

Graph - 2

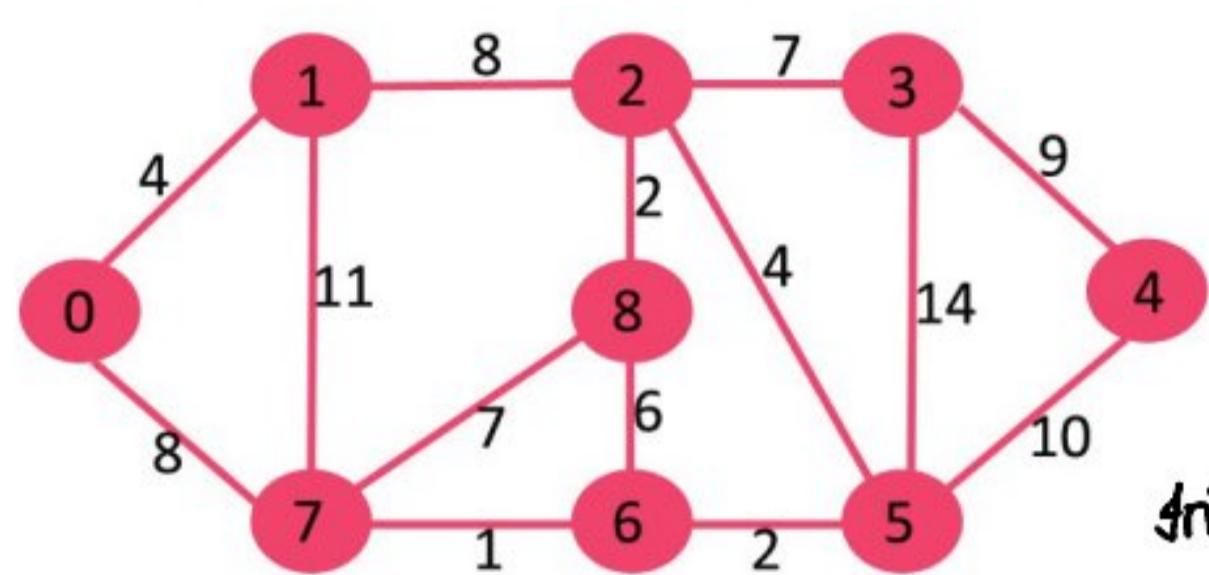
- Karun Karthik

Content

11. Dijkstra Algorithm
12. Network Delay Time
13. Bellman Ford Algorithm
14. Negative Weight Cycle
15. Floyd Warshall Algorithm
16. Prim's Algorithm
17. Min Cost to Connect All Points
18. Is Graph Bipartite ?
19. Possible Bipartition
20. Disjoint Set
21. Kruskal's Algorithm
22. Critical Connection in a Network

11 Dijkstra Algorithm → single source shortest path (only +ve weights)

→ Helps in finding the shortest path to every node from src node.



$n = 9$ (nodes from 0 to 8)

$\text{src} = 1$

$\text{cost array} = \min \text{cost from src to every other vertex}$

Initially $\text{cost} =$

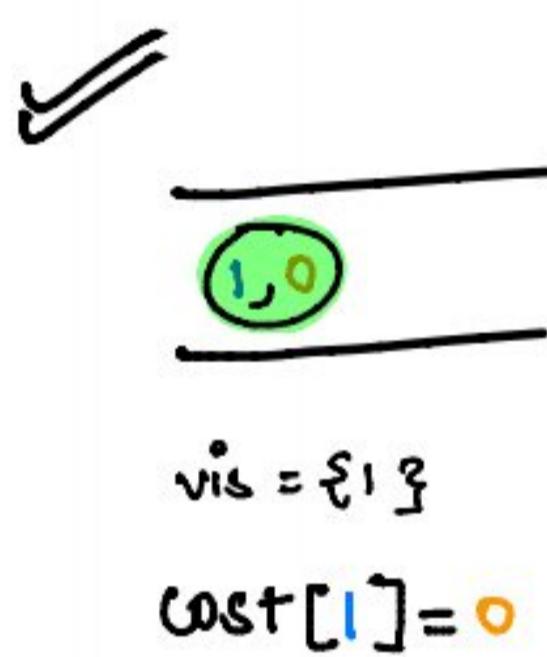
0	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8

$\text{vis} = \{1\}$

→ As it is weighted graph, we'll use priority queue (pq) instead of normal queue. q element pushed into it will be of form

curr node, curr cost

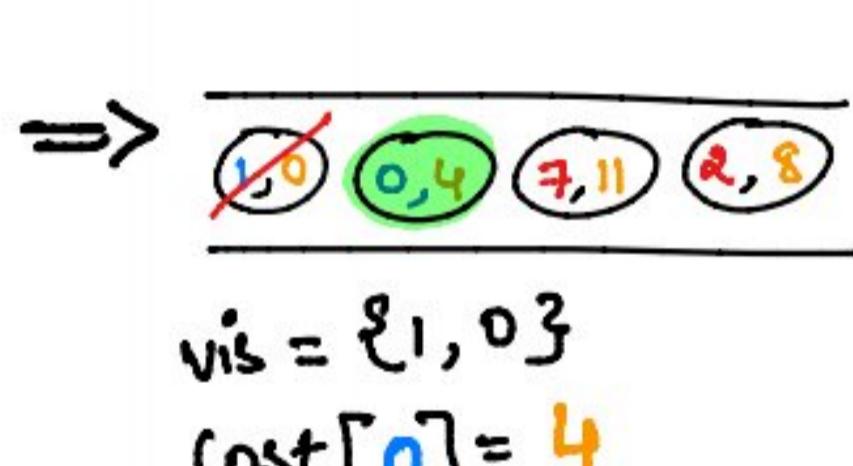
→ pq always pops element with least curr cost. always calculated from src to curr node.



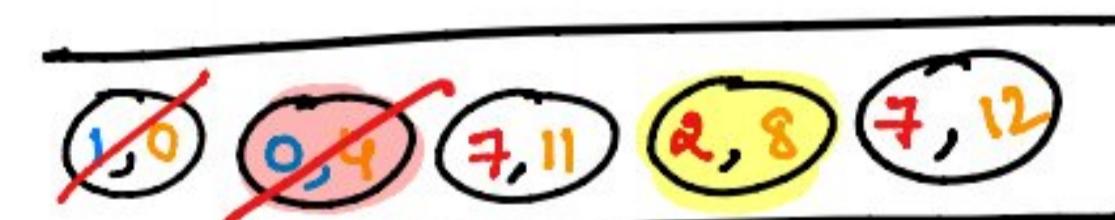
⇒ now neighbours of 1 = $(0, 4)$, $(7, 11)$, $(2, 8)$ ∴ push



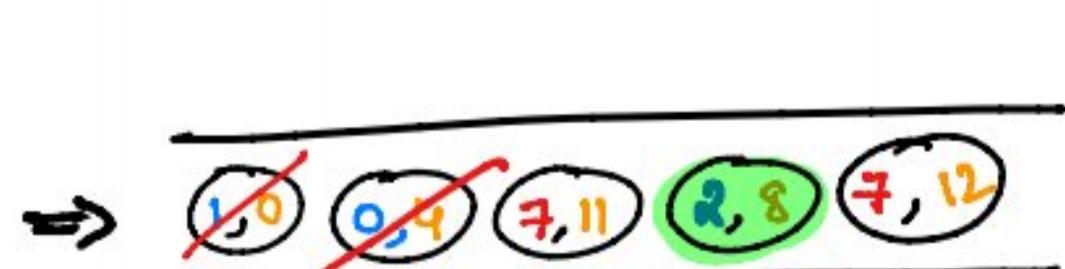
→ lowest cost among $4, 11, 8$ is 4 ∴ pop it & push its neighbours.



