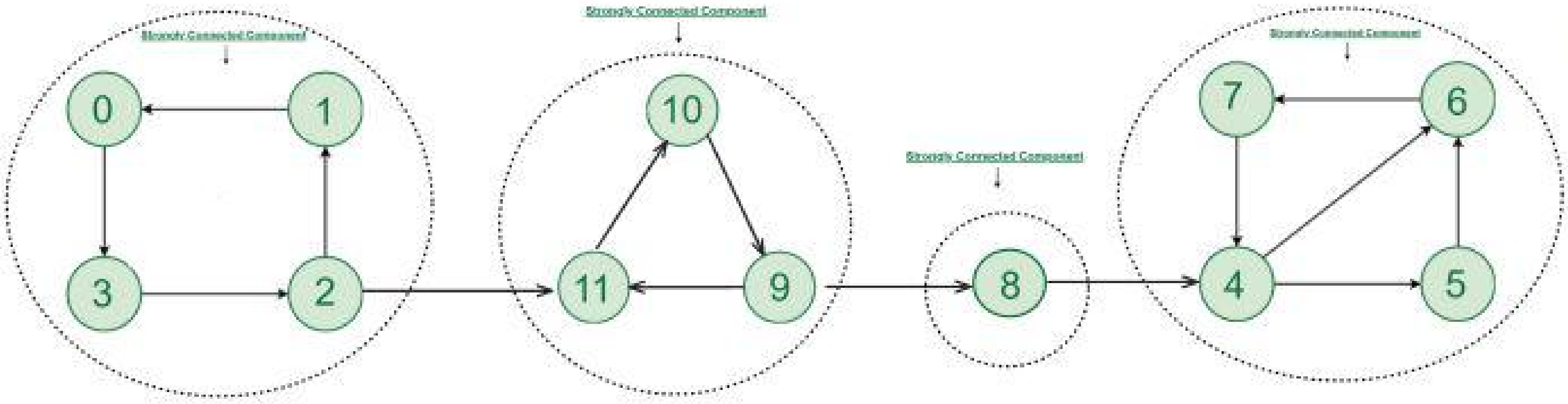
| | | | | | | |
|---|---|---|---|---|---|---|
| X | X | X | O | X | X | X |
| O | O | O | X | O | O | X |
| X | X | O | X | O | O | X |
| X | X | O | X | O | X | X |
| X | O | X | X | O | X | O |
| O | O | O | X | X | X | O |

T → O
Other O's → X

#60daysofcode -> Day 21/60
Lecture - 152

# Kosaraju's Algorithm
# Strongly Connected Components

In a directed graph, a Strongly Connected Component is subset of vertices where every vertex in the subset is reachable from every other vertex in the same subset by traversing the directed edges.

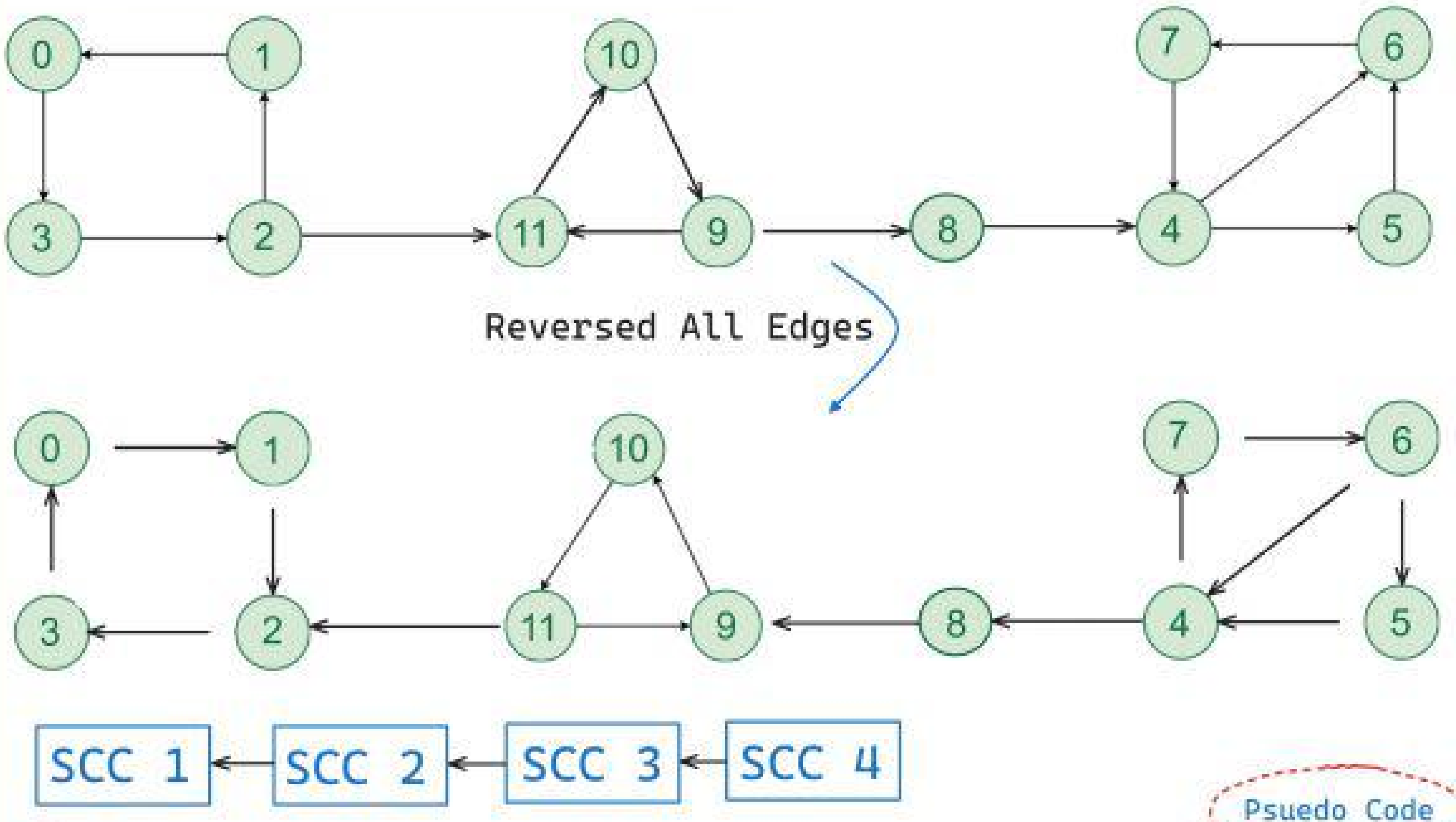

Strongly Connected Component
Strongly Connected Component
Strongly Connected Component
Strongly Connected Component
0
1
3
2
10
11
9
8
7
6
4
5

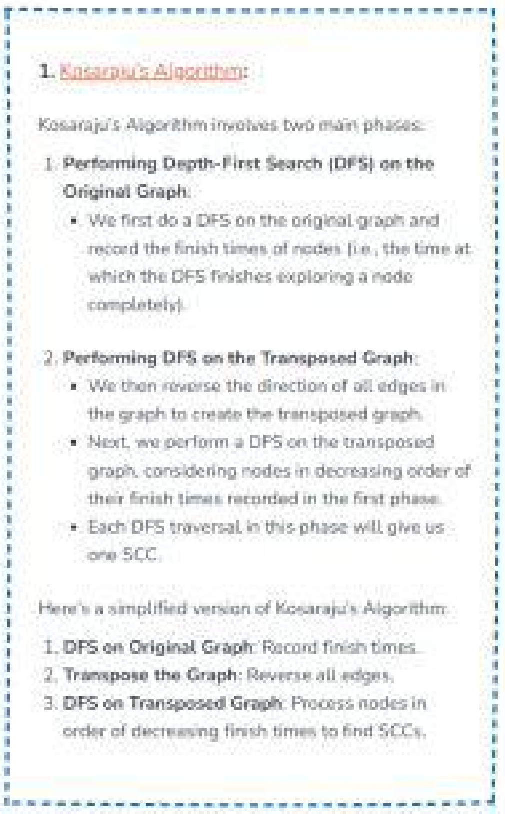
Answers

```
1. SCC → 4
2. 0 1 2 3
   10 11 9
   8
   5 4 6 7
```

How To Solve ?