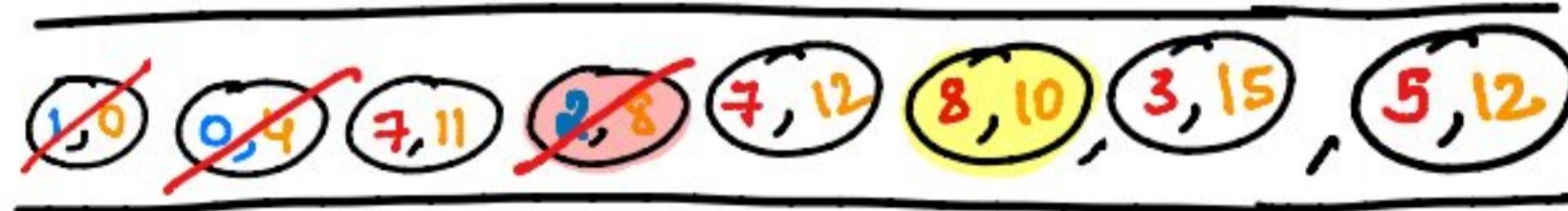
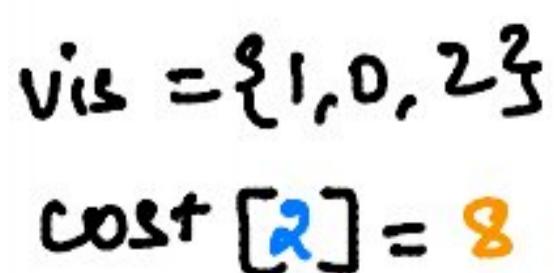
⇒ now neighbours of 0 = 1 (visited), $(7, 12)$ ∴ push



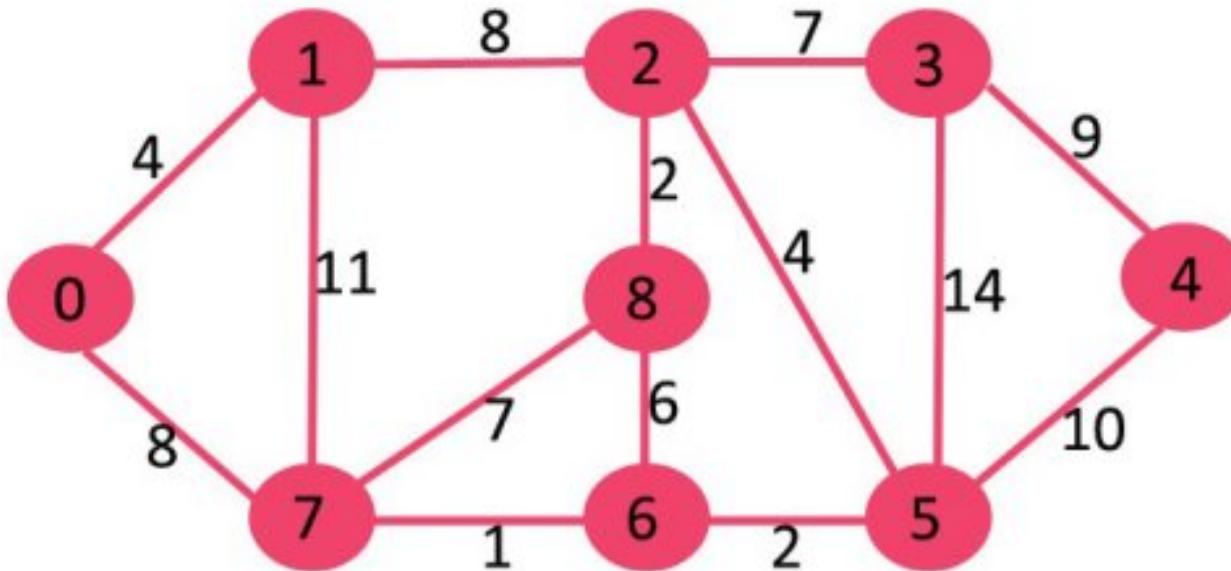
→ lowest cost is 8 ∴ pop & push its neighbour



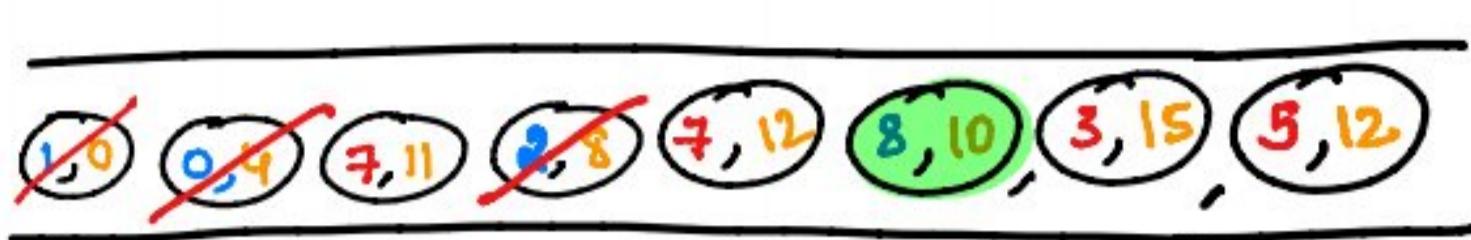
⇒ neighbours of 2 = 1 (visited), $(8, 10)$, $(3, 15)$, $(5, 12)$ ∴ push



→ lowest cost is 10 ∴ pop & push its neighbour



⇒



⇒ neighbours of 8 = 2 (visited), 7, 17, 6, 16
∴ push

$$\text{vis} = \{1, 0, 2, 8\}$$

$$\text{cost}[8] = 10$$



↳ lowest cost = 11 ∴ pop & push its neighbours.

⇒

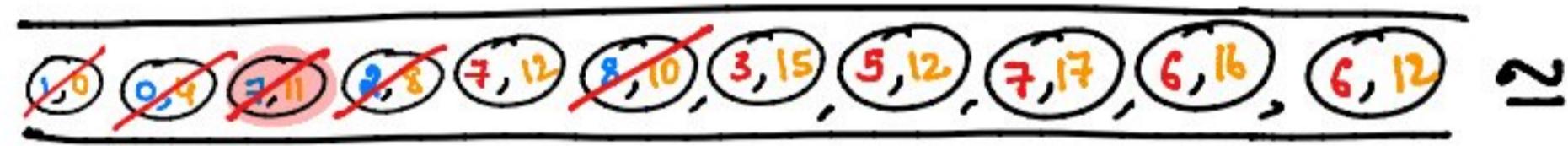


⇒ neighbours of 7 = 0, 1, 8 are visited.

ε 6, 12 ∴ push

$$\text{vis} = \{1, 0, 2, 8, 7\}$$

$$\text{cost}[7] = 11$$



↳ lowest cost = 12

∴ Anything among 5, 6 can be selected ε, pop & push its neighbours
Not 7, because it is already visited & cost is < 12.

⇒



⇒ neighbours of 5 = 4, 22, 3, 26, 6, 14 ∴ push

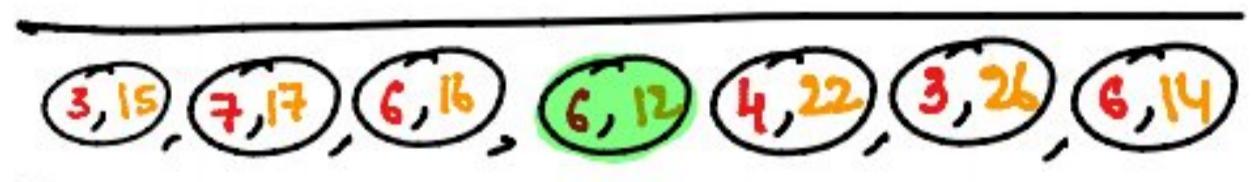
$$\text{vis} = \{1, 0, 2, 8, 7, 5\}$$

$$\text{cost}[5] = 12$$



→ lowest cost = 12
∴ pop & push its neighbours

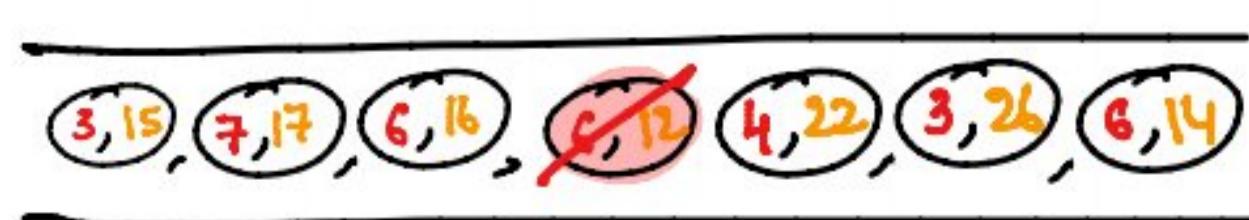
⇒



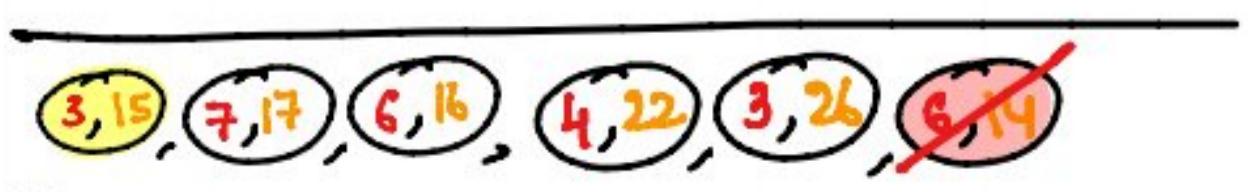
⇒ neighbours of 6 = 5, 7, 8 are visited.
∴ no push