## Kosaraju's Algorithm





1. Kosaraju's Algorithm:

Kosaraju's Algorithm involves two main phases:

1. **Performing Depth-First Search (DFS) on the Original Graph**:
   - We first do a DFS on the original graph and record the finish times of nodes (i.e., the time at which the DFS finishes exploring a node completely).
2. **Performing DFS on the Transposed Graph**:
   - We then reverse the direction of all edges in the graph to create the transposed graph.
   - Next, we perform a DFS on the transposed graph, considering nodes in decreasing order of their finish times recorded in the first phase.
   - Each DFS traversal in this phase will give us one SCC.

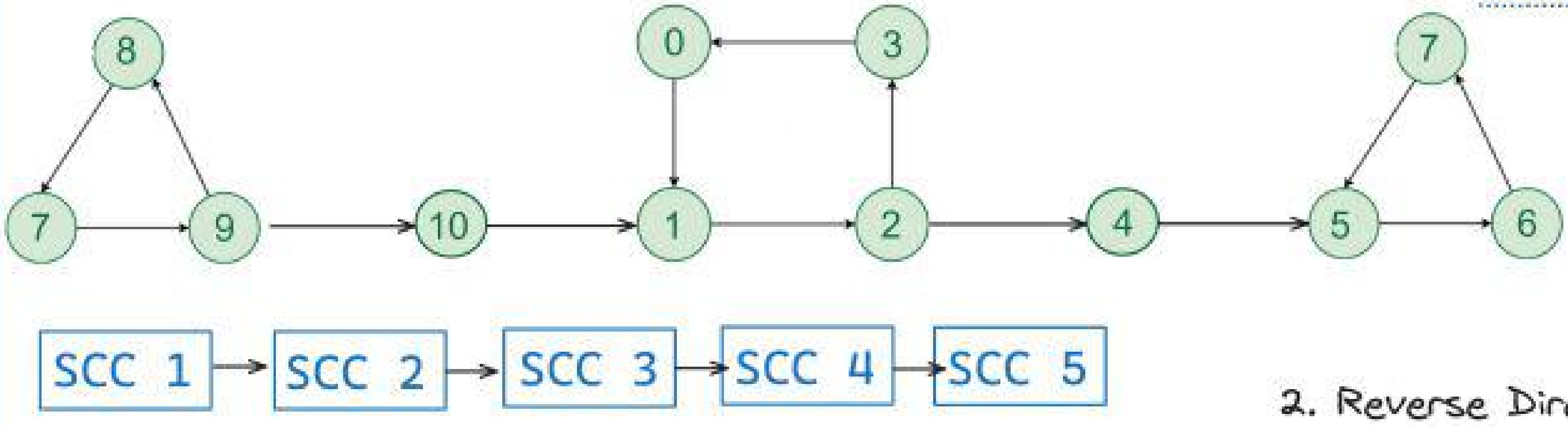
Here's a simplified version of Kosaraju's Algorithm:

1. **DFS on Original Graph**: Record finish times.
2. **Transpose the Graph:** Reverse all edges.
3. **DFS on Transposed Graph**: Process nodes in order of decreasing finish times to find SCCs.

Psuedo Code

```
1. Topological Sort
2. Reverse the Edge
3. Pop Element from Stack
One by One
if Node is univisited
    → Call the DFs
    → SCC++
```

Eg.2



Complete Code

GeeksforGeeks

TC -> V+E
SC -> V+E

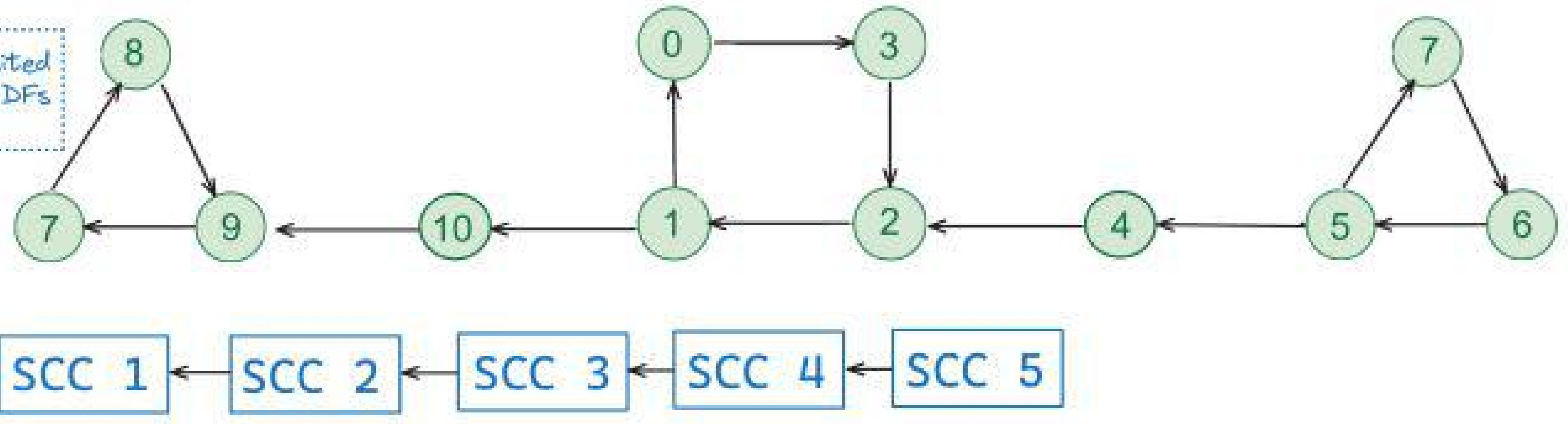
2. Reverse Direction

1. Topological Sort

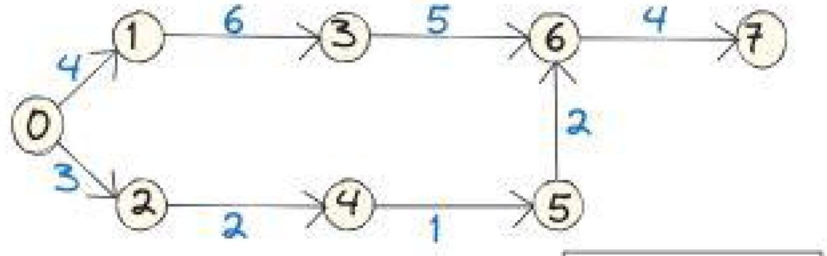
| | |
|---|---|
| 8 | 2 |
| 7 | 4 |
| 9 | 5 |
| 10 | 6 |
| 0 | 7 |
| 1 | 3 |

Stack

```
if Node is univisited
    -> Call the DFs
    -> SCC++
```



3. Pop Elements



Source Node → 0

Adjacency List

```
0 → {1,4},{2,3}
1 → {3,6}
2 → {4,2}
3 → {6,5}
4 → {5,1}
5 → {3,2}
6 → {7,4}
7 →
```