$$\text{vis} = \{1, 0, 2, 8, 7, 5, 6\}$$

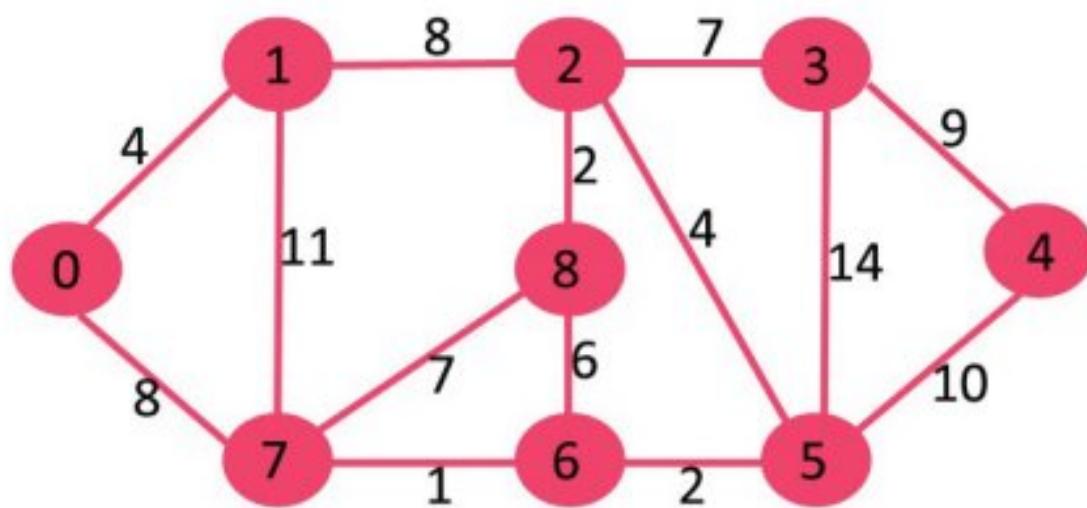
$$\text{cost}[6] = 12$$



→ next lowest is 14, but 6 is already visited.



∴ Next lowest is 15, ∴ pop & push its neighbours



\Rightarrow $(3, 15), (7, 17), (6, 16), (4, 22), (3, 24)$ \Rightarrow neighbours of 3 = 2, 5 (visited), $(4, 24)$ \therefore push

$$vis = \{1, 0, 2, 8, 7, 5, 6, 3\}$$

$$cost[3] = 15$$

$(3, 15), (7, 17), (6, 16), (4, 22), (3, 24), (4, 24)$ \rightarrow next lowest cost = 16
but 6 is already visited \therefore pop

$(7, 17), (6, 16), (4, 22), (3, 24), (4, 24)$

\Rightarrow

$(4, 22), (3, 24), (4, 24)$

\rightarrow next lowest cost = 22

\therefore pop q & push its neighbour.

\rightarrow next lowest cost = 17
but 7 is already visited \therefore pop

\Rightarrow

$(4, 22), (3, 24), (4, 24)$

\Rightarrow neighbours of 4 = 3, 5 (visited) \therefore no push

$$vis = \{1, 0, 2, 8, 7, 5, 6, 3, 4\}$$

$$cost[4] = 22$$

$(4, 22), (3, 24), (4, 24)$

\rightarrow next lowest cost = 24
but 4 is already visited
 \therefore pop

$(3, 24), (4, 24)$

\rightarrow next lowest cost = 26
but 3 is already visited
 \therefore pop

$(3, 24)$

\Rightarrow

\quad

\therefore empty PQ

answer \Rightarrow

$$wst = \boxed{4 \mid 0 \mid 8 \mid 15 \mid 22 \mid 12 \mid 12 \mid 11 \mid 10} \\ 0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8$$

Dijkstra's = BFS + PQ

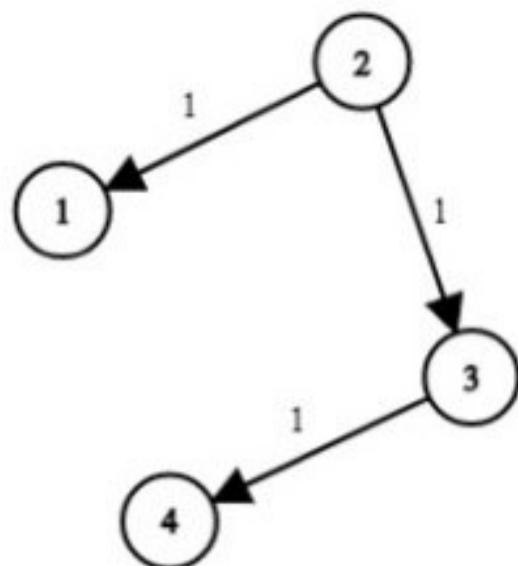
$T_C \rightarrow O(V + E \log V)$

$S_C \rightarrow O(V)$

Code →

```
● ● ●  
1 class Solution  
2 {  
3     public:  
4         vector <int> dijkstra(int V, vector<vector<int>> adj[], int src)  
5     {  
6         vector<int>cost(V,0);  
7         cost[src]=0;  
8  
9         vector<bool>vis(V, false);  
10        priority_queue<pair<int,int>,vector<pair<int,int>>,greater<pair<int,int>>> pq;  
11  
12        pq.push({0,src}); // {cost, node}  
13  
14        while(!pq.empty())  
15        {  
16            pair<int,int>p = pq.top();  
17            int currCost = p.first;  
18            int currNode = p.second;  
19            pq.pop();  
20  
21            if(vis[currNode])    continue;  
22  
23            vis[currNode] = true;  
24            cost[currNode] = currCost;  
25  
26            for(int i=0;i<adj[currNode].size();i++)  
27            {  
28                int neighbourNode = adj[currNode][i][0];  
29                int weight = adj[currNode][i][1];  
30                // if already visited then skip  
31                if(vis[neighbourNode])    continue;  
32                // else push  
33                pq.push({currCost + weight, neighbourNode});  
34            }  
35        }  
36        return cost;  
37    }  
38};  
39
```

12 Network Delay Time



You are given a network of n nodes, labeled from 1 to n . You are also given times, a list of travel times as directed edges times[i] = (u_i , v_i , w_i), where u_i is the source node, v_i is the target node, and w_i is the time it takes for a signal to travel from source to target.

We will send a signal from a given node k . Return the time it takes for all the n nodes to receive the signal. If it is impossible for all the n nodes to receive the signal, return -1.

$\text{src} = 2$.

✓ Similar to dijkstra's algo. cost =

0	0	0	0	0
0	1	2	3	4

 vis = { } pq = _____

⇒ push $(2, 0)$ to pq. ⇒

$(2, 0)$

⇒

$(2, 0)$

 neighbours = $(1, 1)$, $(3, 1)$ ∴ push
 $\text{vis} = \{2\}$ $\text{cost}[2] = 0$

$(2, 0)$	$(1, 1)$	$(3, 1)$
_____	_____	_____

 → next lowest cost = 1 ∴ choose 1 or 3
 pop & push their neighbour.

⇒

$(1, 1)$	$(3, 1)$
_____	_____

 no new neighbours ∴ pop
 $\text{vis} = \{2, 1\}$ $\text{cost}[1] = 1$

$(1, 1)$	$(3, 1)$
_____	_____

 → next lowest cost = 1
 ∴ pop & push their neighbour.

⇒

$(3, 1)$

 neighbour = $(4, 2)$ ∴ push
 $\text{vis} = \{2, 1, 3\}$ $\text{cost}[3] = 1$

$(3, 1)$	$(4, 2)$
_____	_____

 → next lowest cost = 2
 ∴ pop & push neighbours.

⇒

$(4, 2)$

 no new neighbours ∴ pop
 $\text{vis} = \{2, 1, 3, 4\}$ $\text{cost}[4] = 2$

$(4, 2)$

 → pq is empty.

∴ cost =

0	1	0	1	2
0	1	2	3	4

Tc $\rightarrow O(V + E \log V)$
 Sc $\rightarrow O(V)$

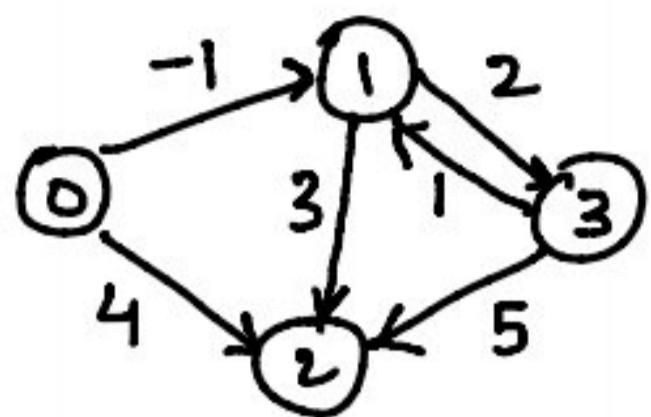
→ check if all nodes are in visited,
 else return -1.
 → Return max value in cost as

Code →

```
1  class Solution {
2  public:
3
4      int networkDelayTime(vector<vector<int>>& times, int n, int k) {
5          vector<vector<vector<int>>> graph = createGraph(times,n);
6          return minTime(graph,n,k);
7      }
8
9      vector<vector<vector<int>>> createGraph(vector<vector<int>>& edges,int n) {
10
11         vector<vector<vector<int>>> graph(n+1);
12
13         for(int i=0;i<=n;i++) {
14             graph.push_back({{}});
15         }
16         // add every edge to the graph
17         for(vector<int> edge:edges) {
18             int source = edge[0];
19             int dest = edge[1];
20             int cost = edge[2];
21             graph[source].push_back({dest,cost});
22         }
23         return graph;
24     }
25
26     int minTime(vector<vector<vector<int>>> &graph,int n,int src) {
27
28         vector<int> cost(n+1,0);
29         cost[src] = 0;
30         vector<bool>vis(n+1, false);
31
32         priority_queue<pair<int,int>,vector<pair<int,int>>,greater<pair<int,int>>>pq;
33         pq.push({0,src}); // {cost, node}
34
35         while(!pq.empty()) {
36             pair<int,int>p = pq.top();
37             int currNode = p.second;
38             int currCost = p.first;
39             pq.pop();
40             // if already visited then skip
41             if(vis[currNode]) continue;
42
43             vis[currNode] = true;
44             cost[currNode] = currCost;
45
46             for(int i=0;i<graph[currNode].size();i++)
47             {
48                 int neighbourNode = graph[currNode][i][0];
49                 int weight = graph[currNode][i][1];
50                 // if already visited then skip
51                 if(vis[neighbourNode]) continue;
52                 // else push into pq
53                 pq.push({currCost + weight, neighbourNode});
54             }
55         }
56
57         for(int i=1; i<=n; i++)
58             if(vis[i]==0) return -1;
59
60         int ans = 0;
61         for(int x:cost) ans = max(ans,x);
62         return ans;
63     }
64 }
```

⑬ Bellman Ford Algorithm → useful when weights < 0 (Dijkstra fails)
 ↳ dp algo → useful when finding negative weight cycle.
 [src, dst, wt]

Ex n = 4 edges = $\{[0, 1, -1], [0, 2, 4], [1, 2, 3], [1, 3, 2], [3, 1, 1], [3, 2, 5]\}$



initially dist

inf	inf	inf	inf
0	1	2	3

⇒ dist[src] = 0 &

⇒ relax every edge n-1 time i.e run for loop & perform the following operation

$$\text{dist}[dst] = \min(\text{dist}[src] + \text{weight}, \text{dist}[dst])$$

⇒ finally relax one more time &

if $\text{dist}[dst] > \text{dist}[src] + \text{wt} \Rightarrow$ -ve weight cycle present

⇒ we should relax 3 times & src=0 ⇒ dist[src]=0 dist

0	inf	inf	inf
0	1	2	3

→ for edge $[0, 1, -1]$, $\text{dist}[1] = \min(0 + (-1), \text{inf}) = -1$

$[0, 2, 4]$, $\text{dist}[2] = \min(0 + 4, \text{inf}) = 4$

$[1, 2, 3]$, $\text{dist}[2] = \min(-1 + 3, 4) = 2$

$[1, 3, 2]$, $\text{dist}[3] = \min(-1 + 2, \text{inf}) = 1$

$[3, 1, 1]$, $\text{dist}[1] = \min(1 + 1, -1) = -1$

$[3, 2, 5]$, $\text{dist}[2] = \min(1 + 5, 2) = 2$.

$$\therefore \text{dist} = \begin{array}{|c|c|c|c|} \hline 0 & -1 & 2 & 1 \\ \hline 0 & 1 & 2 & 3 \\ \hline \end{array}$$

→ now use the above dist & perform same operation twice, in this case dist remains same.

→ during final relaxation, -ve weight cycle condition is not met.

Answer ⇒ dist =

0	-1	2	1
0	1	2	3

TC $\rightarrow O(V * E)$

SC $\rightarrow O(V)$

⑯ Negative weight cycle → Bellman Ford Algorithm.

→ To check the presence of negative weight cycle using Bellman Ford algorithm.

$$TC \rightarrow O(V * E)$$
$$SC \rightarrow O(V)$$

Code →

```
● ● ●  
1 class Solution {  
2 public:  
3     int isNegativeWeightCycle(int n, vector<vector<int>>edges){  
4         vector<int>dis(n, INT_MAX);  
5         // initially, dist to src is 0  
6         dis[0] = 0;  
7         // relax n-1 times  
8         for(int i=0;i<n-1;i++)  
9         {  
10             for(auto edge:edges)  
11             {  
12                 int src = edge[0];  
13                 int dest = edge[1];  
14                 int wt = edge[2];  
15                 if(dis[src]!=INT_MAX) // to avoid integer overflow  
16                     dis[dest] = min(dis[dest],dis[src]+wt);  
17             }  
18         }  
19         // final relaxation  
20         for(auto edge:edges)  
21         {  
22             int src = edge[0];  
23             int dest = edge[1];  
24             int wt = edge[2];  
25             if(dis[src]!=INT_MAX && dis[dest]>dis[src]+wt)  
26                 return 1;  
27         }  
28         return 0;  
29     }  
30 };
```

(15) Floyd Warshall Algorithm

- All source shortest path & -ve edges allowed.
- Since it's all source shortest path we need to run the loop for all nodes, considering it as intermediate vertex.
- $\text{cost}[i][j] = \min(\text{cost}[i][j], \text{cost}[i][k] + \text{cost}[k][j])$

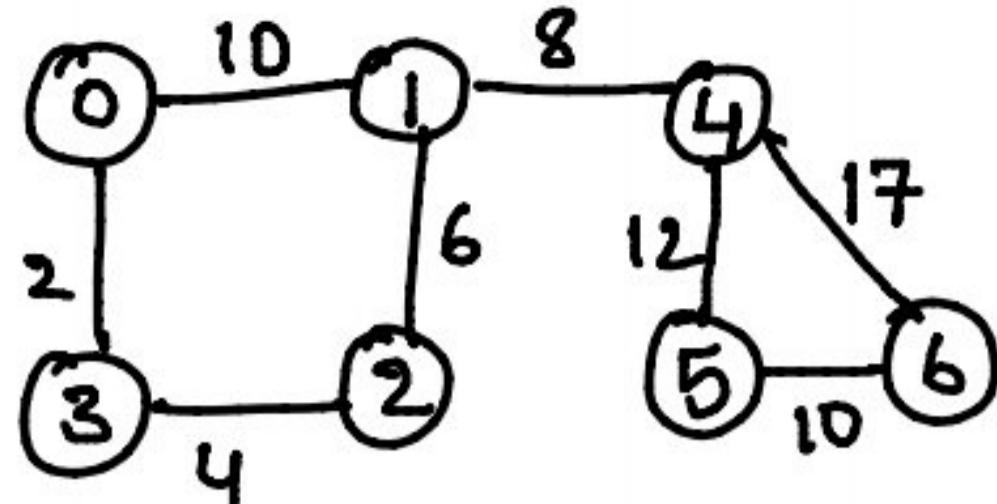
$$TC \rightarrow O(N^3) \quad SC \rightarrow O(N^2)$$