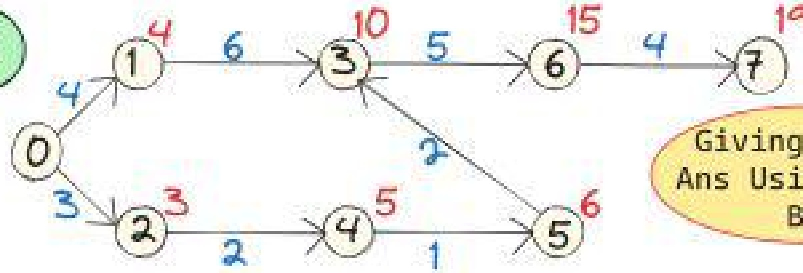
#60daysofcode
Day 11/60
Lecture - 142

Distance

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 0 | 4 | 3 | 8 | 5 | 6 | 13 | 17 |

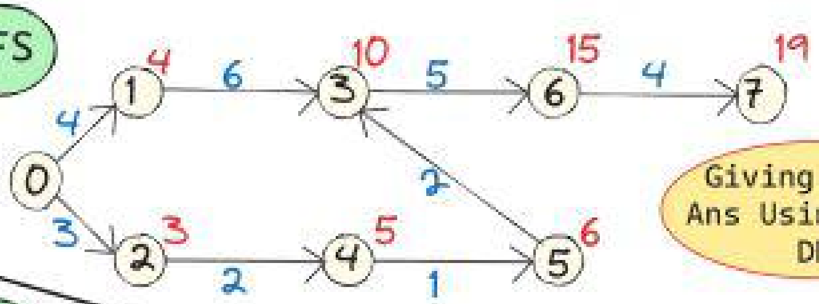
→ Required Array

BFS



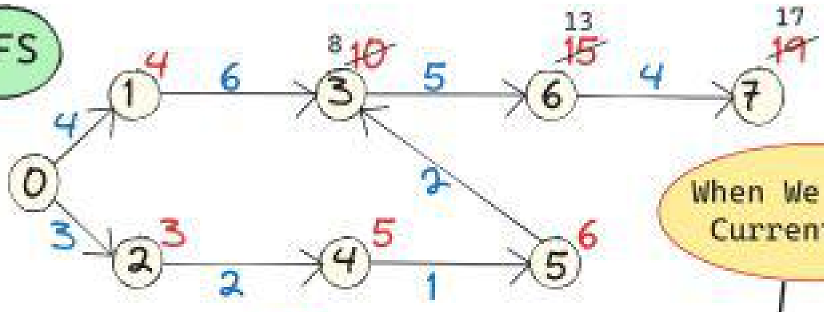
Giving Wrong Ans Using Only BFs

DFS



Giving Wrong Ans Using Only DFs

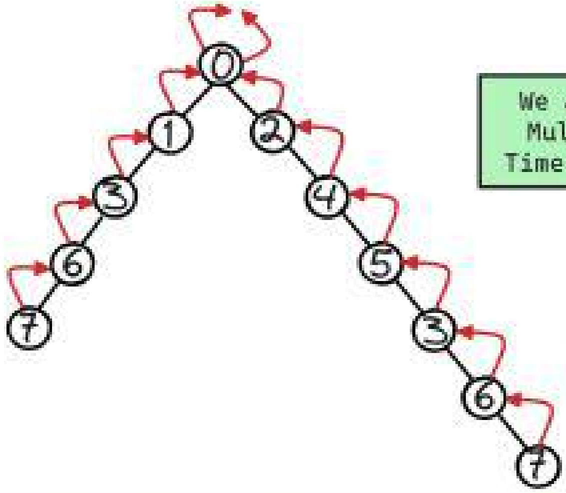
DFS



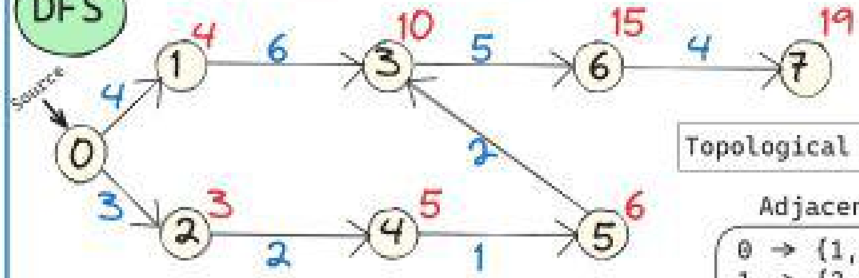
When We follow Current Path

We are Exploring a Path Multiple Times So that Time Complexity Very High

Decrease Time Complexity



DFS



Topological Sort → Khan's, DFs ✓

Adjacency List

```
0 → {1,4},{2,3}
1 → {3,6}
2 → {4,2}
3 → {6,5}
4 → {5,1}
5 → {3,2}
6 → {7,4}
7 →
```

Stack

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

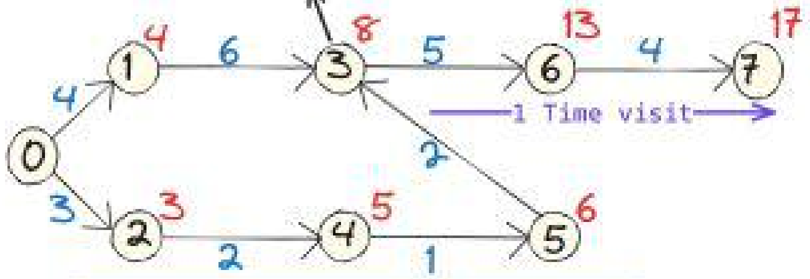