Code →

```
● ● ●  
1 class Solution {  
2     public:  
3         void shortest_distance(vector<vector<int>>&matrix){  
4             int V = matrix.size();  
5             vector<vector<int>> costs(matrix.size(), vector<int>(matrix.size()));  
6  
7             for(int i=0;i<V;i++)  
8                 for(int j=0;j<V;j++)  
9                     costs[i][j] = matrix[i][j];  
10  
11            for(int k=0;k<V;k++)  
12                for(int i=0;i<V;i++)  
13                    for(int j=0;j<V;j++){  
14                        // if intermediate is not -1 then  
15                        if(costs[i][k]!=-1 && costs[k][j]!=-1){  
16                            if(costs[i][j]==-1)  
17                                costs[i][j] = costs[i][k]+costs[k][j];  
18                            else  
19                                costs[i][j] = min(costs[i][j], costs[i][k]+costs[k][j]);  
20                        }  
21                    }  
22  
23                    for(int i=0;i<V;i++)  
24                        for(int j=0;j<V;j++)  
25                            matrix[i][j] = costs[i][j];  
26  
27            }  
28        };
```

⑯ Prim's Algorithm → Minimum Spanning Tree (MST)

Eg

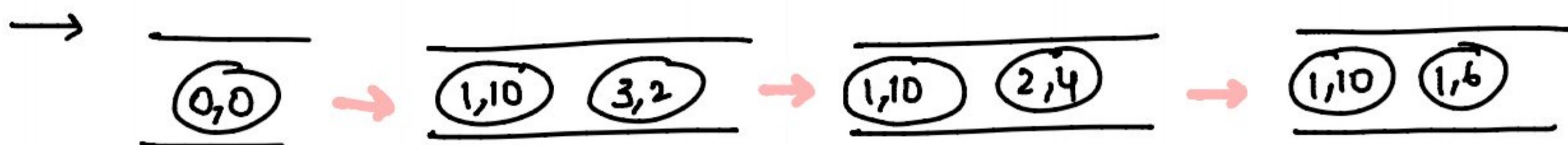


$$Vis = \{ \}$$

PQ $\underline{\quad}$
node, weight

↑ returns node
with lowest cost/weight.

* To find MST, just push node along with its weight.

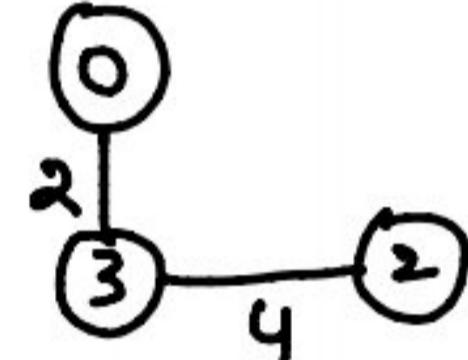


$$Vis = \{ \}$$

$$Vis = \{ 0 \}$$

$$Vis = \{ 0, 3 \}$$

$$Vis = \{ 0, 3, 2 \}$$



$$\underline{(1,10) \quad (4,8)}$$

$$Vis = \{ 0, 3, 2, 1 \}$$

$$\underline{(1,10) \quad (5,12) \quad (6,17)}$$

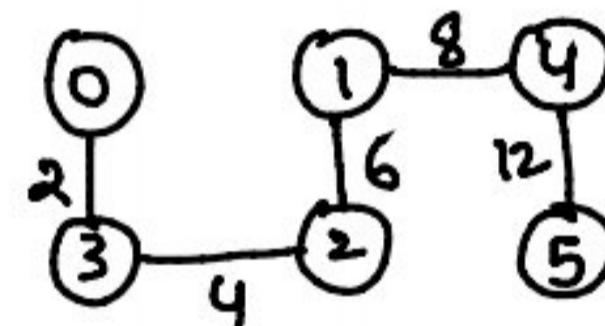
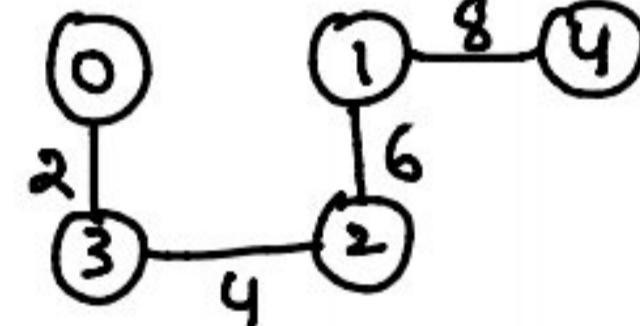
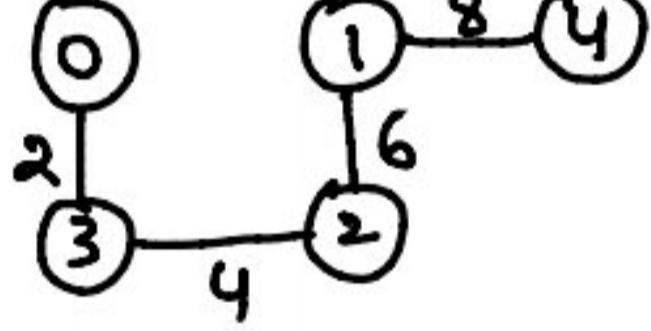
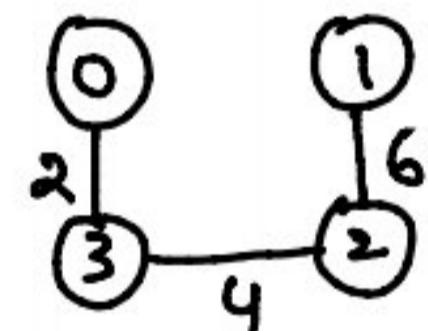
$$Vis = \{ 0, 3, 2, 1, 4 \}$$