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | |
|---|---|---|---|---|---|---|---|---|
| ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | Dist |
| 0 | 4 | 3 | ∞ | ∞ | ∞ | ∞ | ∞ | Dist |
| 0 | 4 | 3 | ∞ | 5 | ∞ | ∞ | ∞ | Dist |
| 0 | 4 | 3 | ∞ | 5 | 6 | ∞ | ∞ | Dist |
| 0 | 4 | 3 | 8 | 5 | 6 | ∞ | ∞ | Dist |
| 0 | 4 | 3 | 8 | 5 | 6 | 13 | ∞ | Dist |
| 0 | 4 | 3 | 8 | 5 | 6 | 13 | 17 | Dist |

TC → O(V+E)
SC → O(V+E)

2 Step Solution
1. Find Topological Sort
2. Stack to empty karo ek ek karke

```
dist[neighbour] = min(dist[neighbour], weight(node,neighbour) + dist[node])
```

Minimum Distance from source Node



```
dist[3] = min(6+dist[1], 2+dist[5])
```

Topological Sort

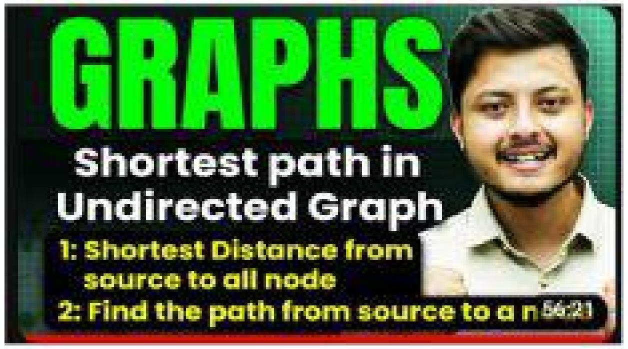
| 0 | 2 | 4 | 1 | 5 | 3 | 6 | 7 |
|---|---|---|---|---|---|---|---|