$$\underline{(5,12) \quad (6,17)}$$

$$Vis = \{ 0, 3, 2, 1, 4 \}$$

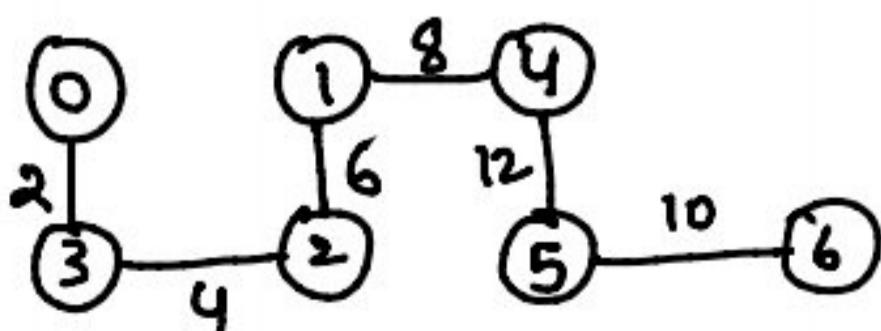
$$\underline{(6,17) \quad (6,10)}$$

$$Vis = \{ 0, 3, 2, 1, 4, 5 \}$$



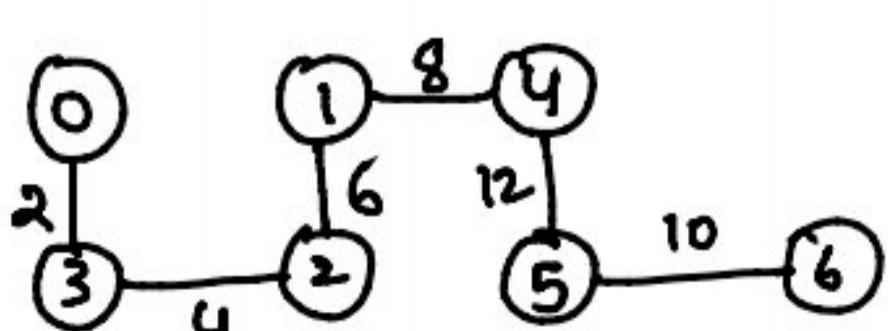
$$\underline{(6,17)}$$

$$Vis = \{ 0, 3, 2, 1, 4, 5, 6 \}$$



$$\underline{\quad}$$

$$Vis = \{ 0, 3, 2, 1, 4, 5, 6 \}$$



$$TC \rightarrow O(V + E \log V)$$

$$SC \rightarrow O(V)$$

Code →

```
1 class Solution
2 {
3     public:
4     //Function to find sum of weights of edges of the Minimum Spanning Tree.
5     int spanningTree(int V, vector<vector<int>> adj[])
6     {
7         int minCost = 0;
8         vector<int> costs(V, INT_MAX);
9         costs[0] = 0;
10        vector<bool> vis(V, false);
11        priority_queue<pair<int,int>, vector<pair<int,int>>, greater<pair<int,int>>>pq;
12        pq.push({0,0}); // {cost, Node}
13
14        while(!pq.empty())
15        {
16            pair<int,int> p = pq.top();
17            int currNode = p.second;
18            int currCost = p.first;
19            pq.pop();
20
21            if(vis[currNode])    continue;
22
23            minCost += currCost;
24
25            vis[currNode] = true;
26            costs[currNode] = currCost;
27
28            for(int i=0;i<adj[currNode].size();i++)
29            {
30                int neighbourNode = adj[currNode][i][0];
31                int neighbourNodeCost = adj[currNode][i][1];
32                if(vis[neighbourNode])  continue;
33                pq.push({neighbourNodeCost, neighbourNode});
34            }
35        }
36        return minCost;
37    }
38 };
39
```

17) Min cost to connect all points

→ Create graph with each node containing Wt & Node value

$$Wt = \text{abs}(X_i - X) + \text{abs}(Y_i - Y)$$

→ Perform Prims algo.

$$TC \rightarrow O(V + E \log V)$$

$$SC \rightarrow O(V)$$

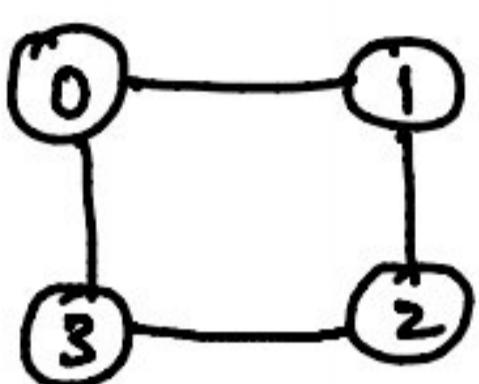
Code →

```
● ● ●  
1 2  class Solution {  
3  public:  
4      int minCostConnectPoints(vector<vector<int>>& points) {  
5  
6          int n = points.size();  
7          vector<vector<pair<int, int>>> graph(n);  
8  
9          for (int i = 0; i < n; i++) {  
10             for (int j = 0; j < n; j++) {  
11                 if (i == j) continue;  
12                 graph[i].push_back({abs(points[i][0] - points[j][0]) + abs(points[i][1] - points[j][1]), j});  
13             }  
14         }  
15  
16         priority_queue<pair<int,int>, vector<pair<int,int>>, greater<pair<int,int>>> pq;  
17         vector<bool> vis(n, false);  
18         pq.push({0, 0}); // {cost, Node}  
19  
20         int ans = 0;  
21         while (!pq.empty())  
22         {  
23             pair<int,int> p = pq.top();  
24             int currNode = p.second;  
25             int currCost = p.first;  
26             pq.pop();  
27  
28             if (vis[currNode]) continue;  
29             ans += currCost;  
30             vis[currNode] = true;  
31  
32             for(int i=0;i<graph[currNode].size();i++)  
33             {  
34                 int neighbourNode = graph[currNode][i].second;  
35                 int neighbourNodeCost = graph[currNode][i].first;  
36                 if(vis[neighbourNode]) continue;  
37                 pq.push({neighbourNodeCost, neighbourNode});  
38             }  
39         }  
40     }  
41     return ans;  
42 }  
43 };  
44 }
```

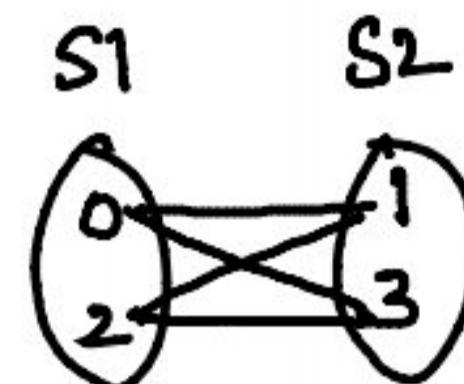
⑯ Is graph Bipartite

Bipartite graph is undirected graph, such that all vertices can be divided into 2 sets, S_1 & S_2 and no two vertices present in same set share an edge.

Eg $n = 4$



thus

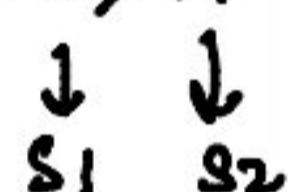


\therefore the graph is bipartite.

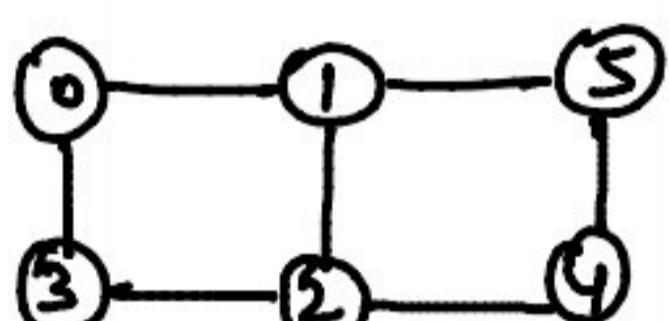
\Rightarrow for graph to be bipartite,

- it needs to be undirected acyclic graph (or)
- it needs to be even length cyclic graph

\rightarrow we generally denote sets by coloring it, color = 0, 1.