Detailed Solution

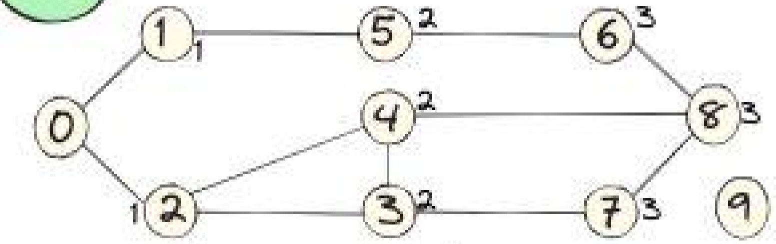
CODE PART → GfG

# SHORTEST PATH IN AN UNDIRECTED GRAPH

#60daysofcode
Day 10/60
Lecture - 141



## BFS



Adjacency List

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

Source Node → 0

TC → O(V+E)
SC → O(V+E)

unconnected Node = -1

Queue 0 1 2 5 4 3 6 8 7

CODE PART →

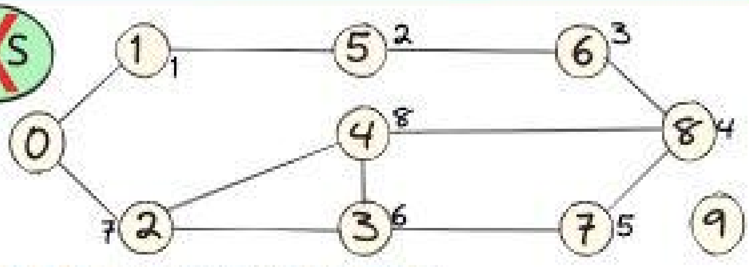
Distance

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
| 0 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
| 0 | 1 | 1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
| 0 | 1 | 1 | -1 | -1 | 2 | -1 | -1 | -1 | -1 |
| 0 | 1 | 1 | 2 | 2 | 2 | -1 | -1 | -1 | -1 |
| 0 | 1 | 1 | 2 | 2 | 2 | 3 | -1 | -1 | -1 |
| 0 | 1 | 1 | 2 | 2 | 2 | 3 | -1 | 3 | -1 |
| 0 | 1 | 1 | 2 | 2 | 2 | 3 | 3 | 3 | -1 |

Visited

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |

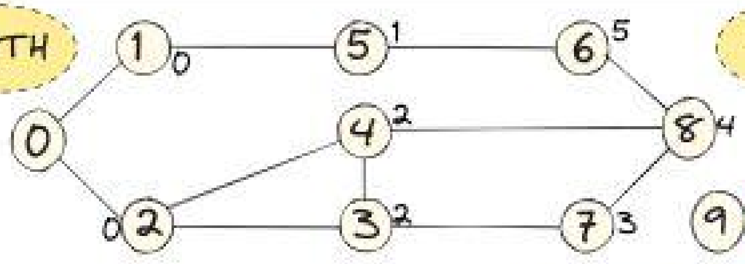
## ~~DFS~~



Simple DFS - Gives Wrong Answer

If Apply Time Complexity Increases.
That's Why We Don't Use DFS

## PATH



CODE PART

Parent Information

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | 0 | 0 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | 0 | 0 | -1 | -1 | 1 | -1 | -1 | -1 | -1 |
| -1 | 0 | 0 | 2 | 2 | 1 | -1 | -1 | -1 | -1 |
| -1 | 0 | 0 | 2 | 2 | 1 | 5 | -1 | -1 | -1 |
| -1 | 0 | 0 | 2 | 2 | 1 | 5 | -1 | 4 | -1 |
| -1 | 0 | 0 | 2 | 2 | 1 | 5 | 3 | 4 | -1 |