Eg $n = 6$



$$vis = \{3\} \quad S_1 = \{3\} \quad S_2 = \{3\}$$

initially color

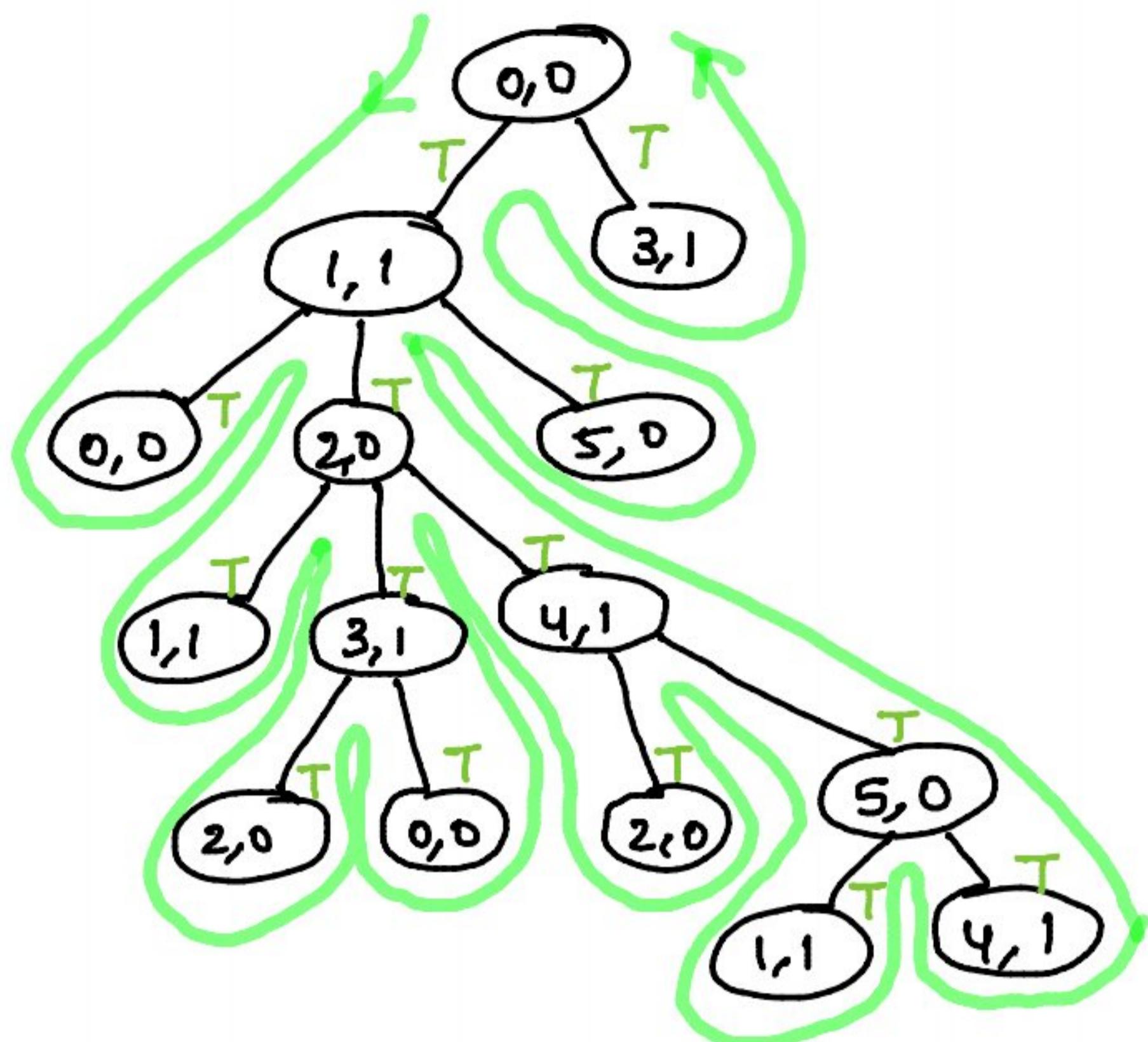
-1	-1	-1	-1	-1	-1
0	1	2	3	4	5

\rightarrow at each vertex, check if it visited or not.

\rightarrow if visited then check if it present in the intended set or not.

\rightarrow if yes then return true, else false

\rightarrow return AND of all the boolean values.



Code →

```
1 class Solution {
2 public:
3
4     bool isBipartite(vector<vector<int>>& graph) {
5
6         int n= graph.size();
7         vector<int>colors(n,-1);
8
9         for(int curr=0; curr<n ; curr++){
10             // if already colored then skip
11             if(colors[curr]!=-1)    continue;
12             // check for even length cycle
13             if(hasEvenLengthCycle(graph, curr, 0, colors)==false)  return false;
14         }
15         return true;
16     }
17
18     bool hasEvenLengthCycle(vector<vector<int>>& graph,int curr,int color,vector<int>&colors)
19     {
20         if(colors[curr]!=-1)
21             return colors[curr]==color;
22
23         // if not colored then color it
24         colors[curr] = color;
25
26         // check for neighbours
27         for(int neigh: graph[curr])
28         {
29             if(hasEvenLengthCycle(graph, neigh, 1-color, colors)==false)
30                 // 1- color will handle both changing colors 0 to 1 and 1 to 0
31                 return false;
32         }
33         return true;
34     }
35
36 };
```

19 Possible Bipartition →

- Create a graph using dislikes array.
- use previous problem's approach to solve it.

Code →

TC → O(V+E) SC → O(V+E)

```
● ● ●  
1 class Solution {  
2 public:  
3  
4     bool dfs(vector<int> graph[], int curr, vector<int>& color){  
5  
6         // if not colored then color  
7         if(color[curr] == -1)  
8             color[curr] = 1;  
9  
10        // process the neighbours and check their colors  
11        for(auto neigh : graph[curr])  
12        {  
13            if(color[neigh] == -1)  
14            {  
15                color[neigh] = 1 - color[curr];  
16                if(dfs(graph, neigh, color)==false) return false;  
17            }  
18            else if(color[neigh] == color[curr]) return false;  
19        }  
20        return true;  
21    }  
22  
23    bool possibleBipartition(int n, vector<vector<int>>& dislikes) {  
24        vector<int> color(n+1, -1);  
25        vector<int> graph[n+1];  
26  
27        // populating the graph  
28        for(auto edge : dislikes){  
29            graph[edge[0]].push_back(edge[1]);  
30            graph[edge[1]].push_back(edge[0]);  
31        }  
32  
33        for(int i=1; i<=n; i++){  
34            if(color[i] == -1)  
35                if(!dfs(graph, i, color)) return false;  
36        }  
37  
38        return true;  
39    }  
40};
```

20 Disjoint set \rightarrow UNION & FIND./getParent
 ↴ helps in finding parent of component
 helps in UNION of components/vertices.

Eg $0 \ 1 \Rightarrow \text{UNION}(0,1) \rightarrow 0 - 1$

Eg $n=7$ initially every component is parent of itself



parent =	[0	1	2	3	4	5	6]
		0	1	2	3	4	5	6	

now $\text{getParent}(2) = 2, \text{getParent}(3) = 3.$

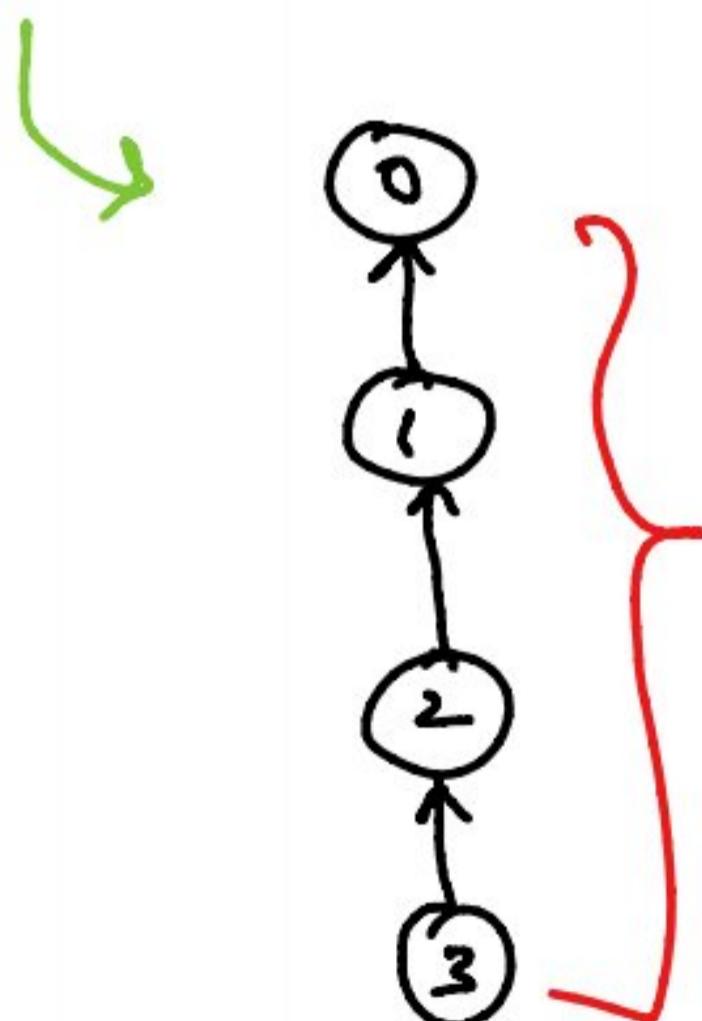
if $\text{UNION}(0,1) \Rightarrow 0 - 1$ & $\text{parent}[1] = 0$

now $\text{getParent}(1) = 0$

if $\text{UNION}(1,2) \Rightarrow 0 - 1 - 2$

$\text{UNION}(2,3) \Rightarrow 0 - 1 - 2 - 3$

if $\text{getParent}(3) = 0$



This increases the recursive calls and the tree is unbalanced so we'll use rank array to store min. height tree for node.