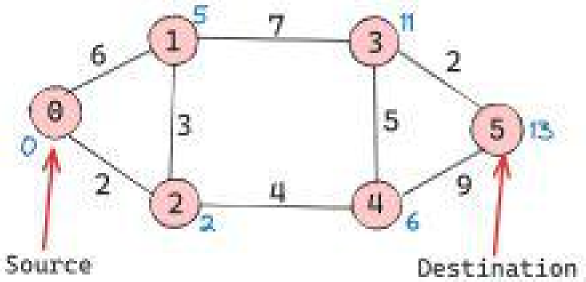
8 4 2 0
0 2 4 8 → Answer

Queue 0 1 2 5 4 3 6 8 7

Visited

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |

# Shortest Path in Weighted Undirected Graph



Required Answer **0 2 4 3 5**

## Applying Dijkstra

Tc - ElogV + V
Sc - E+V

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 0 | ∞ | ∞ | ∞ | ∞ | ∞ |
| 0 | 6 | 2 | ∞ | ∞ | ∞ |
| 0 | 5 | 2 | ∞ | 6 | ∞ |
| 0 | 5 | 2 | 12 | 6 | ∞ |
| 0 | 5 | 2 | 11 | 6 | 15 |
| 0 | 5 | 2 | 11 | 6 | 13 |

Explore

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| ~~0~~ | ~~0~~ | ~~0~~ | ~~0~~ | ~~0~~ | ~~0~~ |

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 |

Parent

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| ~~-1~~ | ~~-1~~ | ~~-1~~ | ~~-1~~ | ~~-1~~ | ~~-1~~ |

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| -1 | 0 | 0 | -1 | 2 | -1 |

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| -1 | 2 | 0 | 1 | 2 | -1 |

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| -1 | 2 | 0 | 4 | 2 | 4 |

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| -1 | 2 | 0 | 4 | 2 | 3 |

Dest = 5;

```
Vector<int>path
while(dest≠0){
path.push_back(dest)
dest = parent[dest];
}
```

Path : 5 4 3 2 0

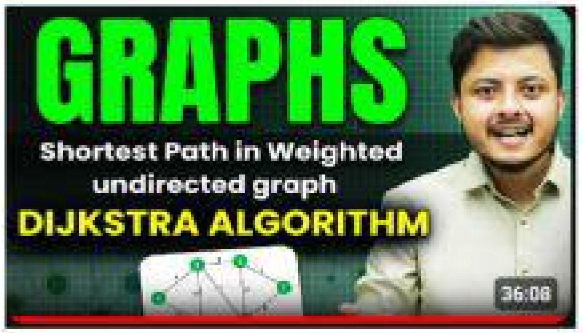
Reverse

Required Answer **0 2 4 3 5**

## Understanding With Real World Example

source
destination
the shortest path between the source and destination
a subpath which is also the shortest path between its source and destination

#60daysofcode
Day 13/60
Lecture - 144



## Complete Code

```
vector<int> shortestPath(int V, int m, vector<vector<int>>& edges) {
        // adjacency list create
        // neighboiur, weight
        vector<pair<int,int>>adj[V+1];
        for(int i=0;i<m;i++){
            int u = edges[i][0];
            int v = edges[i][1];
            int weight = edges[i][2];
            adj[u].push_back({v,weight});
            adj[v].push_back({u,weight});
        }

        // Dijkstra Algorithm
        vector<bool>Explored(V+1,0);
        vector<int>dist(V+1,INT_MAX);
        vector<int>parent(V+1,-1);
        priority_queue< pair<int,int>,vector< pair<int,int>>,greater< pair<int,int>>>p;
        p.push({0,1});
        dist[1]=0;
        while(!p.empty()){
            int node = p.top().second;
            p.pop();
            if(Explored[node])
            continue;
            Explored[node] = 1;
            for(int j=0;j<adj[node].size();j++){
                int neighbour = adj[node][j].first;
                int weight = adj[node][j].second;
                if(!Explored[neighbour]&&dist[node]+weight<dist[neighbour]){
                    dist[neighbour] = dist[node]+weight;
                    p.push({dist[neighbour],neighbour});
                    parent[neighbour] = node; // line added } } }
        vector<int>path;
         // I can't reach my destination
        if(parent[V]==-1){
            path.push_back(-1);
            return path;
        }
        // I will reach my destination
        int dest = V;
        while(dest!=-1){
            path.push_back(dest);
            dest = parent[dest];}

        path.push_back(dist[V]);
        reverse(path.begin(),path.end());
        return path;}
```