$n=7$ initially every component is parent of itself



parent =

0	1	2	3	4	5	6
0	1	2	3	4	5	6

rank =

0	0	0	0	0	0	0
0	1	2	3	4	5	6

$\Rightarrow \text{UNION}(0,1) \Rightarrow$ then $\text{find}(0) \neq \text{find}(1) \neq 0 \neq 1 \therefore$ diff components.
as they are diff components find rank & $\text{rank}[0] = \text{rank}[1] = 0$

\therefore select either 0 or 1 & make it as root & inc the rank by 1



parent =

0	0	2	3	4	5	6
0	1	2	3	4	5	6

rank =

1	0	0	0	0	0	0
0	1	2	3	4	5	6

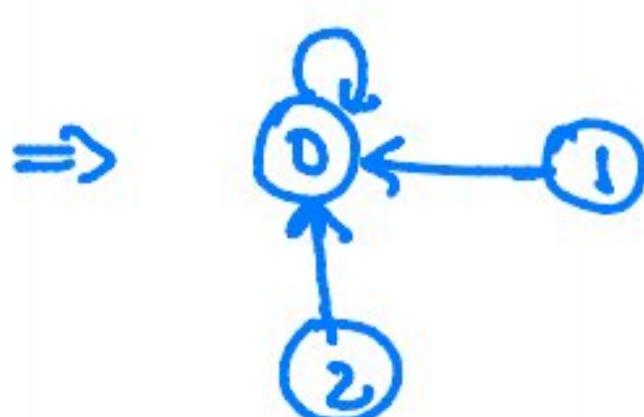
$\Rightarrow \text{UNION}(1,2) \Rightarrow \text{parent}(1)=0 \neq \text{parent}(2)=2$

now $\text{rank}[0]=1 \neq \text{rank}[2]=0$

as $\text{rank}[0] > \text{rank}[2]$,

vertex 0 should be the parent

& do not update rank if they are unequal.



parent =

0	0	0	3	4	5	6
0	1	2	3	4	5	6

rank =

1	0	0	0	0	0	0
0	1	2	3	4	5	6

Code →

```
1  class DisjSet {
2      int *rank, *parent, n;
3
4  public:
5      DisjSet(int n)
6      {
7          rank = new int[n];
8          parent = new int[n];
9          this->n = n;
10         makeSet();
11     }
12
13     void makeSet()
14     {
15         for (int i = 0; i < n; i++) {
16             parent[i] = i;
17         }
18     }
19
20     int find(int x)
21     {
22         // if x is not parent of itself then
23         // find parent recursively
24         if (parent[x] != x) {
25             parent[x] = find(parent[x]);
26         }
27         return parent[x];
28     }
29
30     void Union(int x, int y)
31     {
32         int xset = find(x);
33         int yset = find(y);
34
35         // if set of x and y are same then return
36         if (xset == yset)    return;
37
38         // place the elements in small rank
39         if (rank[xset] < rank[yset]) {
40             parent[xset] = yset;
41         }
42         else if (rank[xset] > rank[yset]) {
43             parent[yset] = xset;
44         }
45         // if same rank then increment it
46         else {
47             parent[yset] = xset;
48             rank[xset] = rank[xset] + 1;
49         }
50     }
51 };
52
```

(21)

Kruskal's Algorithm

- This is used to find minimum spanning tree.
- can be implemented using Disjoint set.
- sort all the edges in ↑ order of weight.
- pick smallest edge & check if it contributes to cycle in graph
- if yes then discard else include.

Code →

```

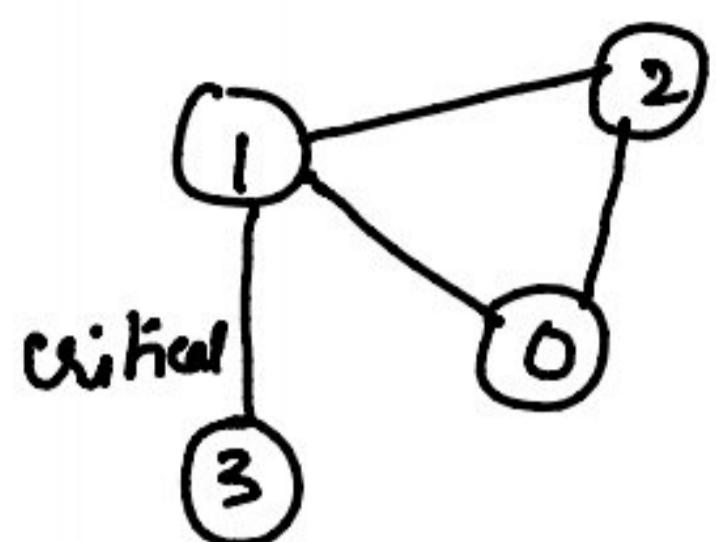
● ● ●

1 class Graph {
2     vector<vector<int>> edgelist;
3     int V;
4
5 public:
6     Graph(int V) { this->V = V; }
7
8     void addEdge(int x, int y, int w)
9     {
10         edgelist.push_back({ w, x, y });
11     }
12
13     void kruskals_mst()
14     {
15         // 1. Sort all edges
16         sort(edgelist.begin(), edgelist.end());
17
18         // Initialize the DSU - DisjointSet
19         DSU s(V);
20         int ans = 0;
21         for (auto edge : edgelist) {
22             int w = edge[0];
23             int x = edge[1];
24             int y = edge[2];
25             // take that edge in MST if it does form a cycle
26             if (s.find(x) != s.find(y)) {
27                 s.union(x, y);
28                 ans += w;
29                 cout << x << " -- " << y << " == " << w
30                 << endl;
31             }
32         }
33         cout << "Minimum Cost Spanning Tree: " << ans;
34     }
35 };

```

22 Critical Connection in a Network

Eg $n=4$ edges = $[[0, 1], [1, 2], [2, 0], [1, 3]]$



→ Critical connection is a connection, when removed from graph, would result in breaking graph into different components.

Here if $[1, 3]$ is removed then graph becomes disconnected.

Approach 1

- Remove one edge each time
- Perform dfs
- If all vertices are not visited then
- Removed edge is a critical connection.

Approach 2

- Initialise distime array & mintime array with -1.
- discovery time for vertex → min time for vertex to be discovered.
- perform dfs from one node
 - if neighbour == parent then continue
 - else if neighbour is already visited then
 $\text{mintime}[\text{curr}] = \min(\text{mintime}[\text{curr}], \text{distime}[\text{neigh}])$
 - while returning $\text{mintime}[\text{curr}] = \min(\text{mintime}[\text{curr}], \text{mintime}[\text{neigh}])$
 - & at any point if $\text{distime}[\text{curr}] < \text{mintime}[\text{neigh}]$
- This indicates critical connection

Code →

```
1 class Solution {
2 public:
3
4     vector<vector<int>> criticalConnections(int n, vector<vector<int>>& connections) {
5         vector<int> graph[n];
6         for(vector<int> edge: connections){
7             int u = edge[0];
8             int v = edge[1];
9             graph[u].push_back(v);
10            graph[v].push_back(u);
11        }
12        return findCriticalConnections(n, graph);
13    }
14
15    vector<vector<int>> findCriticalConnections(int n, vector<int> graph[]){
16        vector<int> disTime(n,-1);
17        vector<int> lowTime(n,-1);
18        int time = 0;
19        vector<vector<int>> answer;
20        tarjansDFS(graph, 0, -1, disTime, lowTime, time, answer);
21        return answer;
22    }
23
24    void tarjansDFS(vector<int> graph[], int curr, int parent, vector<int>&disTime,
25    vector<int> &lowTime, int &time, vector<vector<int>> &answer){
26
27        disTime[curr] = time;
28        lowTime[curr] = time;
29        time += 1;
30
31        for(int neigh: graph[curr]){
32            if(neigh == parent) continue;
33
34            if(disTime[neigh]!=-1){
35                lowTime[curr] = min(lowTime[curr], disTime[neigh]);
36                continue;
37            }
38
39            tarjansDFS(graph, neigh, curr, disTime, lowTime, time, answer);
40            lowTime[curr] = min(lowTime[curr], lowTime[neigh]);
41
42            if(disTime[curr] < lowTime[neigh]){
43                vector<int> temp;
44                temp.push_back(curr);
45                temp.push_back(neigh);
46                answer.push_back(temp);
47            }
48        }
49        return;
50    }
51}
52};
```

